Nuno António de Lira Fernandes Faria dos Santos

An Agile Process for Modeling Logical Architectures: Demonstration Cases from Large-scale Software Projects

Universidade do Minho
Escola de Engenharia

Nuno António de Lira Fernandes Faria dos Santos

An Agile Process for Modeling Logical Architectures: Demonstration Cases from Large-scale Software Projects

December 2020

Funding:

The ISOFIN project was developed within Quadro de Referência Estratégico Nacional, under reference QREN 2010/013837.

iFloW was part of "*HMIEXCEL - I&D crítica em torno do ciclo de desenvolvimento e produção de soluções multimédia avançadas para automóvel*" program under Bosch | UMinho consortium, funded through Sistema de Incentivos à Investigação e Desenvolvimento Tecnológico do QREN, under reference nº 36265/2013.

UH4SP: Unified Hub 4 Smart Plants (Project ID 017871) was developed through Portuguese National Grants Program for R&D projects (P2020 – SI IDT) and through COMPETE: POCI-01-0145-FEDER-007043

IMP_4.0: Integrated Management Plaftorm 4.0 project was developed through POCI-01-0247-FEDER-009147, under Portuguese National Grants Program for R&D projects (P2020 – SI IDT)

FCT – Fundação para a Ciência e Tecnologia within the Project Scope: UID/CEC/00319/2019

Universidade do Minho
Escola de Engenharia

Nuno António de Lira Fernandes Faria dos Santos

An Agile Process for Modeling Logical Architectures: Demonstration Cases from Large-scale Software Projects

Doctoral Thesis
Doctoral Program in Information Systems and Technology

Work done under the guidance of
Prof. Dr. Ricardo J. Machado
Dr. Nuno C. Ferreira

December 2020

Acknowledgements

To my wife Mónica, with her never-ending support and understanding of what this personal project meant to me - and its impact in our lives throughout these years. To my boy Duarte, as many times I was lifting him up when we were playing, he was the one actually lifting me up. To my brothers, Zeza and Tiago, for their unconditional support throughout my life. To my father António, who taught me the values from which I guide until today. To my mother Maria Fernanda, always on my mind pushing me towards my goals.

To my supervisors, Prof. Dr. Ricardo J. Machado and Dr. Nuno Ferreira, for the advices. I started my career with them, for four years even before this adventure started. Thank you both for guiding me the way but letting me figure it out on my own, never closing me the paths I suggested.

To my employers, CCG\ZGDV Institute, for providing me the conditions for doing what I love. To everybody involved within the EPMQ lab: IT Engineering Process, Maturity and Quality. To my Director, Ana Lima, for always supporting my research work. Specially, to Francisco Morais, Jaime Pereira, Prof. Dr. Helena Rodrigues, Dr. Carlos Salgado, Prof. Marcelo Noguera for the insightful discussions and debates during the research work and research projects we performed together. But also, a big appreciation to Isabel Varajão, Cláudia Amaro, Rui Costa, João Peixoto, and many, many others.

Dedicated...

To my beloved family.

To my wife, Mónica.

To my son, Duarte.

To my brothers, Zeza and Tiago.

To my father, António.

In the loving memory of my mother, Maria Fernanda.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Abstract

Developing software solutions is about providing responses to a set of business needs in form of a working software. These business needs are the starting point for the development process, which states the required business support that the software will provide, in form of functional and non-functional requirements. Then, every implementation decision aims satisfying those requirements. Of course, over time, the requirements change. In that sense, agile software development (ASD) approaches bring a set of practices towards embracing those changes as soon as possible. As the complexity of software increases, namely in terms of the quantity of the defined features, these approaches face some challenges, typically related in properly defining how different teams have to work together in order to deliver a unified solution. In complex solutions, the traceability between business (or stakeholder) and software (components) perspectives may not be properly assured. Using a logical architecture provides a view that organizes software components in order to meet functional requirements. In a large-scale agile (LSA) setting, the logical architecture provides a view in how different teams' outputs fit together. Thus, this thesis presents a process for modeling logical architectures adequate for ASD settings (a.k.a., "*Agile Modeling*" – AM) with requirements elicitation and modeling techniques and, additionally, uses an architectural design method called "*Four Step Rule Set*" (4SRS) in order to trace the requirements models to the architectural components. An AM process should support evolutionary design, preventing the so-called "*Big Design Up Front*" (BDUF) with eventual efforts that are afterwards disregarded in "*You Aren't Going to Need It*" (YAGNI) elements. The proposed process is called "*Agile Modeling Process for Logical Architectures*" (AMPLA). This research work aimed defining how AMPLA covers the model evolution and abstraction level from business to service-oriented logical architectures in LSA settings. Additionally, adopting an architectural style called "microservices", eases the agility in developing (and deploying) the solutions, where its design principles promote continuous integration/delivery (CI/CD) and DevOps. Thus, AMPLA includes architecture modeling as well as maintenance and evolution during ASD iterations.

Keywords: agile software development, agile modeling, large-scale agile, logical architectures, microservices architecture

Resumo

Desenvolver soluções de software é fornecer respostas a um conjunto de necessidades de negócios na forma de um software executável. Essas necessidades de negócios são o ponto de partida para o processo de desenvolvimento, que define como o software suportará o negócio, na forma de requisitos funcionais e não funcionais. Então, qualquer decisão de implementação visa satisfazer esses requisitos. Obviamente, com o tempo, os requisitos mudam. Nesse sentido, as abordagens de desenvolvimento ágil de software (ASD) trazem um conjunto de práticas para abraçar essas mudanças o mais rápido possível. À medida que a complexidade do software aumenta, principalmente em termos da quantidade de recursos definidos, essas abordagens enfrentam alguns desafios, geralmente relacionados à definição correta de como equipas diferentes devem trabalhar em conjunto para fornecer uma solução unificada. Em soluções complexas, a rastreabilidade entre as perspetivas do negócio (ou dos *stakeholders*) e software (componentes) pode não estar adequadamente garantida. Usando uma arquitetura lógica, é fornecida uma visão que organiza os componentes de software para que os requisitos funcionais sejam suportados. Num contexto ágil de larga-escala (LSA), fornece uma visão de como os diferentes resultados se encaixam. Assim, esta tese apresenta um processo para modelar arquiteturas lógicas adequadas às configurações de ASD (também conhecido como "*Agile Modeling*" - AM), composto pelas técnicas de levantamento e modelação de requisitos e, adicionalmente, usa um método arquitetural chamado "*Four Step Rule Set*" (4SRS) para rastreabilidade entre os requisitos e os componentes arquiteturais. Um processo de AM deve oferecer suporte a uma conceção evolutiva, impedindo o chamado "*Big Design Up Front*" (BDUF), com eventuais esforços que serão posteriormente desconsiderados em elementos "*You Aren't Going to Need It*" (YAGNI). O processo proposto é chamado de "*Agile Modeling Process for Logical Architectures*" (AMPLA). Este trabalho de investigação pretendeu definir como o AMPLA cobre a evolução dos modelos e nível de abstração desde o negócio até a arquiteturas lógicas orientadas a serviços em contextos de LSA. Além disso, a adoção de um estilo arquitetural chamado "micro-serviços" facilita a agilidade no desenvolvimento (e instalação) das soluções, onde suas bases da conceção promovem a integração / entrega contínua (CI / CD) e cultura DevOps. Assim, o AMPLA inclui modelação da arquitetura, bem como manutenção e evolução durante ciclos ágeis.

Palavras-chave: ágil em larga escala, arquitetura de micro-serviços, arquiteturas lógicas, desenvolvimento ágil de software, modelação ágil.

Table of Contents

APPENDIXES

Acronyms

4SRS – Four Step Rule Set

4SRS-MSLA - Four Step Rule Set for microservices logical architecture

AAL - agile architecting lifecycle

AC – Acceptance Criteria

AM - Agile Modeling

AMPLA - Agile Modeling Process for Logical Architectures

ASD - agile software development

BDUF - "big design upfront"

CA - Continuous architecture

CI/CD - continuous integration/delivery

CIA - Change Impact Analysis

CRE - continuous requirements engineering

CSE - continuous software engineering

DAD - Disciplined Agile Delivery

DDD - Domain-driven Design

DoD – Definition of Done

DoR – Definition of Ready

DSR - Design Science Research

DT – Design theory

DUARTE - Decomposing User Agile Requirements ArTEfacts

iFloW - Inbound Logistics Tracking System project

IMP_4.0 - Integrated Management Platform 4.0 project

IMSPM - Internal Management System of Project Management project

ISOFIN - Interoperability in Financial Software project

IoT - internet of things

LeSS - Large-Scale Scrum

LSA - large-scale agile

MSA – microservices architecture

MSLA – microservices logical architecture

MTS - multiteams systems

MVP - minimum viable product

PBI - Product Backlog Items

RE - requirements engineering

SAFe - Scaled Agile Framework

SDLC - Software development lifecycle

SOA - service-oriented architectures

SoaML - Service oriented architecture Modeling Language

SISoS - Software-intensive system of systems

SoS - Scrum of Scrums

SRP - single responsibility principle

TBI - team backlog items

TDD - Test Driven Development ´

TPB - team product backlog

UH4SP - Unified Hub for Smart Plants project

UML – Unified Modeling Language

W2ReqComm – “What? and Why?” Requirements Communication

XP - eXtreme Programming

YAGNI - "*You Aren't Gonna Need It*"

List of Figures

List of Tables

PART I

INTRODUCTION

Chapter 1 - Introduction

This chapter introduces the topic of the presented research for a proper understanding of this research. It describes the motivations for this thesis, the research question and objectives towards answering the question, and finally the research method.

Chapter 1 - Introduction

"*A problem well stated is a problem half solved.*"

Charles F. Kettering, inventor, engineer, businessman

1.1 Motivations

Software architecture design, when performed in context of agile software development (ASD), sometimes referred as "agile architecting", promotes the emerging and incremental design of the architectural artifact, in a sense of avoiding "big design upfront" (BDUF). Performing "agile architecting" is not always straightforward, mainly because the architecture has a required life cycle and each stage responds to different needs. There is a lack of a roadmap that guides agile architecting in an end-to-end approach (from business requirements to deployment).

The role of architecture and architects have been changing due to the adoption of agile software development (ASD) approaches. Although an initial misconception because popular ASD frameworks (Scrum, XP, Kanban, DSDM) did not explicitly include architectural artifacts or roles, this role has been emerging towards a balanced design and implementation as the architecture emerges throughout the process. The architect plays a role in upfront planning, storyboarding and backlogs, Sprints and working software stages of a project (Madison, 2010). More recently, agile scaling frameworks , like Disciplined Agile Delivery (DAD) (Scott Ambler & Lines, 2012), Large-Scale Scrum (LeSS) (Larman & Vodde, 2016), Scaled Agile Framework (SAFe) (Leffingwell, 2016), Scrum@Scale (Sutherland, 2018) and Nexus (K Schwaber, 2015), have been adopted in industry. The architect's role have been specified by actively and passively support agile teams by driving architectural initiatives, participating in architectural runways, harmonizing governance requirements, and ensuring technical alignment in solution contexts (Uludag, Kleehaus, Xu, & Matthes, 2017).

The plethora of agile practices relate to management (e.g., Sprints, Scrum ceremonies), development (e.g., pair programming, TDD, BDD, DevOps) or strategy (e.g., Lean Startup), but lack a comprehensive description on how its adoption influences requirements modeling. Agile software development (ASD) is currently the worldwide-adopted approach in software engineering. The mashup of agile practices and industry coins (e.g., Scrum, XP, MVP, DevOps, large-scale agile, Squads/Tribes, Management 3.0, and many others) cover all software and application lifecycle. Although none of this practices relate to requirements engineering (RE) discipline, or specifically to Agile modeling (AM) (S Ambler, 2002), performing this practices into an ASD process has direct implications on how RE practices are performed and how artifacts are built.

Stakeholders are crucial participants for eliciting requirements towards a new software solution. However, agreeing a common understanding among them is a complex task in a project's initial phase when solution requirements and design need to be refined and/or are unknown. Companies often strive to properly perform requirements engineering (RE) tasks in software solutions for complex ecosystems (mainly those related to the emergence of new paradigms like Cloud Computing and more recently Industry 4.0, Internet of Things (IoT), machine-to-machine, cyber-physical systems, etc.). The elicitation for the required functionalities regarding the adoption of these recent technologies typically ends up without consensus when technical decisions are required. This trend does not have yet mature references and standards that companies may blindly follow, so the product development results in refactoring efforts towards new architectural patterns.

Stakeholders must able to communicate in what way a future solution improves their business, by defining the product roadmap. A product roadmap is an initial high level project scope and direction (IIBA, 2017). Typically, a first release on a new product encompasses a product's subset able to address priority scenarios, previously identified in order to respond to market needs. In fact, many of these product releases are market-driven, where the release is deployed into the market so it is possible to get feedback from it, *i.e.*, a minimum viable product (MVP).

In plan-driven approaches (e.g., Waterfall), tasks related to RE discipline are traditionally managed in a phase separated in time from design and development. In change-driven approaches, like ASD, RE discipline – also called "*Agile RE*" – activities remain the same but are executed continuously (Grau & Lauenroth, 2014), and takes an iterative discovery approach (Cao & Ramesh, 2008). Elicitation, analysis, and validation are present in all ASD processes (Paetsch, Eberlein, & Maurer, 2003).

ASD widely use User Stories (Cohn, 2004) as items in the backlog for "reminders of a conversation" about a functionality. However, using only User Stories, without attached requirements specifications or models, may be insufficient to assure a common understanding, or, in case of multi-teams, to clearly define inter-systems interactions. Additionally, requirements modeling should prevent unnecessary efforts in "*You Aren't Gonna Need It*" (YAGNI) features, hence the need for an Agile Modeling (AM) (S Ambler, 2002) approach.

Applying AM should start by enabling a first iteration of requirements modeling, which is then the basis for further refinements, and later support discovery when they emerges, as the software increments are being delivered throughout the Sprints. The inception, like the pregame phase or Sprint zero in Scrum, aims providing a shared understanding of the project and the required information for the development phase. In the same line of reasoning, Ambler presents an evolution and emerge-oriented approach for using models in ASD, called "Agile Model-Driven Development" (AMDD) (SW Ambler, 2003), where the starting point is "just-enough" requirements and architecture, which are updated alongside Delivery Cycles phase.

The architecture should emerge gradually sprint after sprint, as a result of successive small refactoring (Abrahamsson, Babar, & Kruchten, 2010). However, adopting any ASD approach heavily depends on the project context (Philippe Kruchten, 2007). Typically contexts like size, large systems with a lack of architectural focus, novice teams, high constraint on some quality attribute, among others, have high risks in ASD projects (P Kruchten, 2013).

In order to ease architectural management incrementally as it emerges, the "power of small" (Erder & Pureur, 2015) supports continuous architecting activities. In this sense, microservices architectures (MSA) propose small and interconnected services. Developing such solutions faces several challenges beyond typical architecture and service design concerns, including service exposition (API), inter-service communication, and infrastructure deployment, among others. Designing microservices for a given business capability or domain, typically uses patterns such as Domain-driven Design (DDD) (Evans, 2004), single responsibility principle (SRP) or Conway's Law (Conway, 1968). However, microservice design often faces challenges related to database partition, the proper size of the microservice, inter-service communication and messaging, which are not addressed systematically by those patterns. By applying a modeling method in the process of designing a MSA, one may foresee issues on bounded contexts for microservices, namely intra-service behavior, interfaces and data models separation, and inter-service communication and messaging requirements (Newman, 2015).

This thesis presents *Agile Modeling Process for Logical Architectures* (AMPLA), an Agile Modeling (AM) oriented process composed by UML diagrams (Sequence, Use Cases and Component). AMPLA uses agile practices in order to deliver small increments (of a requirements package) and to promote continuous customer feedback. The proposed AM process also includes a candidate architecture and further requirements refinement in parallel with a software increment delivery. By eliciting a set of "just-enough" UML Use Cases, *i.e.*, that includes at least the core requirements information, it is proposed the use of a logical architecture derivation method, the *Four Step Rule Set* (4SRS). This approach is suitable in agile software development contexts, where the solution's architecture is unknown upfront.

AMPLA is an approach for supporting the emergence of a candidate (logical) architecture, rather than BDUF the architecture in an early phase. AMPLA includes the core known features within the initial phase and designs a logical architecture using a stepwise method, without refining information. The emerging characteristics of AMPLA are supported in four stages, two performed before development cycles or Sprints and two in parallel with ASD cycles: (1) eliciting a small set of high-level requirements; (2) deriving a candidate logical architecture; (3) define subsystems for refinement; and (4) refine requirements and the architecture regarding the subsystem in small cycles or Sprints.

Projects at large-scale have been adopting agile practices in order to optimize how a group of teams deliver software. Such adoption however has faced difficulties on how to assign work items, set boundaries, and address communication and coordination. Process management thus deals with work items that are dependent on each other, need for well-defined interfaces and shared understanding of the existing knowledge. We propose a framework, built upon a design theory, based on previously derived logical architectures to serve as the basis for the delivery of work items to distributed agile teams. The logical architecture derived from AMPLA, and other artefacts, support the identification of boundaries, dependencies and coordination needs. Although acknowledging the importance of architecture in managing inter-team processes in a '*large-scale agile development*' context, these approaches lack of a structured approach for using such information to manage the software delivery process. The term '*large-scale agile*' (LSA) has been used to describe agile development in everything from large teams to large multi-team projects to making use of principles of agile development in a whole organization (Dingsøyr & Moe, 2014). Models are about presenting an abstraction of reality towards a shared understanding of the problem, but a proper analysis allows depicting their input in assigning work, derive dependencies, and manage inter-team communication and coordination.

Accordingly, this research proposes an approach for designing a microservices-oriented logical architecture (MSLA), *i.e.*, a logical view (Philippe Kruchten, 1995) on the behavior of microservices and relationships between microservices. This approach uses UML use cases diagrams for domain modeling, which are further used as an input for designing a MSLA in an automated way, by using an adaptation of the 4SRS method. Each of these functionally decomposed UML use cases give origin to one or more components, which will then compose the microservices.

1.2 Key definitions

Agility

Agility is defined as the continuous readiness "to rapidly or inherently create change, proactively or reactively embrace change, and learn from change while contributing to perceived customer value (economy, quality, and simplicity), through its collective components and relationships with its environment" (Conboy, 2009). It is also "the ability of an organization to both create and respond to change in order to profit in a turbulent business environment" (Highsmith, 2002).

Agile Modeling

Agile modeling is the task of developing emerging model-based artifacts, related to requirements and design, properly performed under agile software development. The models emerge as a "just-in-time" need for further implementation.

Large-scale agile

The dimensions used to define a project as large-scale agile relate to number of involved teams, costs, code size and number of requirements. For scope definition of the work, large-scale agile is characterized by having more than one team, to have more team members than the numbers that are typically suggested, or when large quantities of user stories (or requirements) or lines of code are required.

Logical Architectures

A logical architecture is an abstraction view of functionality-based elements that support a system's functional requirements, relations between them and with external systems, embodying design decisions. It is typically represented as objects or object classes, or as components.

Evolutionary design

Evolutionary design means that the design of the system grows as the system is implemented. As the software solution evolves, the design changes.

1.3 Core Concepts and Definitions

Software Architectures

The concept of software architecture is regarded as a distinct discipline, however still tied closely to other disciplines and communities, such as software design (in general), software reuse, systems engineering and system architecture, enterprise architecture, reverse engineering, requirements engineering, and quality (Philippe Kruchten, Obbink, & Stafford, 2006). So, why is software architecture so important in requirements engineering? Literature on software architecture encompasses a plethora of definitions. Most agree that an architecture concerns both structure and behavior (Philippe Kruchten et al., 2006), and is used for capturing key software system structural characteristics (Shaw & Garlan, 1996). Thus, what is software architecture?

The IEEE Recommended Practice for Architectural Description of Software Intensive Systems, IEEE 1471 defines architecture as the "*fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution*" (IEEE Computer Society, 2000).

The Software Engineering Institute (SEI), as a reference institute on software engineering and software architecture field, defines software architecture as "*the blueprint for both the system and the project developing it, defining the work assignments that must be carried out by design and implementation teams*" (SEI, n.d.). They position it as an artifact used in the early analysis, able to assess if the output of the design approach comprises the elicited requirements.

The Rational Unified Process (RUP) defines software architecture as the "*set of significant decisions about the organization of a software system, the selection of the structural elements and their interfaces by which the system is composed together with their behavior as specified in the collaboration among those elements, the composition of these elements into progressively larger subsystems, the architectural style that guides this organization, these elements and their interfaces, their collaborations, and their composition*" (Philippe Kruchten, 2004).

The view regarding the software architecture strongly depends on the desired goal and context. Kruchten uses five concurrent views for representing different concerns of a software architecture in its approach called "*4+1 View Model*" (Philippe Kruchten, 1995). For addresses a specific set of concerns of interest to different stakeholders in the system, Kruchten uses the logical view, the process view, the physical view, the development view and the scenarios.

Software architectures are useful artifacts for development teams, especially for enterprise integration and interoperability, which gave origin to a plethora of frameworks and references (Chen, Doumeingts, & Vernadat, 2008), mainly address heterogeneous environments.

Agile Software Development (ASD)

ASD is not a framework or methodology, but rather a culture and a set of self-replicated ideas (Philippe Kruchten, 2007).

Agile development processes are based in self-organized teams for resolving their problems, dividing the implementation of complex software in small iterations periodically assessed, in order to solve eventual problems as soon as they emerge.

Based in the agile manifesto, Cho states common characteristics on agile methodologies, as well as some common perception on this methodologies' strengths and weaknesses, as follows (Cho, 2009) in Table 1.

Table 1. Characteristics of agile methodologies (Cho, 2009)

Characteristics	Strengths	Weaknesses
Iterative and incremental development Customer collaboration Frequent delivery Light and fast development cycle Tacit knowledge within a team Light documentation	Short development cycle High customer satisfaction Low bug rate Quick adaptation to rapidly changing business requirements	Significant document reduction and heavy dependence on tacit knowledge Not sufficiently tested for mission/safety-critical projects Not adequate for highly stable projects Can be successful only with talented individuals who favor many degrees of freedom Not appropriate for large-scale projects

Large-scale Agile Development

The term 'large-scale agile development' (LSA) has been used to describe agile development in everything from large teams to large multi-team projects to making use of principles of agile development in a whole organization (Dingsøyr & Moe, 2014). For Eckstein, LSA development occurs when, for instance, a team in a XP-based project is composed by more than 12 members (Eckstein, 2013), or when in Scrum projects is performed Scrum of Scrums (Cristal, Wildt, & Prikladnicki, 2008). For Dingsøyr and Moe, LSA development is characterized based on aspects of size such as number of people involved in the development, lines of code in the solution, number of development sites, number of teams, to definitions such as "agile in larger organizations" (Dingsøyr & Moe, 2014). Moreover, LSA is defined as "agile development efforts with more than two teams" (Dingsøyr, Fægri, & Itkonen, 2014), "projects are those with more than 100 people and are longer than 1 year duration" (Crocker, 2004). Additionally, "very large-scale agile development" occurs when ten or more teams are in the project (Dingsøyr et al., 2014).

1.4 Research question and objectives

This PhD's research question is as follows:

"How to adopt logical architectures in agile large-scale projects?"

Prause and Durdik argue that architectural design can be improved in agile methods by (Prause & Durdik, 2012): (1) agile architectural modelling using an incremental, customer-involved process; and (2) an initial vision of the system including initial design is created during the first iteration of the development, where architectural design is more a draft that is changed during later development; (3) more detailed design followed further on several iterations for designing the system; and (4) continuous iterative design where design is embedded into agile development and architectural artifacts are updated regularly.

The adoption of a logical architecture implies its usage as a complementary approach to agile in the development life cycle, in parallel with up-front planning, storyboarding, Sprint, and working software (Madison, 2010).

It is expected that this research output a method for using logical architectures in a typical software agile development, as well as in agile large-scale contexts.

This topic must be addressed in the very beginning of the agile process, and afterwards the research must aim at providing the architecture with the required flexibility during the iteration-cycles of the process. Thus, these two project phases require distinct approaches. One addressed by one objective, and the other one addressed by three objectives:

O1: To develop an approach capable of deriving logical architectures in order to establish the initial requirements that are passed on to agile development teams.

Using logical architectures for establishing initial requirements allows to combine requirements from backlogs (that focus only on functional features) with the quality attributes of the software (Jeon, Han, Lee, & Lee, 2011). This PhD research aims at including some upfront design in the set-up phase of the project (by some we do not mean BDUF, rather using existing research related to the sufficient amount of information for architecture design (Waterman, Noble, & Allan, 2012)) – like in a "waterfall" approach. Additionally, it aims using the architecture as input for an ASD approach (back to requirements again) to build almost the totality of the Product Backlog, with ongoing architectural refinement during the iterative development (Abrahamsson et al., 2010). This set-up phase of the project is called "pregame" (Ken Schwaber, 1997), or "Sprint 0".

The 4SRS method allows deriving logical architectures aligned with the corresponding, and previously elicited and modeled, user requirements. In this PhD thesis, the 4SRS method will be adapted for adopting logical architectures in a typical software agile development. The conventional version of the 4SRS method is typically applied in large-scale projects, but demands high quantity of information (use cases, textual descriptions), which is often time consuming and, in every way, misaligned with the general paradigm adopted by ASD approaches ("*Working software over comprehensive documentation*").

Within this software development phase, some existing knowledge and concepts are considered, namely:

- The architecture should emerge gradually Sprint after Sprint, as a result of successive small refactoring (Abrahamsson et al., 2010);
- Performing lightweight amount of effort in up-front design, by using, for instance, a "predefined architecture" (Waterman et al., 2012), walking skeleton (Farhan, Tauseef, & Fahiem, 2009) and simple artifacts (informal box-and-line diagrams, descriptions of a system metaphor, a succinct document capturing the relevant decisions, etc.) (Erdogmus, 2009).

This way, the architecture is able to handle all the known "big rocks", i.e., requirements that are particularly hard to incorporate late in the project (Cockburn, 2006) and used as a starting point for the generation of User Stories to be incorporated in the Backlog artifact.

O2: To adopt flexibility and agility mechanisms in the refinement of logical architectures throughout the iterations of ASD teams.

The adoption of a logical architecture implies that it is used as a complementary approach to agile in the development life cycle (Madison, 2010). This objective relies in adding the 4SRS method with mechanisms to refine the requirements from pregame phase (addressed in O1) but also to respond to changes during the ASD iterations. This research will aim in using the 4SRS to trace every decision made during the development ("game" phase), from the stated user requirements (in O1) to the delivered software.

Considering that changes in requirements are frequent (and embraced) in agile environments, the resulting artifact from the previous objective must imperatively be able to respond to those changes without losing information and not being subjected to unnecessary refactoring efforts. A research opportunity arises, where some existing knowledge and concepts will be taken in consideration, namely:

- Assessing the impact of changes in features within the architecture (Díaz, Pérez, & Garbajosa, 2014) as well as evaluating the architecture at the end of every cycle (Kanwal, Junaid, & Fahiem, 2010);
- As the requirements are being developed and refined, the architect should identify architecturally significant requirements (ASR), feature- (or functionally-) oriented requirements, and the dependencies between them to ensure the necessary elements from the architecture are present in upcoming iterations (Nord, Ozkaya, & Kruchten, 2014);
- It is, thus, required to provide the logical architecture with agility (that, for Farhan *et al.*, relates to evaluate, discuss and correct quickly the architecture (Farhan et al., 2009)) during the small cycles of the process.

The outputs from this research objective may be used in contexts where: substantial changes to the software architecture need to be explored (Farhan et al., 2009); given a change in features (adding, deleting or updating), it is possible to trace the changes to the stated requirements and assess the changes to the architecture (Díaz et al., 2014); "small rocks" (in opposition to the "big rocks" stated within the previous research objective) are handled as they appear during the project (Cockburn, 2006).

O3: To develop an approach oriented for continuous architecting, aiming to specify microservices logical architectures (MSLA), identifying them and their interfaces.

Use domain-driven design for requirements engineering where, included in the proposed agile modeling logical architecture, uses the 4SRS method is used for proposing MSLA in:

- Projects for breaking monoliths to microservices;
- Greenfield projects of microservices-based solutions.

O4: To use logical architectures to manage a team assignment and orchestration process

Research regarding the use of a logical architecture artefact as a supporting basis for an LSA project that includes:

- a modelling approach for identification of concerns within the architecture;
- a set of issues for validating subsystems size;
- a format to communicate subsystems specifications to teams;
- steps for delivering dependencies, priorities, of subsystems to agile distributed teams.

Additionally, for this research objective to be fulfilled, the method must be able to be used in a typical ASD and in contexts of LSA projects. For scope definition of the work, LSA is characterized by having more than one team, when the number of team members is larger than the typically suggested limit (7 or 8 elements), or when large quantities of user stories (or requirements) or lines of code are required. For both contexts (but especially in LSA projects), some existing knowledge and concepts will be taken in consideration, among them: prioritizing requirements and depicting dependencies between features; coordinating and synchronizing distributed teams (Dingsøyr & Moe, 2014).

1.5 Research method

Within the research objectives, it is clear that the way to fulfill them is by designing a method that is able to derive logical architectures with the desired capabilities as the ones stated previously. In order to fulfill the research objectives, this PhD thesis is structured according the Design Science Research (DSR) methodology. The decision on using DSR relies mainly in the fact that the focus is to develop an artifact, namely an agile modeling process. The artifact is developed under the execution in demonstration cases. Due to the fact that these demonstration cases occur in different organizations and environments, this thesis uses DSR instead of, for instance, Action Research (Baskerville & Wood-Harper, 1998; Coughlan & Coghlan, 2002) or Design Action Research (Sein, Henfridsson, Purao, Rossi, & Lindgren, 2011).

In this section, the DSR method is overviewed and the research process to be conducted in the PhD thesis is described.

DSR addresses important unsolved problems in unique or innovative ways or solved problems in more effective or efficient ways. The key differentiator between routine design and design research is the clear identification of a contribution to the archival knowledge base of foundations and methodologies. The design-science paradigm seeks to extend the boundaries of human and organizational capabilities by creating new and innovative artifacts (Hevner, March, Park, & Ram, 2004), a body of knowledge about the design of artificial objects and phenomena (*i.e.*, artifacts) designed to meet certain desired goals (Simon, 1996). It seeks to create innovations that define the ideas, practices, technical capabilities, and products (Denning, 1997) through which the analysis, design, implementation, management, and use of information systems can be effectively and efficiently accomplished.

This PhD thesis structures its phases using the proposal from Kuechler and Vaishnavi (Kuechler & Vaishnavi, 2008), as depicted in Figure 1.

This proposal begins with the Awareness of a Problem, and then a solution is created, drawn abductively from existing knowledge. The rigor of DSR is derived from the effective use of prior research (existing knowledge base). Solution and respective Artifacts are evaluated through metrics that instantiate the research goals. These steps are repeated until a satisfactory solution to problem is found.

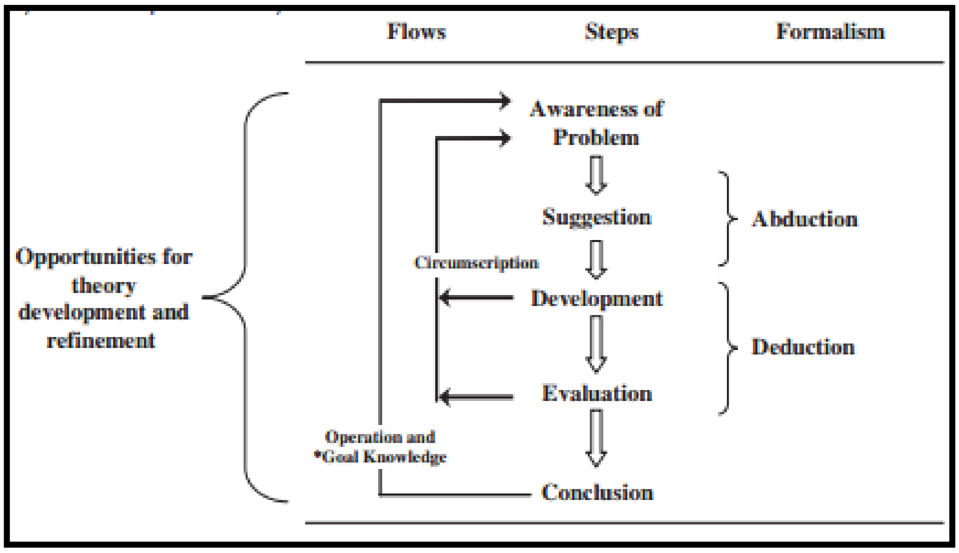


Figure 1. Design Science Research Cycle (Kuechler & Vaishnavi, 2008)

The performed research strategies throughout the research process will be as follows:

1) Awareness of Problem

Some literature review on the topics under the PhD thesis provides the foundations that are required to define the theory development (Webster & Watson, 2002) on agile, architectures, and agile architecting. This phase intents to analyze the existing knowledge and to identify a research opportunity that is not addressed by the analyzed literature.

2) Suggestion

The proposal of hypothesis is built based in the identification of the research opportunity. This hypothesis is formalized in four research objectives that together answer the research question.

3) Development

By performing demonstration cases, the proposed method and process are applied in R&D funded research projects conducted within CCG\ZGDV institute. These projects are conducted with ASD teams, but in different organizations and contexts. It is thus necessary to identify for which research objectives each demonstration case will have contributions.

4) Evaluation

The evaluation used demonstration cases, aiming to experiment and evaluate of the method applicability (Yin, 2014). Kitchenham proposes steps for conducting a demonstration case, namely Context, Setting the Hypothesis, Planning, Validating and Analyzing the Results (Kitchenham, Pickard, & Pfleeger, 1995). For the purpose of the PhD thesis, five demonstration cases were performed within contexts of R&D funded research projects conducted within CCG\ZGDV institute, each one with specific contributes within the thesis.

5) Conclusion

The demonstration case's contribution is added to the research objective (O1, O2, O3, O4 or all). Until all four objectives are validated, the research process is conducted by new DSR cycles. Additionally, work publications relating to the main findings are prepared for submission to journals and conferences.

Every doctoral work intends to develop a new theory, however its research should (or must) be based in a supporting research theory. Design theory (DT) is about having in consideration the analysis and evaluation of design within research (Larsen, Allen, Vance, & Eargle, 2015). Within the case of IS research, DT focus on the design of IT artifacts. These artifacts are broadly defined as *constructs* (vocabulary and symbols), *models* (abstractions and representations), *methods* (algorithms and practices), and *instantiations* (implemented and prototype systems) (Hevner et al., 2004). It describes the world as acted upon (processes) and the world as sensed (artifacts) (Hevner et al., 2004). From the author's point of view, DT is the theory in which the designed artifacts are the basis.

Hevner *et al.* defined a framework for understanding, executing, and evaluating IS research, combining behavioral-science and design-science paradigms. It is composed by three spaces - environment, knowledge base and IS research – and is structured as represented in Figure 2.

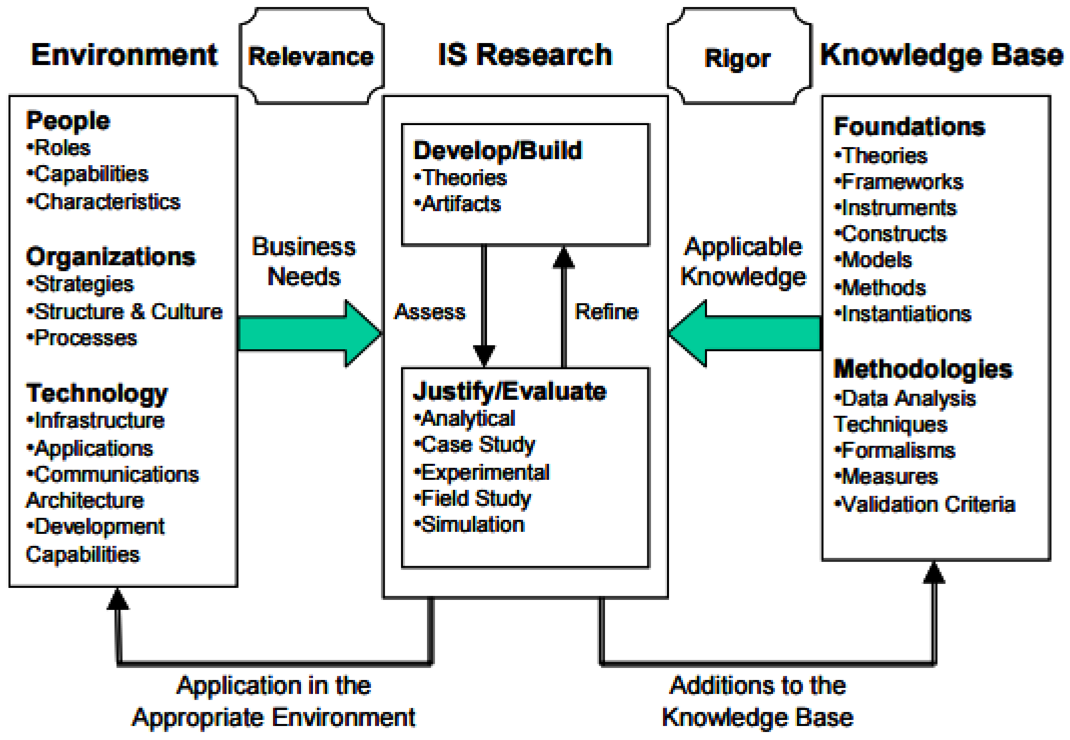


Figure 2. Design Theory for IS research (from (Hevner et al., 2004))

The environment defines the problem space in which reside the phenomena of interest. For IS research, it is composed of people, (business) organizations, and their existing or planned technologies. Together these define the business need or problem as perceived by the researcher (Hevner et al., 2004; Silver, Markus, & Beath, 1995; Simon, 1996). Framing research activities to address business needs assures research relevance.

Given such an articulated business need, IS research is conducted in two complementary phases. Behavioral science addresses research through the *development* and *justification* of theories that explain or predict phenomena related to the identified business need. Design science addresses research through the *building* and *evaluation* of artifacts designed to meet the identified business need.

The knowledge base provides the raw materials from and through which IS research is accomplished. It is composed of foundations and methodologies. Additionally, uses reference disciplines provide foundational theories, frameworks, instruments, constructs, models, methods, and instantiations used from prior IS research and results in the develop/build phase of a research study; and methodologies provide guidelines used in the justify/evaluate phase. Rigor is achieved by appropriately applying existing foundations and methodologies.

Regarding the use of DT, this thesis focus in developing an IT artifact, namely an agile architecting method, i.e., a method for logical architectures that fulfills requirements for use within agile projects.

Agile teams are the surrounding environment of this thesis. Teams, their process and the development artifacts have faced some critics when in large-scale contexts. These issues regard the business need for this thesis' underdevelopment DT.

As for assuring rigor, this thesis uses, for a knowledge base: (1) frameworks that propose emerging architecture design - like Abrahamsson et al., Waterman et al. or Farhan et al. (Abrahamsson et al., 2010; Farhan et al., 2009; Waterman et al., 2012); (2) as well as architectural design tasks within agile iterations – like Jeon et al., Díaz et *al.* or Kanwal et al. (Díaz et al., 2014; Jeon et al., 2011; Kanwal et al., 2010); (3) models related to the logical architecture artifacts; (4) constructs for the 4SRS method execution; and (5) methods - namely the Design Science Research – that enable organizing the study. All these concepts allow developing a rigorous research.

Upon these concepts, the hypothesis about use of the 4SRS in an emerging, iterative and continuous approach is constructed, ultimately resulting in a research question and its research objectives. The assessment of the hypothesis will be based upon the performance of demonstration cases.

After the execution of demonstration cases, that result as well in refinements of the hypothesis, later assessed in further demonstrated cases. As an output, the developed artifacts are applied in the environment where the business need arose, as well as the new theory is scientifically validated and able to be added to the knowledge base.

The demonstration cases

In this thesis, the research projects were used as demonstration cases, separately, within the scope of DSR cycles. Each project had a clearly defined input for the research.

The ISOFIN Cloud (Interoperability in Financial Software) project

ISOFIN Cloud is a Portuguese funded project in co-promotion (QREN 2010/013837, under Fundos FEDER through Programa Operacional Fatores de Competitividade – COMPETE and Fundos Nacionais through FCT – Fundação para a Ciência e Tecnologia, FCOMP-01-0124-FEDER-022674). This project is executed in a consortium comprising eight entities (private companies, public research centers and universities), namely CCG\ZGDV Institute, i2S Insurance Knowledge, University of Minho, Faculty of

Sciences and Technology (FCT NOVA) of Lisbon, Maisis - Information Systems, Knowledgebiz, and I-Zone Knowledge Systems.

This project aimed to deliver a set of coordinating services in a centralized infrastructure, enacting the coordination of independent services relying on separate infrastructures. The ISOFIN platform supports the semantic and application interoperability between enrolled financial institutions (Banks, Insurance Companies and others). The cloud solution is able to be deployed in an *Infrastructure-as-a-Service* (IaaS) layer. That layer will support the execution of a set of services that will allow suppliers to specify the behavior of the services they intend on supplying, in a *Platform-as-a-Service* (PaaS) layer. This will allow customers, or third-parties, to use the platform's services, in a *Software-as-a-Service* (SaaS) layer and billed accordingly.

The project included a set of 52 deliverables. This thesis used the following project deliverables:

- M/D207 – ISOFIN Logical Architecture;
- M/D210 – Financial Domain Applications/Services Specifications

The iFloW (Inbound Logistics Tracking System) project

The iFloW project is an R&D project that is part of a consortium program, called Human-Machine Interface Excellence (HMIExcel), between University of Minho and Bosch Car Multimedia Portugal, sponsored in co-promotion nº 36265/2013 (Project HMIExcel - 2013-2015). iFloW is an R&D project that aims at developing an integrated logistics software system for inbound supply chain traceability. iFloW is a real-time tracking software system of freights in transit from the suppliers to the Bosch plant, located in Braga. The main goal of the project is to develop a tracking platform that allows to control the raw material flow from remote (Asian) and local (European) suppliers to the Bosch's warehouse, alerts users in case of any deviation to the Estimated Time of Arrival (ETA) and anticipates deviations of the delivery time window. The iFloW project, as its name refers, relates to logistics domain, and was mainly focused in integration with third party logistics (3PL) service providers and integrating Radio Frequency Identification (RFID) technology, Global Positioning System (GPS) technologies, and an integrated web-based RFID- Electronic Product Code (EPC) compliant logistics information system.

The project included a set of four deliverables. This thesis used the following project deliverables:

- D4.4.2 - Specification of the model for experimental development;
- D5.3.8 – development of functionalities
- D6.7.9 – verification and validation of functionalities developed

The IMP_4.0 (Integrated Management Platform 4.0) project

The IMP_4.0 platform (POCI-01-0247-FEDER-009147, under Portuguese National Grants Program for R&D projects P2020 – SI IDT) enables a software-house, *F3M – Information Systems, SA*, located in Braga, Portugal, to optimize the development process of delivering solutions to their customers with tools to support all their decision-making processes. The solution is based on public and private clouds, which are interoperable with devices in an IoT and Cyber-Physical Systems (CPS) approach.

The IMP_4.0 project is about an ERP system for the textile production domain, where the focus is to support milling, weaving and clothing processes, by providing a set of reusable and integrated software modules. Additionally, the platform's development includes establishment of generic modules and variability management for enabling its extension to textile, footwear, cutlery, metal-mechanic, glassware and other sectors. The research is conducted within an F3M's software team. The team was composed by one Product Manager that owned the business vision, four software architects, four analysts that modeled requirements and architecture design, and two development teams responsible for implementing the resulting architecture. The architects and analysts also performed the measurements within this research.

The project included a set of 36 deliverables. This thesis used the following project deliverables:

- D.1.4 – Functional requirements specifications – initial version;
- D.1.5 – Functional requirements specifications – final version;
- D1.8 – Traceability mechanisms for production management;
- D1.9 – IMP_4.0 logical architecture – initial version;
- D1.10 – IMP_4.0 logical architecture – final version;
- D1.11 – IMP_4.0 platform services specification.

The UH4SP (Unified Hub for Smart Plants) project

UH4SP is a Portuguese funded project in co-promotion (Project ID 017871, under Portuguese National Grants Program for R&D projects P2020 – SI IDT, and under COMPETE: POCI-01-0145-FEDER-007043). The UH4SP project aims developing a platform for integrating data from distributed industrial unit plants, allowing the use of the production data between plants, suppliers, forwarders and clients. The consortium was composed with five different entities for software development where each had specific expected contributes, from cloud architectures to industrial software services and mobile applications. The solution is based in the Industry 4.0 paradigm, and IoT and cloud computing technologies. The entities are geographically distributed, but each entity had a single located team.

The project included a set of 50 deliverables. This thesis used the following project deliverables:

- D.3.1 – Functional and Technical Requirements Specification;
- D.3.2 – Technical and logical architecture;
- D3.3 – Service Specification For Material Reception And Shipment;
- D3.5 – Interoperability Between Platform And Services Requirements;
- D3.7 – Solution modelling;
- D4.1.1 – UH4SP Management Platform – Initial Version;
- D4.1.2 – UH4SP Management Platform – Final Version;
- D5.4 - Integration Services and Platform.

The Internal Management System of Project Management (IMSPM) project

In this case, the IMSPM is not a funded R&D project, but rather an internal project for i2S. This project is an initiative from i2S for refactoring an existing platform for their internal project management procedures, migrating it from a monolith system to a microservices architecture system.

Because it is an internal project, the existing documentation for this project is private. The only available documentation is in form of an MSc thesis, whose work was associated with this PhD thesis. This MSc thesis can be found in: Amaral, José Diogo Coelho, "The evolution of monolithic architectures to microservice-based architectures" (free translation of "*A evolução das arquiteturas monolíticas para as arquiteturas baseadas em microserviços*"), ISEP - DM – Engenharia Informática[1].

If some projects were used in same DSR cycles as complimentary validation with each other, other were used to validate as alternative approaches. Finally, some cases were used for specific stages of AMPLA, for instance UH4SP was used for entire AMPLA process.

Throughout the thesis, the UH4SP is described as the main demonstration case in the contributions. Whenever it is justifiable, whether complimentary or alternative, the inputs from the remaining demonstration cases are described in the respective sections.

1.6 Document Structure

This document is structured in four parts: **Part I – Introduction**, refers to *Awareness of the Problem* phase of the DSR, as well as the stating the *Business Need* of the DT; **Part II – State of the**

[1] Available at: http://hdl.handle.net/10400.22/11920

Art, refers to *Suggestion* phase of the DSR and the outputs of the *Knowledge Base* of the DT; **Part III – Contributions**, refers to the *Development and Evaluation* phases of the DSR and also the *Develop/Build* and *Justify/Evaluate* from the IS Research of the DT, including the results of the demonstration cases for each of the research contributions; and **Part IV – Conclusions**, refers to *Conclusion* phase of the DSR as well as the *Application in the Appropriate Environment* and the *Addition to the Knowledge Base* of the DT.

Part I is composed by one chapter. In Chapter 1, *Introduction*, the thesis' motivation and background is firstly introduced, including the research question and objectives, the research method and the contributions.

Part II is composed by two chapters. Chapter 2, *Requirements and Logical Architecture Design in Large-scale Agile*, introduces the modeling approaches and techniques of requirements engineering and architecture design in ASD and LSA settings. Chapter 3, *Continuous design as part of Agile Architecting*, describes continuous software engineering and continuous architecting approaches, and includes discussions in architecture lifecycle, architecture management and microservices architectures.

Part III is composed by three chapters. These chapters relate to the research results of the demonstration cases, towards the thesis' research contributions. Chapter 4, *Agile Modeling for Candidate Logical Architectures*, describes the modeling approach within AMPLA for candidate logical architecture, namely upfront and emerging approaches for requirements engineering, which in the later proposes DUARTE: Decomposing User Agile Requirements ArTEfacts (Upfront modeling; and Emerging modeling) and "Just-Enough" Modeling. Chapter 5, *Agile Logical Architecting using AMPLA*, discusses the model-abstraction evolution of a logical architecture throughout the project lifecycle, from a candidate version to a refined one. Chapter 6, *A design theory for a LSA process based in logical architectures*, the use of a design artifact, such as the logical architecture, in an agile-oriented multi-teams process- and project-management. The chapters are structured this way because in chapters 4 and 5 the discussion relates to the modeling discipline, namely the architecture evolution throughout the AMPLA process as well as a decrease of the abstraction, and in Chapter 6 the discussion relates to the use of models in process- and project-management disciplines.

Finally, Part IV is composed by one chapter. Chapter 7, *Conclusions*, synthetizes how the research contributions in a set of addressed topics allowed addressing the research objectives - and, consequently, the research question – but also a synthesis of the research efforts in terms of the methods (DSR and DT) and the research projects, the scientific outputs and the published papers, and, finally, future work.

References

Abrahamsson, P., Babar, M. A., & Kruchten, P. (2010). Agility and architecture: Can they coexist? *IEEE Software*, 27(2), 16–22. https://doi.org/10.1109/MS.2010.36

Ambler, S. (2002). *Agile modeling: effective practices for extreme programming and the unified process*. John Wiley & Sons, Inc.

Ambler, S. (2003). Agile model driven development is good enough. *IEEE Software*, 20(5), 71–73. https://doi.org/10.1109/MS.2003.1231156

Ambler, S., & Lines, M. (2012). *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise*. IBM Press.

Baskerville, R., & Wood-Harper, A. T. (1998). Diversity in information systems action research methods. *European Journal of Information Systems*, 7(2), 90–107. https://doi.org/10.1057/palgrave.ejis.3000298

Cao, L., & Ramesh, B. (2008). Agile requirements engineering practices: An empirical study. *IEEE Software,* 25(1), 60–67. https://doi.org/10.1109/MS.2008.1

Chen, D., Doumeingts, G., & Vernadat, F. (2008). Architectures for enterprise integration and interoperability: Past, present and future. *Computers in Industry*, 59(7), 647–659. https://doi.org/10.1016/j.compind.2007.12.016

Cho, J. (2009). A hybrid software development method for large-scale projects: rational unified process with scrum. *Issues in Information Systems*, 10(2).

Cockburn, A. (2006). *Agile software development: the cooperative game*. Pearson Education.

Cohn, M. (2004). *User stories applied: For agile software development*. Addison-Wesley Professional.

Conboy, K. (2009). Agility from first principles: reconstructing the concept of agility in information systems development. *Information Systems Research*, 20(3), 329–354. https://doi.org/10.1287/isre.1090.0236

Conway, M. E. (1968). How Do Committees Invent? *Datamation*, 28–31.

Coughlan, P., & Coghlan, D. (2002). Action research for operations management. *International Journal of Operations & Production Management*, 22(2), 220–240. https://doi.org/10.1108/01443570210417515

Cristal, M., Wildt, D., & Prikladnicki, R. (2008). Usage of Scrum practices within a global company. In *International Conference on Global Software Engineering (ICGSE)* (pp. 222–226). IEEE. https://doi.org/10.1109/ICGSE.2008.34

Crocker, R. (2004). Large Scale Agile Software Development. In *Extreme Programming and Agile Methods-XP/Agile Universe 2004* (p. 231). Springer. https://doi.org/10.1007/978-3-540-27777-4_53

Denning, P. J. (1997). A new social contract for research. *Communications of the ACM*, 40(2), 132–134. https://doi.org/10.1145/253671.253755

Díaz, J., Pérez, J., & Garbajosa, J. (2014). Agile product-line architecting in practice: A case study in smart grids. *Information and Software Technology*, 56(7), 727–748. https://doi.org/10.1016/j.infsof.2014.01.014

Dingsøyr, T., Fægri, T. E., & Itkonen, J. (2014). What is Large in Large-Scale? A Taxonomy of Scale for Agile Software Development. In *Product-Focused Software Process Improvement* (pp. 273–276). Springer.

Dingsøyr, T., & Moe, N. B. (2014). Towards Principles of Large-Scale Agile Development. In *XP 2014: Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation* (pp. 1–8). Springer, Cham. https://doi.org/10.1007/978-3-319-14358-3_1

Eckstein, J. (2013). *Agile software development with distributed teams: Staying agile in a global world*. Addison-Wesley.

Erder, M., & Pureur, P. (2015). *Continuous architecture: Sustainable architecture in an agile and cloud-centric world*. Morgan Kaufmann.

Erdogmus, H. (2009). Architecture meets agility. *IEEE Software*, 26(5), 2–4. https://doi.org/10.1109/MS.2009.121

Evans, E. (2004). *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley.

Farhan, S., Tauseef, H., & Fahiem, M. A. (2009). Adding agility to architecture tradeoff analysis method for mapping on crystal. In *WRI World Congress on Software Engineering (WCSE'09)* (Vol. 4, pp. 121–125). IEEE. https://doi.org/10.1109/WCSE.2009.405

Grau, B. R., & Lauenroth, K. (2014). *Requirements engineering and agile development - collaborative, just enough, just in time, sustainable*. International Requirements Engineering Board (IREB).

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, 28(1), 75–105.

Highsmith, J. A. (2002). *Agile software development ecosystems* Addison-Wesley. Boston, MA.

IEEE Computer Society. (2000). *IEEE Recommended Practice for Architectural Description of Software Intensive Systems - IEEE Std. 1471-2000*.

IIBA. (2017). *Agile Extension to the BABOK Guide v2*. International Institute of Business Analysis.

Jeon, S., Han, M., Lee, E., & Lee, K. (2011). Quality attribute driven agile development. In *9th International Conference on Software Engineering Research, Management and Applications (SERA)* (pp. 203–210). IEEE. https://doi.org/10.1109/SERA.2011.24

Kanwal, F., Junaid, K., & Fahiem, M. A. (2010). A hybrid software architecture evaluation method for fdd-an agile process model. In *International Conference on Computational Intelligence and Software Engineering (CiSE)* (pp. 1–5). IEEE. https://doi.org/10.1109/CISE.2010.5676863

Kitchenham, B., Pickard, L., & Pfleeger, S. (1995). Case studies for method and tool evaluation. *IEEE Software*, 12(4), 52–62. https://doi.org/10.1109/52.391832

Kruchten, P. (1995). The 4+1 View Model of Architecture. *IEEE Software*, 12(6), 42–50. https://doi.org/10.1109/52.469759

Kruchten, P. (2004). *The rational unified process: an introduction*. Addison-Wesley Professional.

Kruchten, P. (2007). Voyage in the agile memeplex. *Queue*, 5(5), 38. https://doi.org/10.1145/1281881.1281893

Kruchten, P. (2013). Contextualizing agile software development. *Journal of Software: Evolution and Process,* 25(4), 351–361. https://doi.org/10.1002/smr.572

Kruchten, P., Obbink, H., & Stafford, J. (2006). The past, present, and future for software architecture. *IEEE Software*, 23(2), 22–30. https://doi.org/10.1109/MS.2006.59

Kuechler, B., & Vaishnavi, V. (2008). On theory development in design science research: anatomy of a research project. *European Journal of Information Systems*, 17(5), 489–504.

Larman, C., & Vodde, B. (2016). *Large-Scale Scrum: More with LeSS*.

Larsen, K. R., Allen, G., Vance, A., & Eargle, D. (2015). *Theories Used in IS Research Wiki*. . Retrieved from http://is.theorizeit.org

Leffingwell, D. (2016). *SAFe® 4.0 Reference Guide: Scaled Agile Framework® for Lean Software and Systems Engineering*. Scaled Agile, Inc.

Madison, J. (2010). Agile architecture interactions. *IEEE Software*, 27(2), 41–48. https://doi.org/10.1109/MS.2010.35

Newman, S. (2015). *Building microservices - Designing fine-grained systems*. O'Reilly Media, Inc.

Nord, R., Ozkaya, I., & Kruchten, P. (2014). Agile in distress: architecture to the rescue. In T. Dingsøyr & N. B. Moe (Eds.), *International Conference on Agile Software Development (XP'14)* (pp. 43–57). Springer Verlag. https://doi.org/10.1007/978-3-319-14358-3_5

Paetsch, F., Eberlein, A., & Maurer, F. (2003). Requirements engineering and agile software development. In *Proceedings of Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2003)*. IEEE.

Prause, C. R., & Durdik, Z. (2012). Architectural design and documentation: Waste in agile development? In *2012 International Conference on Software and System Process (ICSSP)* (pp. 130–134). IEEE. https://doi.org/10.1109/ICSSP.2012.6225956

Schwaber, K. (1997). Scrum development process. In *Business Object Design and Implementation* (pp. 117–134). Springer. https://doi.org/10.1007/978-1-4471-0947-1_11

Schwaber, K. (2015). *Nexus Guide*. Scrum.org.

SEI. (n.d.). *Architecture Practice*s. Software Engineering Group - Carnegie Mellon University.

Sein, M., Henfridsson, O., Purao, S., Rossi, M., & Lindgren, R. (2011). Action design research. *Management Information Systems Quarterly*, 35(1), 37–56.

Shaw, M., & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall (Vol. 123).

Silver, M., Markus, M., & Beath, C. (1995). The information technology interaction model: A foundation for the MBA core course. *MIS Quarterly*, 19(3), 361–390.

Simon, H. A. (1996). *The sciences of the artificial* (Vol. 136). MIT press.

Sutherland, J. (2018). *The Scrum@Scale Guide - The Definitive Guide to Scrum@Scale: Scaling that Work*s, Version 1.0.

Uludag, O., Kleehaus, M., Xu, X., & Matthes, F. (2017). Investigating the Role of Architects in Scaling Agile Frameworks. In *2017 IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC)* (pp. 123–132). IEEE. https://doi.org/10.1109/EDOC.2017.25

Waterman, M., Noble, J., & Allan, G. (2012). How much architecture? Reducing the up-front effort. In *AGILE India* (pp. 56–59). IEEE. https://doi.org/10.1109/AgileIndia.2012.11

Webster, J., & Watson, R. T. (2002). Analyzing the past to prepare for the future: Writing a literature review. *Management Information Systems Quarterly*, 26(2), 3.

Yin, R. K. (2014). *Case study research: Design and methods (5th Edition)*. Sage publications.

PART II

STATE OF THE ART

Chapter 2 - Requirements and Architecture Design in LSA

This chapter presents existing research regarding architecture design and large-scale agile (LSA). Firstly, this chapter introduces agile software development (ASD) and the changes in the software development that arose from this paradigm. Then, the chapter discusses how the architectural design discipline suffered some changes within this paradigm as well as how they coexist with ASD. Afterwards, it includes a section with existing approaches oriented towards using architecture design methods in specific stages of ASD processes. Additionally, LSA approaches and all its specific impacts in development practices are also presented. This chapter ends with the conclusions of the previously presented works.

Chapter 2 - Requirements and Architecture Design in LSA

"*Agile architecture: a paradox,*

an oxymoron, two totally incompatible approaches?"

Pekka Abrahamsson, Muhammad Ali Babar

***and Philippe Kruchten**,* Software Engineering researchers

2.1 Introduction

Software development lifecycle (SDLC) methodologies commonly fit within the spectrum of plan-driven or change-driven (IIBA, 2015). Plan-driven approaches are used in stable contexts that allow projects to follow early planned activities. Activities are usually sequential. Known plan-driven approaches are the waterfall model (Royce, 1970), the "Vee" model (or V-Model) (Ferreira, Santos, Machado, & Gasevic, 2013; Forsberg & Mooz, 1991) or the Rational Unified Process (RUP) (Kruchten, 2004). In a plan-driven approach, requirements are defined upfront almost in their totality before moving to implementation.

In opposition, change-driven approaches focus on rapid delivery of business value in short iterations (IIBA, 2015). These approaches arose before ASD. In fact, ASD is seen as one of the possible change-driven approaches, and not as one project management approach. These approaches firstly appeared as prototyping in 1984 (BW Boehm, Gray, & Seewaldt, 1984), followed by *Rapid Prototyping* (Fischer & Schneider, 1984), *Evolutionary Delivery model* (Gilb, 1985) and the *spiral model* (B. W. Boehm, 1988). Other frameworks like *Rapid Application Development* (RAD) (Martin, 1991), the *Dynamic System Development Method* (DSDM) (Stapleton, 1997) and *Adaptive Software Development* (Highsmith, 2000) were the first ones to be related to a specific type of change-driven approaches, called agile software development (ASD). All were prior to *Scrum* and *eXtreme Programming* (XP) (Beck & Andres, 2004) frameworks.

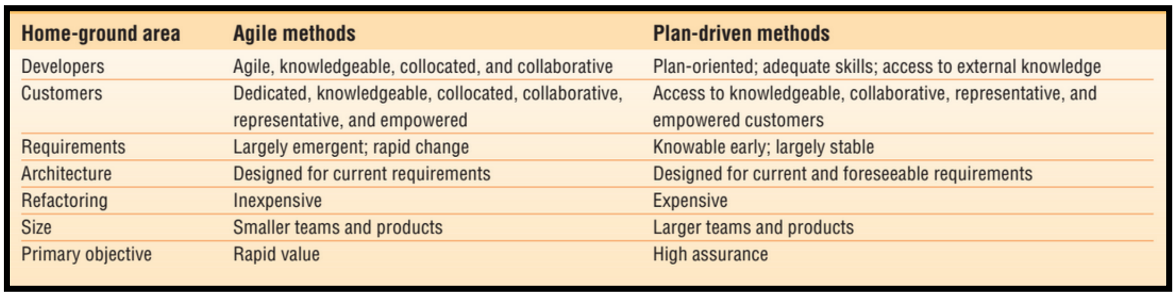
The turning point for the term 'agile' (and ASD) relates to the signing of the 'Agile Manifesto' (Agile Alliance, 2001). The Manifesto does not define any methodologies or practices itself, but rather outlines a philosophy in the form of a set of values and principles that frameworks such as *Scrum* (Ken Schwaber, 1997) and *eXtreme Programming* (XP) adhere. There are also others, like Kanban (Anderson, 2010), Agile Unified Process (AUP) 2005 (SW Ambler, 2005), Crystal

Methodologies (Cockburn, 2004) and Feature-Driven Development (FDD) (Palmer & Felsing, 2001).

Lean software development (Poppendieck & Poppendieck, 2003) is another framework for change-driven approaches. It has been originated by lean production within Toyota (namely in *Toyota Production System* - TPS), as an outgrowth of the larger Lean movement. It embodies seven principles, originally described by Mary and Tom Poppendieck (Poppendieck & Poppendieck, 2003):

1. Eliminate Waste
2. Build Quality In
3. Create Knowledge
4. Defer Commitment
5. Deliver Fast
6. Respect People
7. Optimize the Whole

The agile manifesto values working software over comprehensive documentation, and emphasizes simplicity: maximizing the amount of work not done. This principle can be interpreted in many ways. Most are quite good, but some interpretations can cause problems. For example, XP advocates doing extra work to get rid of architectural features that do not support the system's current version. This approach works fine when future requirements are largely unpredictable (Barry Boehm, 2002). Figure 3 depicts a comparison on best suited for different contexts for agile and plan-driven methods.



Home-ground area	Agile methods	Plan-driven methods
Developers	Agile, knowledgeable, collocated, and collaborative	Plan-oriented; adequate skills; access to external knowledge
Customers	Dedicated, knowledgeable, collocated, collaborative, representative, and empowered	Access to knowledgeable, collaborative, representative, and empowered customers
Requirements	Largely emergent; rapid change	Knowable early; largely stable
Architecture	Designed for current requirements	Designed for current and foreseeable requirements
Refactoring	Inexpensive	Expensive
Size	Smaller teams and products	Larger teams and products
Primary objective	Rapid value	High assurance

Figure 3. Comparison of agile and plan-driven methods (Barry Boehm, 2002)

2.2 Views on Debating Architectures and Agile

There has been some discussion related to the strengths and weaknesses of software architecture with regard of agility. Abrahamsson states key advantages of up-front architecting (also called "*Big Design Up Front*" – BDUF)) (Abrahamsson, Babar, & Kruchten, 2010). Due to the mutual influence between requirements and software architecture (Avgeriou, Grundy, Hall, Lago, & Mistrík, 2011), upfront design implies having consistent and (somehow) stable requirements across the project lifespan (Grundy, 2013).

A major criticism of upfront architecting is the potential efforts in capacity that may never be used (Grundy, 2013), many times referred as "*You Ain't Gonna Need It*" (YAGNI). Non-agile methodologies are accused of not involving the customer properly during all phases of the project. Companies where architectural practices are well developed often tend to see agile practices as "*amateurish, unproven, and limited to very small Web-based sociotechnical systems*" (Kruchten, 2007).

On the other hand, practitioners of agile methods think that architecture-centric methods are "*too much work, equating them with high-ceremony processes emphasizing document production*" (R. L. Nord & Tomayko, 2006), or that "*architectural design has little value*", and that the architecture should emerge gradually Sprint after Sprint, as a result of successive small refactoring (Abrahamsson et al., 2010).

In opposition to these stated accusations, Falessi *et al.* present a study where agile developers perceive software architecture as relevant on the basis of aspects such as communication among team members, inputs to subsequent design decisions, documenting design assumptions, and evaluating design alternatives (Falessi et al., 2010). Practitioners were also questioned about when they should focus on software architecture. The answers were "always" (45%), "never" (5%) and "when the project is complex" (50%), as depicted in Figure 4. Due to the reason of complexity is a broad term, the asked respondents who selected it to choose geographic distribution (19%), number of requirements or lines of code (33%), number of stakeholders (29%), and "other" (19%) as the leading cause of complexity.

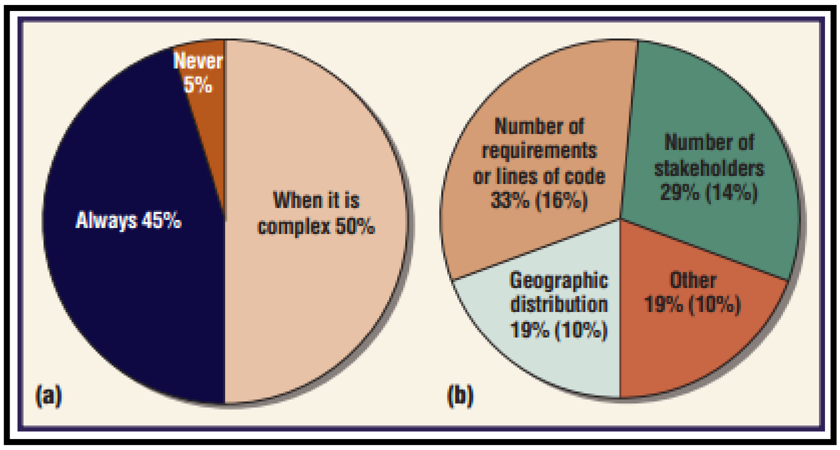


Figure 4. Focus on software architecture by agile developers (Falessi et al., 2010)

Other works also propose useful adoption of architecture design to complement ASD typical development. It is the case of the Zipper Model (Bellomo, Kruchten, Nord, & Ozkaya, 2014; R. Nord, Ozkaya, & Kruchten, 2014). Like in any project, as the requirements are being developed and refined, they are inputs for the architecture design, and allow identifying architecturally significant requirements (ASR). Alongside, more feature- or functional-oriented requirements are identified, as well as relationships between them and between the ASR's. They are further implemented in iterations based in their relationships (Figure 5). This way, the sometimes-disregarded software infrastructure is considered at the same time as the features/functionalities within the ASD iterations.

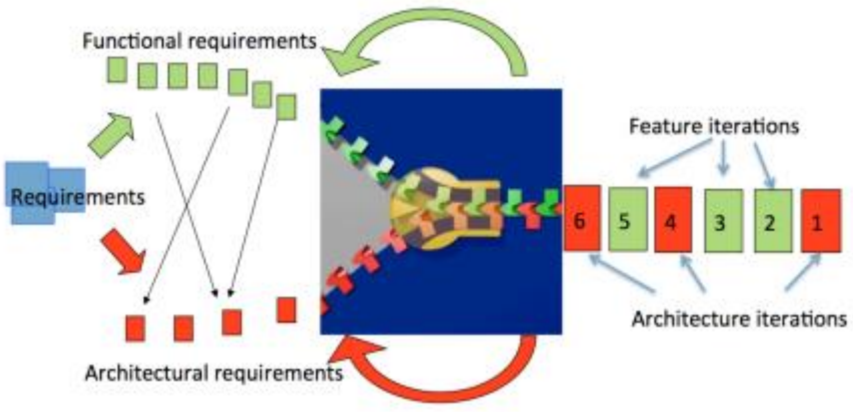


Figure 5. The Zipper model (Bellomo et al., 2014; R. Nord et al., 2014)

Additionally, some aspects must be considered for those interested in designing and deploying agile processes engrained with sound architectural principles and practices (Abrahamsson et al., 2010):

- Understand the context. There is a vast array of software development situations, and although "out of the box" agile practices address many of these, there are outliers that needs understanding: What is the system's size, domain, and age? What is the business model and the degree of novelty and hence of risk? How critical is the system? How many parties will be involved?
- Clearly define the architecture: its scope and the architect's role and responsibility.
- Define an architecture owner, Architects are part of the development group.
- Exploit architecture to better communicate and coordinate among various parties, particularly multiple distributed teams, if any. Define how to represent the architecture, based on what various parties' need to know.
- Use important, critical, and valuable functionality to identify and assess architectural issues. Understand interdependencies between technical architectural issues and visible user functionality to weave them appropriately over time (the zipper metaphor).

The issues of trying to understand the apparent conflict and reconcile the two sides are in multiple dimensions (Abrahamsson et al., 2010):

1) Semantics: What do we mean in this project or organization by "architecture"? The concept of architecture often has fuzzy boundaries. In particular, not all design is architecture. Agreeing on a definition is a useful exercise, and a good starting point.

2) Scope: How much architectural activity will you actually need? Most software projects have a *de facto*, implicit architecture when they start; they will not need much of an architectural effort.

3) Lifecycle: When in the lifecycle should we focus on architecture? Well, early enough, as "architecture encompasses the set of significant decisions about the structure and behavior of the system" (Kruchten, 2004): these are the decisions that will be the hardest to undo, change, refactor. Which does not mean an only focus on architecture, but interleaving architecture "stories" (i.e., stories more focused in quality requirements) and functional "stories" (i.e., stories more focused in functional requirements) in early iterations.

4) Role: Who are the architects? On large, challenging, novel system, you may need a good mix of experience, of "*architectus reloadus*"– maker and keeper of big decisions, focusing on external coordination– and "*architectus oryzus*"– mentor, prototyper, troubleshooter, more code-facing and focused on internal coordination.

5) Documentation: How much of an explicit description of the architecture is needed? While in most cases, an architectural prototype, starting with a walking skeleton, for example, will suffice, and one or a small number of solid metaphors to convey the message, there are circumstances where more explicit software architecture documentation will be needed: to communicate to a large audience, to comply with external regulations, for example.

6) Method: How are we identifying and resolving architectural issues? How to proceed to identify architecturally significant requirements, to perform incremental architectural design, to validate architectural features, etc. There are architectural methods for addressing such issues.

7) Value and cost: All agile approaches strive to deliver business value early and often. The problem seems often that while the cost of architecture is somewhat visible, its value is hard to grasp, as it remains invisible. An approach such as the Incremental Funding Method may allow casting the right compromise between architecture and functionality, without falling into the trap of BDUF.

According to Brown, Nord and Okzaya, ongoing sustainable achievement of Enhancement Agility is only possible when coupled with Architectural Agility (Brown, Nord, & Ozkaya, 2010). To achieve Architectural Agility, the agile community must first expand its focus on end user stories and address the broader topic of capabilities (see Figure 6), including quality attribute requirements and a diverse range of stakeholders.

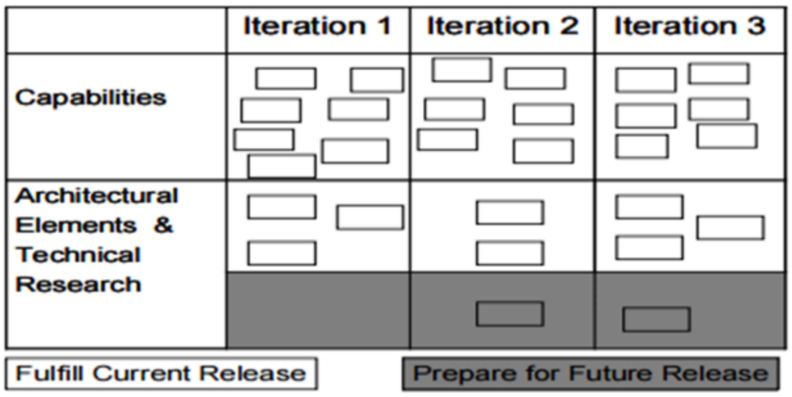

Iteration 1
Iteration 2
Iteration 3
Capabilities
Architectural Elements & Technical Research
Fulfill Current Release
Prepare for Future Release

Figure 6. Informed anticipation in the context of agile release planning (Brown et al., 2010)

The use of dependency analysis practices can be used to facilitate a "just-in-time" approach to building out the architectural infrastructure. Real options and technical debt heuristics can be used to optimize architectural investment decisions by analyzing uncertainty and tradeoffs between incurred cost and anticipated value.

Interest is growing in separating the facts from myths about the necessity, importance, advantages, and disadvantages of having agile and architectural approaches coexist. Like many others in software development research and practice, a healthy focus on architecture isn't antithetic to any agile process, instead of agile practitioners jump directly to refactoring and ignoring architecture (Abrahamsson et al., 2010).

Architectural design can be improved in agile methods by: (1) agile architectural modelling with lowering the overhead of architectural modelling by using an incremental, customer-involved process; and (2) an initial vision of the system including initial design is created during the first iteration of the development, where architectural design is more a draft that is changed during later development (Prause & Durdik, 2012); (3) there are several iterations for designing the system, thus a more detailed design followed further on is created (Prause & Durdik, 2012); and (4) continuous iterative design where design is embedded into agile development and architectural artifacts are updated regularly (Prause & Durdik, 2012).

2.3 Using Architecture Approaches within Agile Software Development

Although there is no explicit support for the concept of architecture in XP methodology, it leads to a software system that should have some specific structures, which we call it, implicit architecture.

One of them is Spike Solutions, used within planning game and when preparing user stories. A Spike solution is a simple program from the potential solutions that could solve a specific problem. In the XP method, the process starts by architectural spikes that could form some kind of initial structure of system. Therefore, it could be mentioned as a kind of analysis and design activity. Spike solution however is created for solving only one problem and the rest of the system is ignored for the purposes of the spike solution for that problem.

Metaphors, on the other hand, are the result of architectural spikes and are claimed to be as a resemblance of architecture in XP (Beck & Andres, 2004). Mainly, the metaphor has two purposes. It is assumed as an abstraction of a system functionality that will keep the team on the same page. A second reason is that the metaphor is supposed to contribute to the team's development of software architecture.

The XP development process is composed of several iterations, each of which results to a set of integrated functionalities at the end. All new functionalities will be tested for their compatibility with other functionalities already developed in the continuous integration step. Although this process may be affective to the system architecture, even indirectly, still architectural smells might be left in the system. These smells represent inefficiencies that could gradually mislead the system`s architecture toward an unmanageable and unsuitable shape unless resolved as soon as possible.

One of the proposals for performing design as concepts and requirements emerge, included in the research of Abrahamsson (Abrahamsson et al., 2010) and Farhan (Farhan, Tauseef, & Fahiem, 2009) is the approach of a walking skeleton. Abrahamsson refers to it as an architectural prototype (Abrahamsson et al., 2010). Farhan refers to it as a tiny implementation of the system that performs minimum functionality. Kazman proposes the design of a candidate architecture (Kazman, 2013). He defines this design as: "*If you are building a large, complex system with relatively stable and well-understood requirements and/or distributed development, doing a large amount of architecture work up-front will likely pay off. On larger projects with unstable requirements, start by quickly designing a candidate architecture even if it leaves out many details.*"

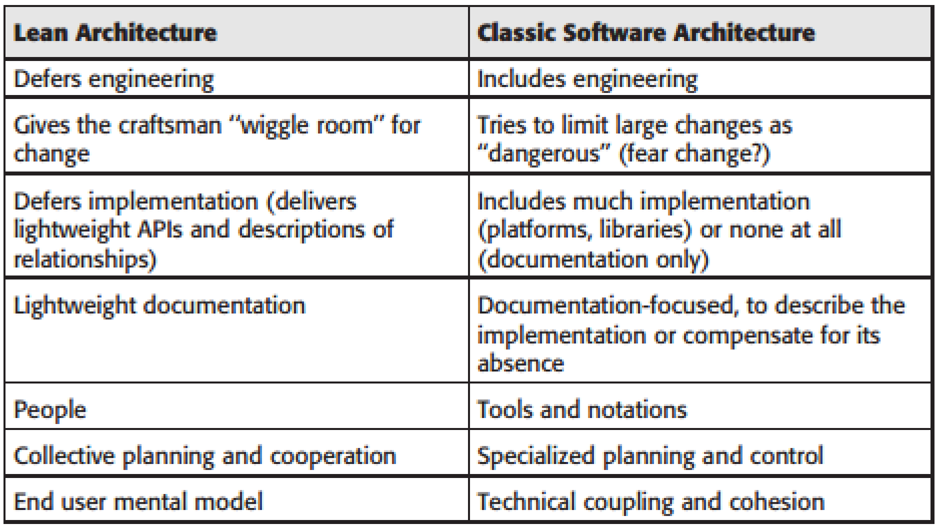
How can a team decide what is "just enough architecture documentation" for their work? Who is the audience for architecture documentation and models? What kinds of architecture documentation might be easier to keep up to date? A detailed architectural plan may be overkill, but an agile architecture model may contain descriptions of the system in several forms (Mancl, Fraser, Opdyke, Hadar, & Hadar, 2009):

- architectural layers
- classes and packages
- interface agreements between internal system components (including internal performance contracts)
- external interfaces
- extension points
- key end-to-end scenarios

When software architecture is expressed in the wrong format or documented excessively, or the representations are used poorly, the result is the illusion of architecting. Clements *et al.* state "*If information isn't needed, don't document it*" (Clements, Ivers, Little, Nord, & Stafford, 2003). All documentation should have an intended use and audience in mind. Effective representation of software architecture is possible with simple artifacts. The artifacts could include specifications written in UML or an architecture description language if appropriate, but a few informal box-and-line diagrams, descriptions of a system metaphor, a succinct document capturing the relevant decisions, and combinations thereof might do the job as well or better (Erdogmus, 2009).

In general, the set-up phase of the project - often called '*iteration 0*' (Abrahamsson et al., 2010) - includes some upfront design, with ongoing architectural refinement during the iterative development. However, how can an architect or developer determine what the correct amount of "just enough up-front design" is, *i.e.*, how can agile teams reduce the up-front effort without sacrificing the benefits of an up-front design (Waterman, Noble, & Allan, 2012)? This work states concepts for reducing the amount of effort in up-front decision making such as "using predefined architecture", "intuitive architecture", "having architectural experience simplifies decision making" and "being familiar with the architecture".

Coplien and Bjørnvig present the concept of "Lean Architecture" (Coplien & Bjørnvig, 2011), where some concerns enabling the architecture design are presented in order to facilitate agility. The Lean perspective focuses on how we develop the overall system form by drawing on experience and domain knowledge. Lean architecture and Agile feature development are much more about focus and discipline, supported by common-sense arguments that require no university degree or formal training. These concerns are depicted in Figure 7.



Lean Architecture	Classic Software Architecture
Defers engineering	Includes engineering
Gives the craftsman "wiggle room" for change	Tries to limit large changes as "dangerous" (fear change?)
Defers implementation (delivers lightweight APIs and descriptions of relationships)	Includes much implementation (platforms, libraries) or none at all (documentation only)
Lightweight documentation	Documentation-focused, to describe the implementation or compensate for its absence
People	Tools and notations
Collective planning and cooperation	Specialized planning and control
End user mental model	Technical coupling and cohesion

Figure 7. Lean Architecture vs Classic Software Architecture (Coplien & Bjørnvig, 2011)

Software Architecture Methods within Initialization

In Attribute-driven sCRUM (ACRUM), three new activities performed in parallel with known Scrum activities. The main steps in the ACRUM progress side by side with the development process of Scrum, keeping its agility intact. This process is depicted in Figure 8. This approach uses a customized QAW and ADD used in Scrum projects (Jeon, Han, Lee, & Lee, 2011).

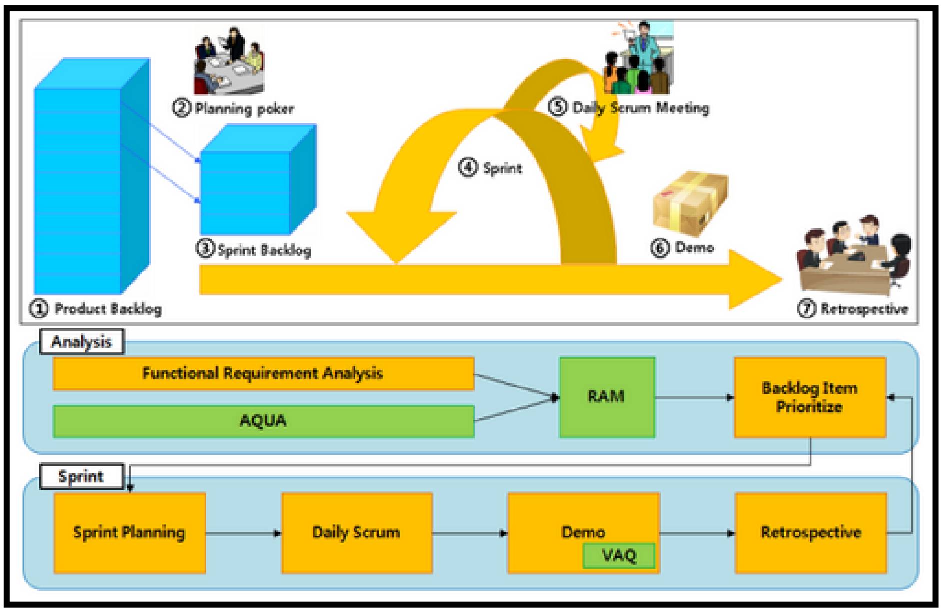


Figure 8. ACRUM development process (Jeon et al., 2011)

In summary, it added three steps to effectively analyze and manage quality attributes. First is the analysis of quality attributes (AQUA). Second is the preparation of the correlation mapping table (RAM) between the requirements and its analysis at the upper part. Finally, the success or failure of the quality attributes is verified through the VAlidation of Quality attributes (VAQ) while demonstration of each final Sprint. Each Sprint was composed of the activities from the Sprint planning to the retrospective meeting. Such activities are repeated for each Sprint.

Software Architecture Methods within Development Iterations

Change-impact architectural knowledge as the main driver for agile architecting in order for documenting architectural knowledge and tracing architecturally significant features with their realization in the architecture (Díaz, Pérez, & Garbajosa, 2014). The models are traversed using a Change Impact Analysis technique to retrieve the architectural design decisions and architectural components and connections that are impacted as a consequence of changing features. The documentation of architectural knowledge supports the rationalization of architectural decisions taken during the solution design. The rationalization of early design decision may help to evolve the architecture while preserving its integrity. The main types of architectural knowledge are the design decisions driving the architecture solution, their dependencies and rationale. The knowledge of adding feature increments or changing features in each agile iteration can be captured in modeling primitives (rationale, constraints, assumptions, etc.) that can be closed, open, optional or alternative design decisions. These four types of design decisions offer a complete support for documenting the knowledge derived from the agile architecting process.

Changes in features affect the system architecture and can lead to ripple effects that are not obvious to detect. The Change Impact Analysis technique (Díaz et al., 2014) consists of two main steps described below:

1. Given a change in features (adding, deleting or updating), the traceability-based algorithm determines (1) the first-order design decisions that are involved with the feature to be changed, (2) the design decisions that depend on the first-order design decisions, and (3) the first-order architectural elements that are involved in each design decisions.
2. Given a change in the working architecture that realizes the change in features, the rule-based inference engine fires propagation rules to obtain the change propagation in the working architecture.

This approach for agile architecting is then deployed in Scrum processes. Figure 9 shows a tailored Scrum development process in which agile architecting is considered as a key activity to prepare the iteration (*i.e.*, Sprint).

The first step consists of capturing the requirements of the Product Owner from the product vision (features). Features may be decomposed into a list of user stories known as product backlog. Then, user stories are prioritized, based on business value, and assigned to Sprints. Scrum implements an iterative lifecycle based on these Sprints. Sprints start with Sprint planning meeting in which the Product Owner and Team plan together what has to be done. In this tailored Scrum, the agile architecting tasks are developed in conjunction with the Sprint planning meetings. Agile architects interact with the rest of the team in planning the features to be done by tracking architectural concerns —constraints, risks, viability, etc. — and balancing them with business priorities. At the end of each Sprint, a working product and a working architecture are delivered. In the Sprint review meeting, the Product Owner assesses the working product to validate that user stories were met, or to introduce changes into the user stories.

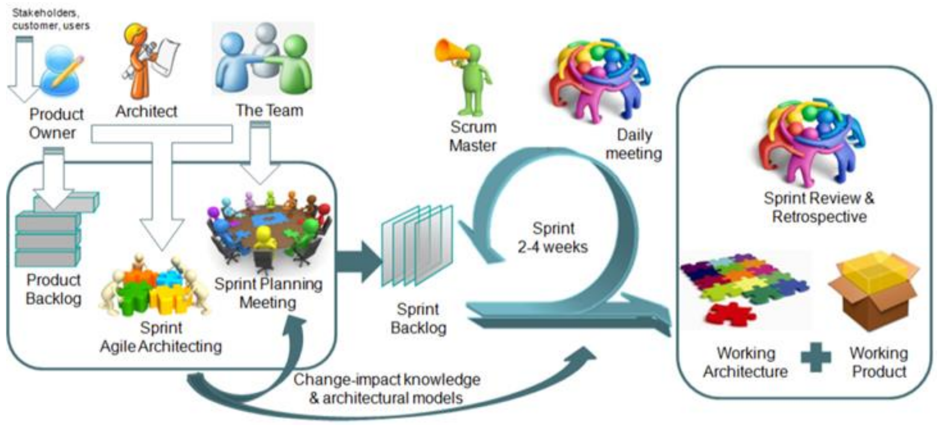


Figure 9. A customization of Scrum for agile product-line architecting (Díaz et al., 2014)

The lightweight ATAM (Farhan et al., 2009), *i.e.*, without some of its activities (see Figure 10), validates architectures in a Crystal project and applying agility to ATAM. Crystal is a non-jealous model and allows integration with other models. Crystal's main theme is that there may be slightly different policies and conventions for each project. It is based upon incremental development not exceeding more than four months.

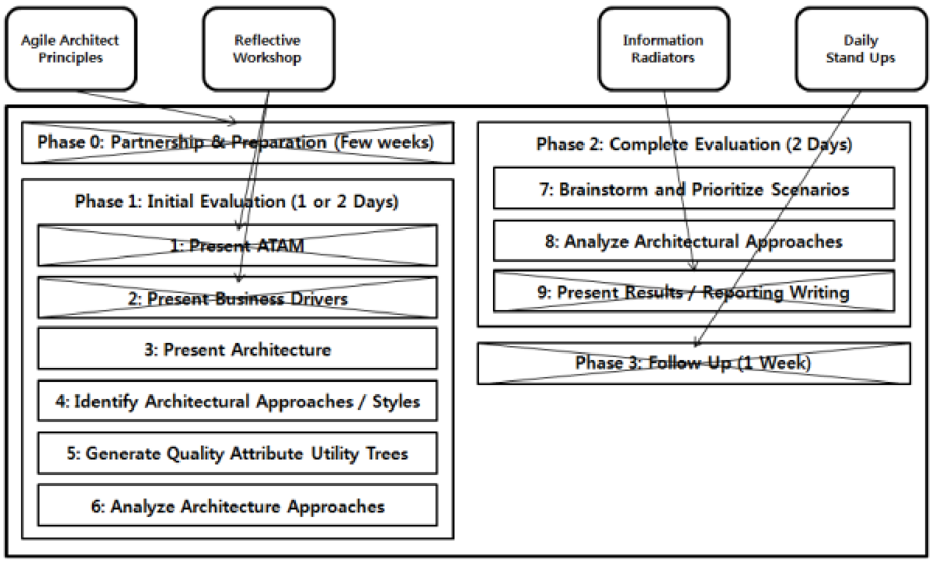

Agile Architect Principles
Reflective Workshop
Information Radiators
Daily Stand Ups
Phase 0: Partnership & Preparation (Few weeks)
Phase 1: Initial Evaluation (1 or 2 Days)
1: Present ATAM
2: Present Business Drivers
3: Present Architecture
4: Identify Architectural Approaches / Styles
5: Generate Quality Attribute Utility Trees
6: Analyze Architecture Approaches
Phase 2: Complete Evaluation (2 Days)
7: Brainstorm and Prioritize Scenarios
8: Analyze Architectural Approaches
9: Present Results / Reporting Writing
Phase 3: Follow Up (1 Week)

Figure 10. Modified ATAM for Crystal Agile Model (Farhan et al., 2009)

The adapted method is composed by the following activities:

- Agile architect principles - is an overall umbrella activity that dictates the modifications in ATAM to introduce agility in it so that it would be compatible with crystal.
- Reflective workshops - is a practice offered by crystal in which teams discuss closely the track, progress, modifications, strengths and weaknesses of the project. These workshops can also eliminate the need for step 1 and 2, and can aid in light weighting the steps 3 and 4.
- Osmotic communication - Crystal, background hearing of information can be provided.
- Information radiators - is a display posted in a place, passage or hallway where people can see easily as they work or walk by. This enables more communication with fewer interruptions. So we can reduce the amount of heavy documentations thus phase 1 and step 9 can be light weighted.
- Ambassador user - Crystal presents the concept of closely involved user, in the development process, called ambassador user. Therefore, teams can have quick feedbacks and can modify requirements as per user satisfaction. This way overall ATAM process can be light weighted.
- Early victory - is the first piece of software in running condition. For that, small problems are solved initially and the underlying principle is that the team should go for easy tasks first. This practice adds a value to the overall ATAM process.

- Walking skeleton - is a tiny implementation of the system that performs minimum functionality. Such kind of skeleton can be evaluated, discussed and corrected quickly, so this adds agility to ATAM process.
- Incremental re-architecture - Once we have walking skeleton of the system, there may be incremental re-architecting quickly. This also lightweight the overall ATAM process and makes it suitable for crystal agile model.
- Daily stand ups - Daily Stand-ups of a few minutes can fix the problems on daily basis. Although this practice does not target any specific step of ATAM but it adds value to overall process. Practices from 4.5 through 4.9 can eliminate phase 3 of ATAM.

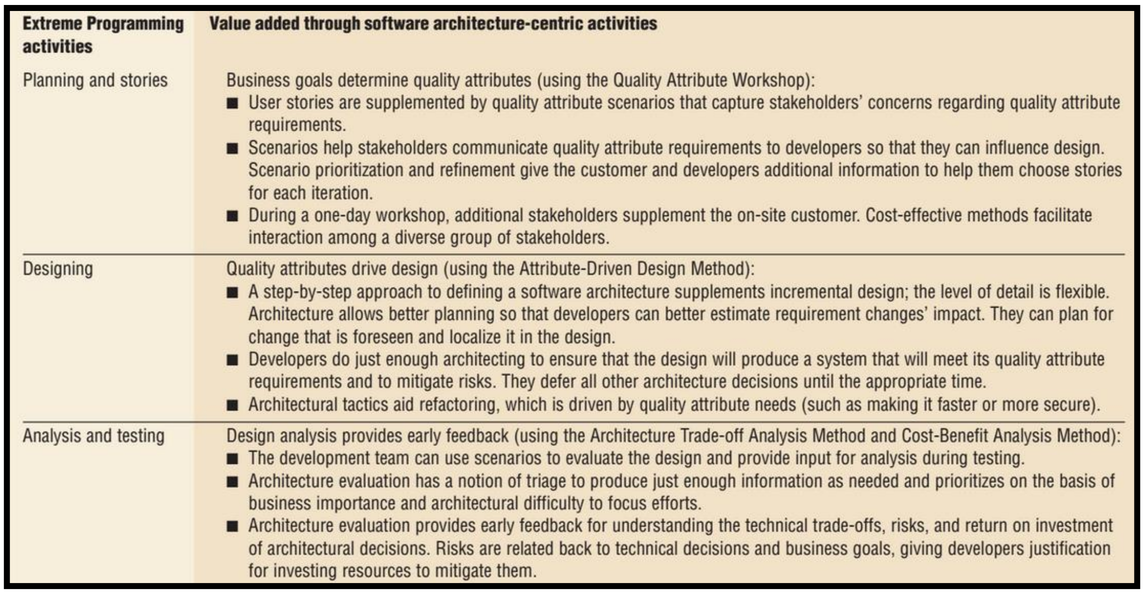
Nord *et al.* explore the relationship and synergies between architecture-centric design and analysis methods and the Extreme Programming framework (R. L. Nord & Tomayko, 2006). Software Architecture Technology Initiative at Carnegie Mellon University's Software Engineering Institute (SEI) has developed and promulgated a series of architecture-centric methods for architecture design and analysis.

The Quality Attribute Workshop (QAW) can help the development team understand the problem by eliciting quality attribute requirements basing on business goals ensures that the developers address the right problems.

The Attribute-Driven Design (ADD) method defines a software architecture by basing the design process on the prioritized quality attribute scenarios that the software must fulfill.

The Architecture Trade-off Analysis Method (ATAM) and Cost-Benefit Analysis Method (CBAM) provide detailed guidance on analyzing the design and getting early feedback on risks. The development team can use incremental design practices to develop a detailed design and implementation from the architecture. Architectural conformance and reconstruction techniques ensure consistency between the architecture and implementation.

These SEI methods can enhance XP practices. The method's applications within the XP practices are presented in Figure 11. Using these methods results in an architecture-centric approach: architecture connects business goals to the implementation, quality attributes inform the design, and architecture-centric activities drive the software system life cycle. These methods make developing software easier and more consistent. Although designing the architecture is integral to the approach, the level of detail can be flexible.



Extreme Programming activities	Value added through software architecture-centric activities
Planning and stories	Business goals determine quality attributes (using the Quality Attribute Workshop): ■ User stories are supplemented by quality attribute scenarios that capture stakeholders' concerns regarding quality attribute requirements. ■ Scenarios help stakeholders communicate quality attribute requirements to developers so that they can influence design. Scenario prioritization and refinement give the customer and developers additional information to help them choose stories for each iteration. ■ During a one-day workshop, additional stakeholders supplement the on-site customer. Cost-effective methods facilitate interaction among a diverse group of stakeholders.
Designing	Quality attributes drive design (using the Attribute-Driven Design Method): ■ A step-by-step approach to defining a software architecture supplements incremental design; the level of detail is flexible. Architecture allows better planning so that developers can better estimate requirement changes' impact. They can plan for change that is foreseen and localize it in the design. ■ Developers do just enough architecting to ensure that the design will produce a system that will meet its quality attribute requirements and to mitigate risks. They defer all other architecture decisions until the appropriate time. ■ Architectural tactics aid refactoring, which is driven by quality attribute needs (such as making it faster or more secure).
Analysis and testing	Design analysis provides early feedback (using the Architecture Trade-off Analysis Method and Cost-Benefit Analysis Method): ■ The development team can use scenarios to evaluate the design and provide input for analysis during testing. ■ Architecture evaluation has a notion of triage to produce just enough information as needed and prioritizes on the basis of business importance and architectural difficulty to focus efforts. ■ Architecture evaluation provides early feedback for understanding the technical trade-offs, risks, and return on investment of architectural decisions. Risks are related back to technical decisions and business goals, giving developers justification for investing resources to mitigate them.

Figure 11. Usage of architecture-centric to improve XP activities (R. L. Nord & Tomayko, 2006)

Within the activities of "Planning and Stories" and "Designing", this approach uses a customized QAW and ADD like in aCRUM (Jeon et al., 2011). Within the activities of "Analysis and Testing" (R. L. Nord & Tomayko, 2006) presented in Figure 11, the approach uses a lightweight ATAM (Farhan et al., 2009).

However, Sharifloo *et al.* argue that this kind of integration presented by Nord *et al.* (R. L. Nord & Tomayko, 2006) is not applicable into a real XP team because of the fact that they are not derived from XP values and practices and are not in the way of agile principles (Sharifloo, Saffarian, & Shams, 2008). In their paper, the primary goal in is to satisfy quality attributes when developing a system using XP method. They introduced two architectural practices Continuous Architectural Refactoring (CAR) that are applied in XP concurrently with other Real Architecture Qualification (RAQ) practices and a new role called Architect is created that performs new responsibilities raised from added practices. Truly, new practices are embedded into XP in order to be conformed to XP values and culture. The purpose is to introduce practices that are going to satisfy architectural needs of a system. In order to provide XP process model with architectural practices, there is a need to think about quality attributes and their characteristics.

Kanwal *et al.* propose a hybrid software architecture evaluation method for FDD agile process model (Kanwal, Junaid, & Fahiem, 2010). The proposed method is hybrid of QAW, ATAM and Active Review for Intermediate Designs (ARID). Due to an emphasis of these models on rapid development, there is an ever-increasing need of architecture evaluation, and a single architecture evaluation method capable of preserving the agility does not exist now. FDD is most suitable for the projects with large team size and low iteration time. Moreover, FDD is very effective in business modeling of the projects.

FDD consists of five major phases with each phase having a set of related activities:

1. Develop an Overall Model
2. Build a Features List
3. Plan by Feature
4. Design by Feature
5. Build by Feature

FDD agile methods are characterized by customer satisfaction, fast response to changes, and release in less time. This approach is hybrid of QAW, ATAM and ARID (see Figure 12). In FDD, architecture is developed in phases 1 and 2.

For phase 1 of FDD, functional as well as non-functional requirements gathering activities should be executed in parallel to ensure the development of proper architecture without affecting the agility. For that, QAW is a very good choice as the major concentration of this architecture method is on determining the quality attributes which establish the non-functional requirements of the project.

For phase 2 of FDD, there are two sub activities that need architecture evaluation. While building the features list, utility trees, sensitivity points and tradeoffs should also be determined to develop a proper architecture. Utility trees, sensitivity points and tradeoffs are the inherent features of ATAM. For the assessments (verifications) ARID is to be executed as it is primarily developed for review activities.

Raatikainen *et al.* describe how software product family engineering and backlog management can be integrated in the light of two approaches called “Agilefant” and “Kumbang” (Raatikainen, Rautiainen, Myllärniemi, & Männistö, 2008). The main element of Kumbang is that it enables describing product family from feature point of view as a *feature model* (see Figure 13). A *feature* is loosely defined as an end-user visible characteristic of a system. As a means of expressing variability and creating dependencies among features, Kumbang features can be composed of other features. A feature can define any number of subfeature definitions, which state what kinds of features can exist under that feature. If a feature does not define any subfeature definitions, it is termed as a *leaf feature*, otherwise as a *composed feature*. Further, a feature can define any number of constraints that create dependencies to other features.

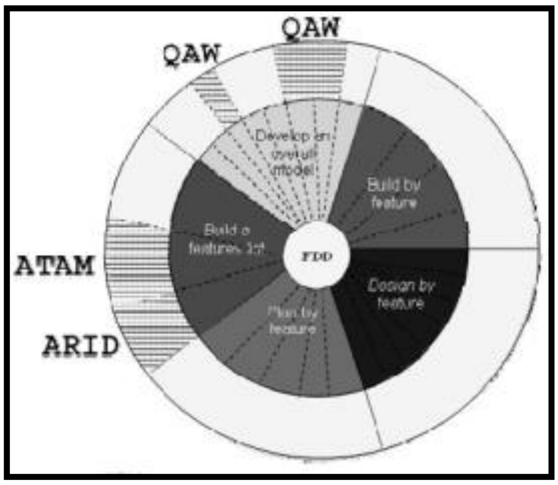


Figure 12. Mapping of QAW, ATAM and ARID on FDD (Kanwal et al., 2010)

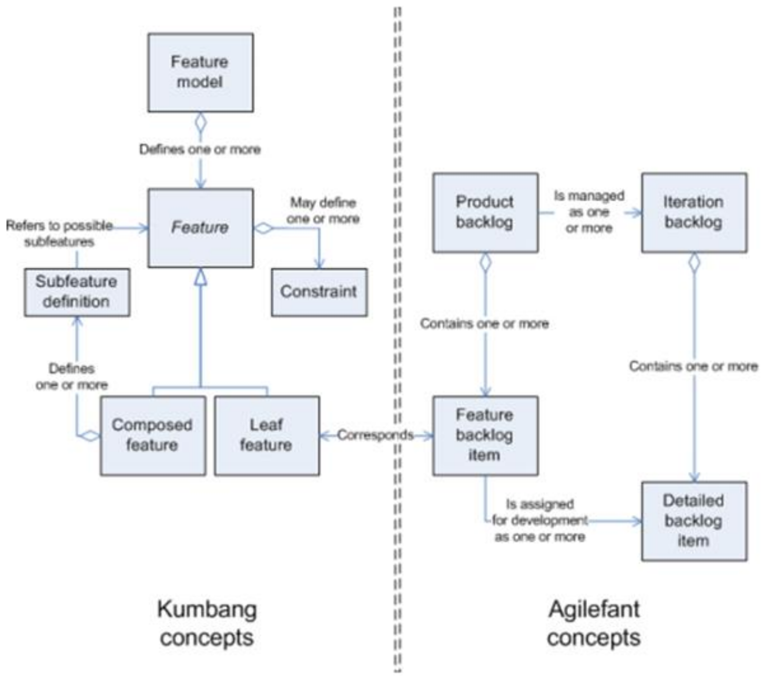


Figure 13. Integrated conceptualisation of Kumbang (Raatikainen et al., 2008)

Consequently, the concept of feature backlog item in Agilefant corresponds with the concept of leaf feature in Kumbang feature model. Further, all leaf features of Kumbang model can have a corresponding feature backlog item in Agilefant, and vice-versa. This mapping, hence, provides integration between a software product family model and items in a backlog.

Madison advocates the coexistence of agile and architecture as complementary approaches and principles (Madison, 2010). He emphasizes the software architect's vital role as a linchpin for combining the two. Madison's approach (see Figure 14), called agile architecture, advocates using agile to get to a good architecture by appropriately applying suitable combinations of architectural functions (such as communication, quality attributes, and design patterns) and architectural skills at four points (up-front planning, storyboarding, Sprint, and working software) in the development life cycle.

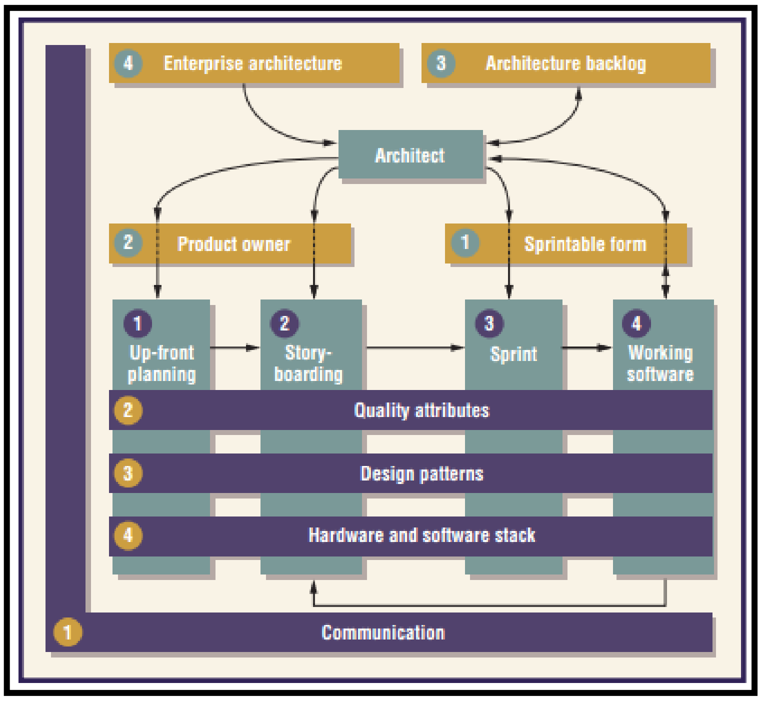


Figure 14. A hybrid framework for agile architecture work (Madison, 2010)

2.4 Large-scale Agile Development (LSA)

Characteristics of LSA

The dimensions used to define a project as large-scale relate to costs, code size and number of requirements (Dingsøyr, Fægri, & Itkonen, 2014). The same work focus on the size of teams when characterizing scaling agile projects, mainly due the coordination and communication needs and practices between teams (Dingsøyr et al., 2014).

There has been an increasing interest on research in this topic, which may include (Reifer, Maurer, & Erdogmus, 2003): (1) Scale agile methods to very large projects with barely sufficient up-front planning and architectural work; (2) Deploy a federation of coordinated teams (each internally operating as an agile team) in scaling up agile ideas; (3) Use agile methods in teams larger than a typical XP team; (4) Characterize the agile continuum through different project caricatures, ranging from typical collocated XP projects to large, multiteam, multiyear ones.

Additionally, LSA initiatives can be about (Uludag, Kleehaus, Caprano, & Matthes, 2018): Culture & Mindset; Communication & Coordination; Enterprise Architecture; Geographical Distribution; Knowledge Management; Methodology; Project Management; Quality Assurance; Requirements Engineering; Software Architecture; and Tooling. Main challenges relate to stakeholders and challenges, Uludag acknowledges that challenges also relate to methodology patterns, architecture principles, viewpoint patterns and anti-patterns. The challenge with more identified papers relates to "Coordinating multiple agile teams that work on the same product", from the "Communication & Coordination" category. From the "Software Architecture" category (in which this thesis focuses on), the "Considering integration issues and dependencies with other subsystems and teams" challenge is the one present in more papers (and actually the second one in all categories), followed by "Managing technical debts". From the "Requirements Engineering" category (in which this thesis also focuses on), the "Creating precise requirement specifications for the Development Team" challenge is the one present in more papers, followed by "Eliciting and refining requirements of end users".

Additionally, at recent events within the International Conference on Agile Software Development ("XP" conferences), there is a dedicated workshop for discussing research trends and challenges in LSA. By gathering the results from workshops during XP2013 (Dingsøyr & Moe, 2013), XP2014 (Dingsøyr & Moe, 2014), XP2016 (Moe, Olsson, & Dingsøyr, 2016), XP2017 (Moe & Dingsøyr, 2017) and XP2018 (Dingsøyr, Moe, & Olsson, 2018), **Table 2** depicts the identified topics throughout the workshops, in an attempt of characterizing recognized challenges and topics within LSA.

There are primarily five frameworks that address scaling agile practices: *Disciplined Agile Delivery* (DAD) (Scott Ambler & Lines, 2012), *Large-Scale Scrum* (LeSS) (Larman & Vodde, 2016), *Scaled Agile Framework* (SAFe) (Leffingwell, 2016), *Scrum@Scale* (Sutherland, 2018) and *Nexus* (K Schwaber, 2015). Each of these frameworks draws from variety of agile and lean practices. Sometimes the "Spotify model" (Kniberg & Ivarsson, 2012) is included within these scaling frameworks, however it is not much as a framework with practices and events for companies to adopt, but rather a cross-matrix structure adopted by Spotify company. These frameworks are now introduced.

Table 2. Research challenges' priority at XP's scientific workhops on LSA

LSA workshop	High	Medium	Low
XP2018	• Agile in public/ IT government • Agile transformation • Business agility • Scaling agile	• Patterns in large scale agile development • The role of architects and architecture in agile • Integrating non-software and software parts of the organization into agile (enterprise agile) • Knowledge sharing / networks	• Inter-team coordination • How DevOps affects agile
XP2017	• Inter-team coordination • Agile transformation	• Agile transformation • Business agility	• Knowledge sharing and knowledge networks
XP2016	• Distributed Large-Scale • Inter-team Coordination	• Knowledge Sharing • Large-scale Agile Transformations	• Multidisciplinary Work • New Ways-of-Organizing
XP2014	• Organisation of large development efforts • Variability factors in scaling • Inter-team coordination • Key performance indicators in large development efforts • Knowledge sharing and Improvement • Release planning and architecture	• Customer collaboration • Scaling agile practices • Agile contracts • Agile transformation • UX design	
XP2013	• Inter-team coordination • Large project organization / portfolio management • Release planning and Architecture	• Scaling agile practices • Customer collaboration • Large-scale agile transformation	• Knowledge sharing and Improvement • Agile contracts

DAD is a hybrid approach that extends Scrum with proven strategies from Agile Modeling (AM), XP, Unified Process (UP), Kanban, Lean Software Development, Outside In Development (OID) and several other methods. A full lifecycle goes from the initial idea for the product, through delivery, to operations and support and often has many iterations of the delivery lifecycle (Figure 15). Because it is not prescriptive and strives to reflect reality as best it can, DAD actually supports several versions of a delivery lifecycle: (1) An agile/basic version that extends the Scrum Construction lifecycle with proven ideas from RUP; (2) An advanced/lean lifecycle; (3) A lean continuous delivery lifecycle; and (4) An exploratory "Lean Startup" lifecycle.

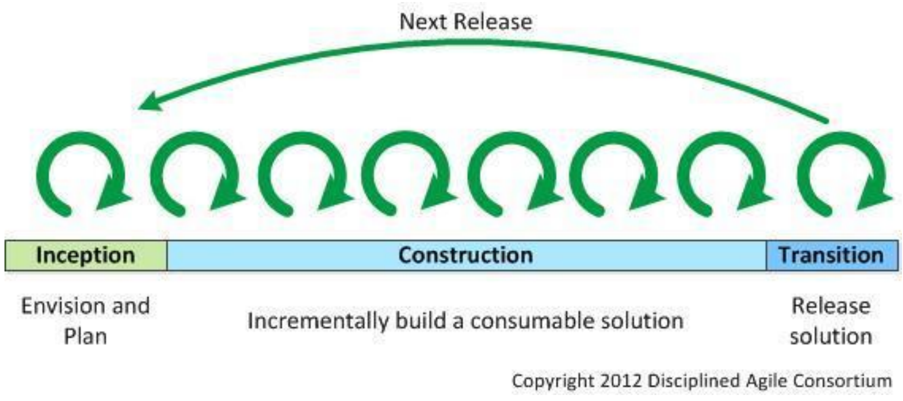

Next Release
Inception
Construction
Transition
Envision and Plan
Incrementally build a consumable solution
Release solution
Copyright 2012 Disciplined Agile Consortium

Figure 15. Disciplined Agile Delivery (DAD)

LeSS is one-team oriented for scaled projects within Scrum practices (Larman & Vodde, 2016). LeSS includes a single Product Backlog (because it's for a product, not a team); one Definition of Done for all teams; one Potentially Shippable Product Increment at the end of each Sprint; one Product Owner; many complete cross-functional teams (with no single-specialist teams); and all Teams in a common Sprint to deliver a common shippable product, every Sprint. The roles, events and artifacts of LeSS are represented in Figure 16.

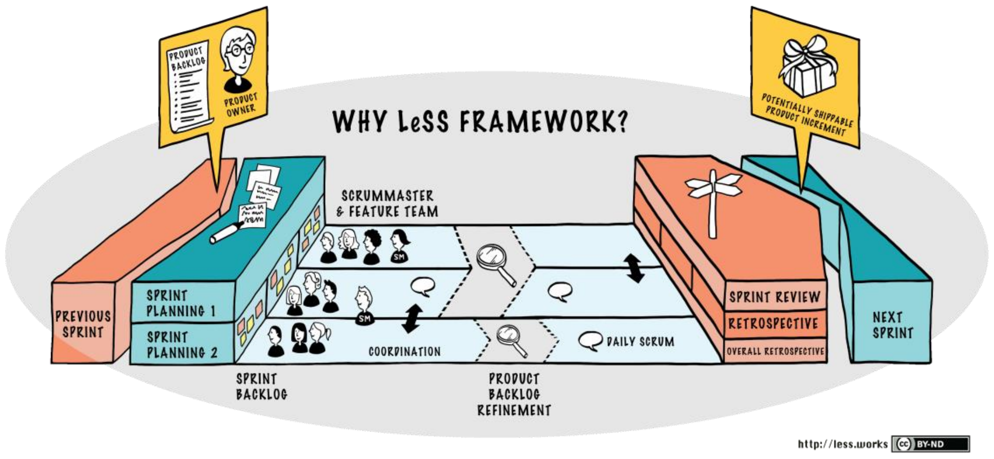


Figure 16. LeSS framework

The Scaled Agile Framework (SAFe) was created by Dean Leffingwell. The framework articulates three levels of organization (Figure 17): Team, Program and Portfolio. Each level incorporates agile and lean practices, has its own activities and all levels are tied together. At the team level, SAFe specifies a blend of Scrum and XP practices. The code practices include Agile Architecture, Continuous Integration, Test-First, Code Refactoring, Pair Work, and Collective Code Ownership. SAFe does not expect teams to produce Potentially Shippable Increment (PSI) every Sprint, but rather over a quarterly cadence. At the program level, provides features, which the teams deconstruct and size to fit into iterations.

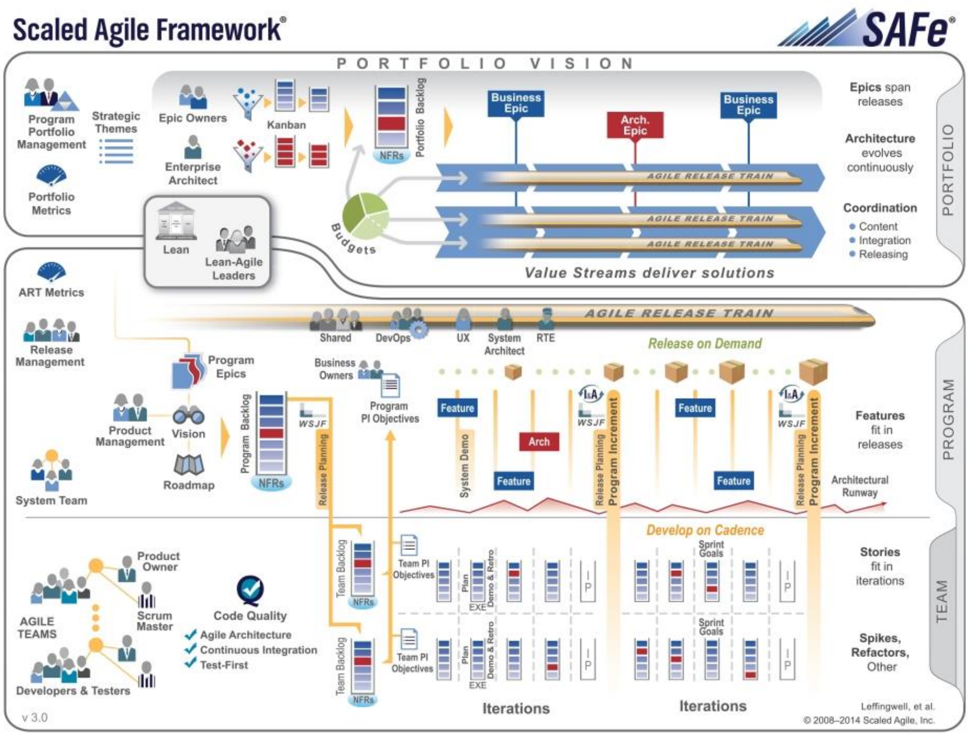


Figure 17. SAFe levels (Portfolio, Program and Team)

The three scaling frameworks provide approaches that attempt to address some of the issues that an organization faces and offer solutions to address these gaps. However, each framework provides some benefits, but they have shortcomings as well. DAD creates four distinct lifecycles, each of which an organization can adapt to fit its context. However, it also specifies an overly complicated work items pool, which an organization can address in much simpler ways. LeSS starts where Scrum leaves off when it comes to scaling agile practices in large organization. However, in the process, it makes recommendations that are problematic, like having a single Product Owner for up to ten teams SAFe organizes its practices into three levels (team, program and portfolio), which is quite useful for larger organization.

At a team level, it embraces certain XP practices, which standard Scrum does not. However, the framework has myriad of issues, including being overtly process heavy (Vaidya, 2014).

Scrum@Scale (Sutherland, 2018) is a framework for scaling Scrum, developed by Jeff Sutherland – "one of the fathers" of Scrum – and Scrum Inc. It is defined as "*A framework within which networks of Scrum teams operating consistently with the Scrum Guide can address complex adaptive problems, while creatively delivering products of the highest possible value*". In short, it is a framework that uses Scrum for scaling Scrum, using approaches of "Scrum of Scrums" (oriented for a team of Scrum Masters) and "MetaScrums" (oriented for a team of Product Owners) for coordinating Scrum teams.

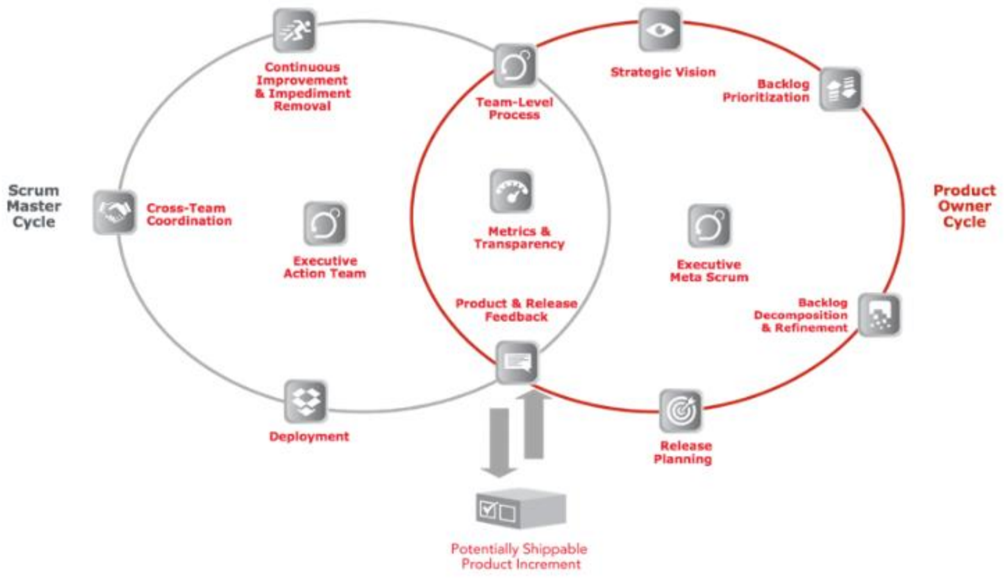


Figure 18. Scrum@Scale

Developed by Ken Schwaber – the "other father" of Scrum - and Scrum.org, the Nexus Framework (K Schwaber, 2015) is a framework for large-scale product or software development largely based on Scrum. By consisting in roles, events, and artifacts, Nexus is defined itself as an exoskeleton resting on top of three to nine Scrum teams. These Scrum teams are dedicated to the development of one integrated "done" product increment, and Nexus framework supports them to deal with dependencies and interoperation.

In contrast to Scrum, Nexus is a quite new framework about which only a small amount of literature was published (Uludag, Kleehaus, Xu, & Matthes, 2017).

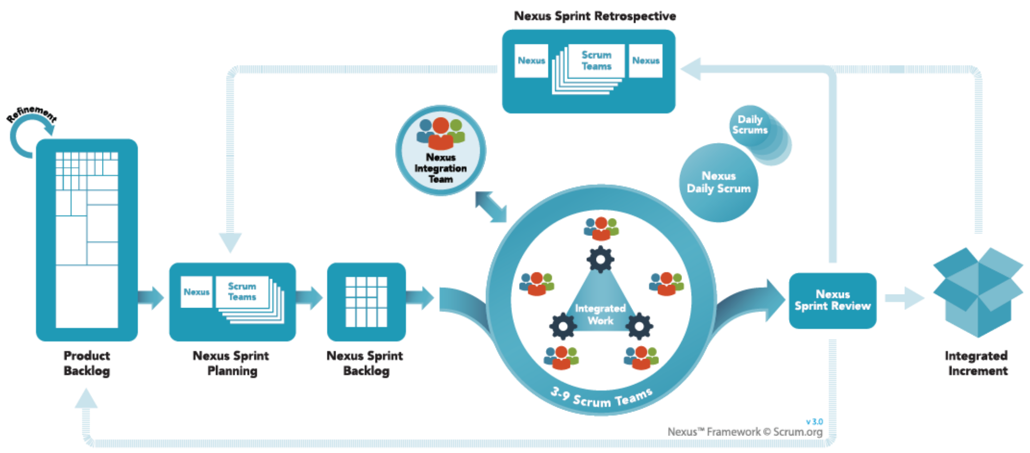


Figure 19. Nexus framework

The “Spotify model” is a cross-matrix structure that promotes knowledge sharing among team members by defining team’s “Squads”, “Tribes”, “Chapters” and “Guilds”. In short, Squads are development teams oriented to feature development. Tribe manages a group of Squads. In order to do so, it is responsible for facilitating synchronization meetings for identifying and resolving dependencies between Squads (“Scrum of Scrums”-like meetings). Chapters gather within the same Squad. Guilds are a more organic and wide-reaching “community of interest”, that cut across the whole organization (in opposition to a Chapter, that is local to a Tribe).

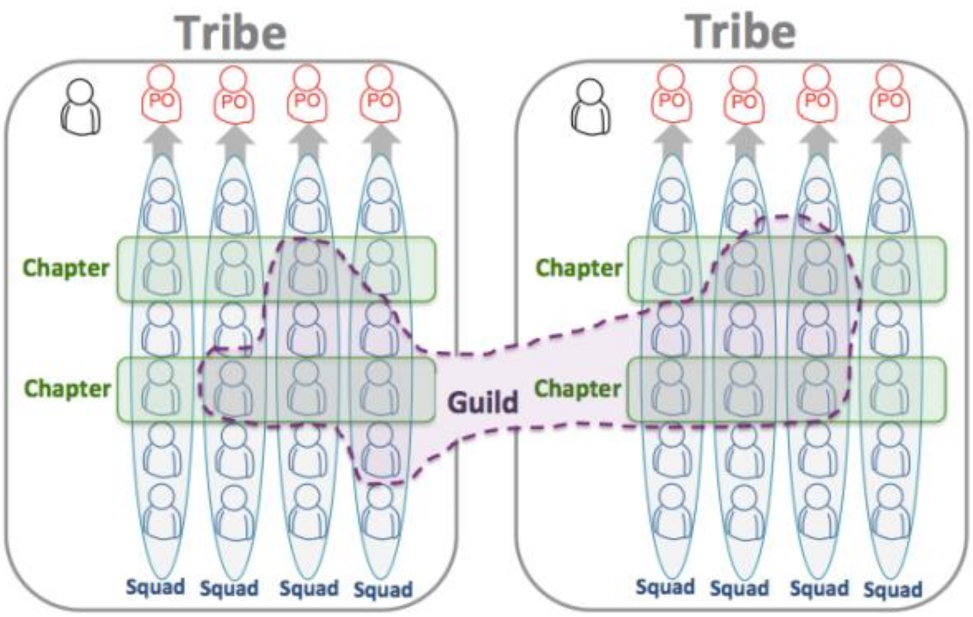


Figure 20. "Spotify model" cross-matrix structure

Agile Practices in Large-scale

Literature encompasses many differences in agile practices when applied in large-scale contexts. Scaling up the projects affect all agile practices and, for that reason, the practices that do not have relations with the architectures are out of the scope of this chapter. In this section are presented agile practices in large-scale for requirements engineering and prioritization, coordination and risk management. From the challenges depicted in **Table 2**, "*Organization of large development efforts*" encompasses requirements engineering and prioritization, "*Inter-team coordination*" encompasses coordination, "*Release planning and architecture*" encompasses requirements engineering and prioritization, and "*Agile contracts*" encompasses risk management.

Requirements Engineering & Prioritization

Requirements Engineering (RE) for agile software is different from traditional Requirements Engineering. Traditional RE is managed by RE specialists, in a phase separated in time from design and development, and documented in specific requirements artefacts. In contrast, in agile RE the detailed requirements are defined gradually in interaction between the customer (or customer representative) and the development team. The International Institute of Business Analysis (IIBA) felt the need to develop an agile version of their Business Analysis Body of Knowledge (BABoK) called "The Agile Extension to the BABoK Guide" (IIBA, 2017). Also, the International Requirements Engineering Board (IREB) coins the adoption of RE in this contexts as "RE@Agile" (IREB, 2018). Also, IREB conducted a study that investigates how the discipline of RE can be adapted to better support an agile project approach (Grau & Lauenroth, 2014), where they state that RE activities remain the same but in agile projects they are executed continuously. Engineering and management of requirements include elicitation, negotiation, prioritization, and documentation (Fernandes & Machado, 2016), whether in an ASD or in a "traditional" setting.

Five RE-related agile practices are introduced for large-scale software development, namely (Bjarnason, Wnuk, & Regnell, 2011):

- One Continuous Scope Flow. The scope for all software releases is continuously planned and managed via one priority-based list (comparable to a product backlog). The business unit gathers and prioritizes features from a business perspective. The software unit estimates the cost and potential delivery date for each feature, based on priority and available software resource capacity.

- Cross-Functional Development Teams that include a customer representative assigned by the business unit (comparable to customer proxy.) These teams have the full responsibility for defining the detailed requirements, implementing and testing a feature (from the common priority-based list) within the given boundaries of time and resources.
- Integrated RE. The RE tasks are integrated with the other development activities, i.e. the detailing and formal documentation of requirements is done at the same time as design and development of the feature and within the same (development) team together with its customer representative (proxy).
- Gradual & Iterative Detailing of Requirements. The requirements are first defined at the high level (features in the priority-based list) and then iteratively refined, by the development team, into more detailed requirements as the design and implementation work progresses.
- User Stories & Acceptance Criteria are used to formally document the requirements agreed for development. The acceptance criteria are then covered by test cases.

IIBA presents principles that guide these activities (IIBA, 2017):

- See the Whole (i.e., the context of the big picture);
- Think as a Customer (i.e., incorporate a clear understanding of the expected user experience);
- Analyze to Determine What is Valuable (i.e., continuously assess and prioritize work to be done in order to maximize the value being delivered at any point in time)
- Get Real Using Examples (i.e., using examples or models to, to identify specific details of the need and the solution, towards setting context and identifying scope)
- Understand What is Doable (continually analyzing the need and the solutions that can satisfy that need within the known constraints)
- Stimulate Collaboration and Continuous Improvement (i.e., promote Continuous structured and unstructured feedback, like Retrospectives)
- Avoid Waste (by removing activities that do not add value), *e.g.*: avoid producing documentation before it is needed, and when documentation is needed do just enough; ensure commitments are met and there are no incomplete work items that adversely impact downstream activities; avoid rework by making commitments at the last responsible moment; try to elicit, analyze, specify, and validate requirements with the same models; make analysis models as simple as possible to meet their intended purpose; ensure clear

and effective communication, and pay continuous attention to technical excellence and accuracy. Quality defects (such as unclear requirements) result in rework and are waste.

IREB states that performing RE in agile context must aim four goals (IREB, 2018):

1. knowing the relevant requirements at an appropriate level of detail (at any time during system development),
2. achieving sufficient agreement about the requirements among the relevant stakeholders,
3. capturing (and documenting) the requirements according to the constraints of the organization,
4. performing all requirements related activities according to the principles of the agile manifesto.

Reifer *et al.* state that, on large projects, some architectural development was needed before pushing ahead with iterations (Reifer et al., 2003). This could be done quickly using an architecture team. Team members then could move on to seed a LSA project's subteams. While the architecture will continue to evolve over the project's life cycle, the tendency will be to stabilize it and discourage any significant changes. However, the same authors state that this approach poses a threat to agility because it might tip the scale in favor of up-front planning rather than letting the architecture emerge naturally.

Requirements prioritization (and reprioritization) plays a crucial role in large-scale and distributed agile projects. Daneva *et al.* seek to understand the implications of these tasks in agile and distributed contexts (Daneva et al., 2013). They found the following:

- Understanding requirements dependencies is of paramount importance for the successful deployment of agile approaches in large outsourced projects.
- The most important prioritization criterion in the setting of outsourced large agile projects is risk.)
- The software organization has developed a new artefact that seems to be a worthwhile contribution to agile software development in the large: 'delivery stories', which complement user stories with technical implications, effort estimation and associated risk. The delivery stories play a pivotal role in requirements prioritization.

- The vendor's domain knowledge is a key asset for setting up successful client-developer collaboration.
- The use of agile prioritization practices depends on the type of project outsourcing arrangement

Kirikova proposes a framework for continuous requirements engineering (CRE), FREEDOM (Kirikova, 2017). It was proposed based in former applications in mobile development, within Scrum setting and within IT startups.

The FREEDOM framework consists of several functional units, namely (Figure 21): F – Future representation, R - Reality representation, E1 - requirements Engineering, E2 – fulfilment Engineering, D - Design and implementation, O - Operations, and M - Management. They are related by a number of links, which correspond to analysis, analytics, monitoring, feedback, and change request information. Requirements engineering (E1 in Figure 21) can be a sub-function of Reality representation (R), Future representation (F) and other functional units of the framework.

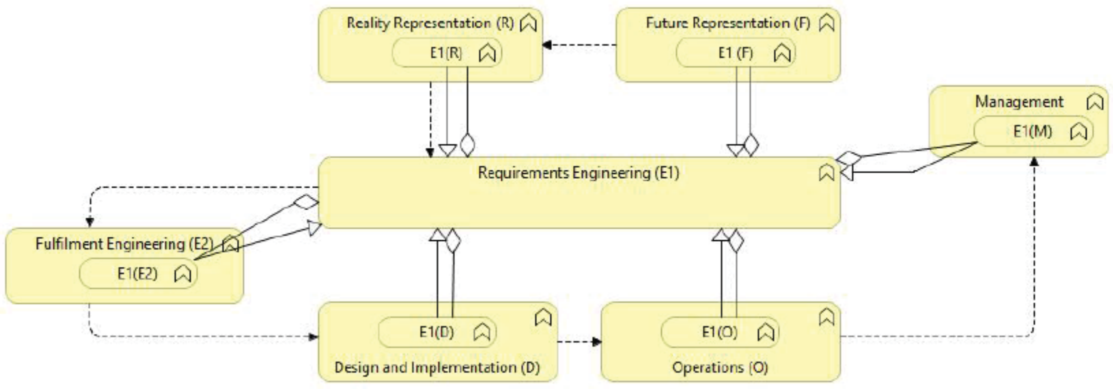

Reality Representation (R)
E1(R)
Future Representation (F)
E1 (F)
Management
E1(M)
Requirements Engineering (E1)
Fulfilment Engineering (E2)
E1(E2)
E1(D)
Design and Implementation (D)
E1(O)
Operations (O)

Figure 21. Architecture of a CRE framework ***(Kirikova, 2017)***

In Figure 22 five generic RE functions are represented, namely: requirements acquisition, requirements analysis, requirements representation, and requirements management. These functions are similar to RE functions used in "traditional" settings (Fernandes & Machado, 2016; Pohl, 2010): elicitation, analysis, specification, validation, and management. However, for CRE, Kirikova uses "Requirements acquisition" rather than "Requirements elicitation", because the term "elicitation" mainly refers to requirements acquisition using interviews, questionnaires, group sessions, or observation, but "acquisition" is more suitable since nowadays requirements are also gained using model analysis, business intelligence and data analytics methods and tools (Kirikova, 2017).

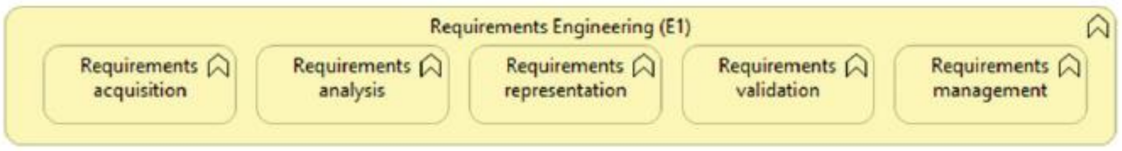


Figure 22. Generic functions of CRE *(Kirikova, 2017)*

Coordination

Most coordination practices proposed by agile methods emphasize an informal management style. When the project is small, close interactions among team members are effective and problems can be quickly spotted and corrected. However, as the size of the project increases, opportunities for close interactions among project team members drop (Van de Ven, Delbecq, & Koenig Jr, 1976). In large projects, it is difficult for developers to make important decisions only through informal conversations. Miscommunications and misunderstandings happen more often and are more difficult to solve. Large projects need to address unique challenges, such as the knowledge loss caused by turnover of team members and long project duration, complex requirements and interdependency of tasks, and limited resources (Xu & Ramesh, 2007). Relying only on informal strategies is no longer adequate. To summarize, the coordination challenges of using agile methods in large projects are (Xu, 2011):

- Lack of interaction among participants
- Communication difficulties
- Loss of knowledge
- Complex and unstable requirements
- Complex interdependency tasks
- Technical complexity

The analysis of Hossain et al. has revealed that the temporal, geographical and socio-cultural distance of software development projects impact on using various Scrum practices in distributed settings (Hossain, Babar, & Paik, 2009). They found that communication issues are the major challenges when using Scrum in distributed settings; cultural differences among distributed team members may also impact on team collaboration and communication processes; managing a large team can also be considered as one of the key challenges; and lack of dedicated meeting room for each site and Scrum team distribution at multiple sites also appear to be challenging factors that restrict the team communication and collaboration processes.

In order to face these challenges, there are some archetypes and strategies for coordination issues. Archetypes for coordination strategies in multiteam systems are classified by their mechanistic, organic and cognitive coordination (Scheerer, Hildenbrand, & Kude, 2014). The “Perfect Plan” strategy type is characterized by high mechanistic, low organic and low cognitive coordination. While within teams, coordination may well be achieved through organic or cognitive mechanisms, the focus of multiteam coordination in this strategy lies solely on mechanistic coordination with little communication between individual actors. This type assumes that software development can be “programmed” from a coordination perspective, e.g. through complete upfront planning all dependencies as well as all contingencies can be resolved and accounted for. Since the coordination is programmed through upfront planning with little communication, one person or a very small set of people, needs to have a deep insight into the full technical details of the entire software system in order to specify all details necessary for individual work packages and correct integration.

Risk Management

Boehm and Turner describe a 5-step risk-based approach (see Figure 23) for benefiting of both agile and plan-driven methods (Barry Boehm & Turner, 2003). They define tasks to evaluate and determine Commercial Off-the-Shelf (COTS), reuse, and architecture choices during Systems definition and architecting.

They include architectures in three distinctive agent-based system application projects within a case study. The three applications were classified as their scalability and criticality:

- Small, relatively noncritical. This agent-based planning system for managing events such as conferences or conventions is based on risk patterns observed in small Web-services applications.
- Intermediate. An agent-based planning system for supply-chain management across a network of producers and consumers, this application is based on risk patterns derived from the experience with scaling up XP techniques to a 50-person project in a lease-management application.

- Very large, highly critical. This agent-based planning system for national crisis management is based on risk patterns observed in the US Defense Advanced Research Project Agency and the US Army Future Combat Systems program—an agent-oriented, network-centric system of systems being developed by more than 2,000 people.

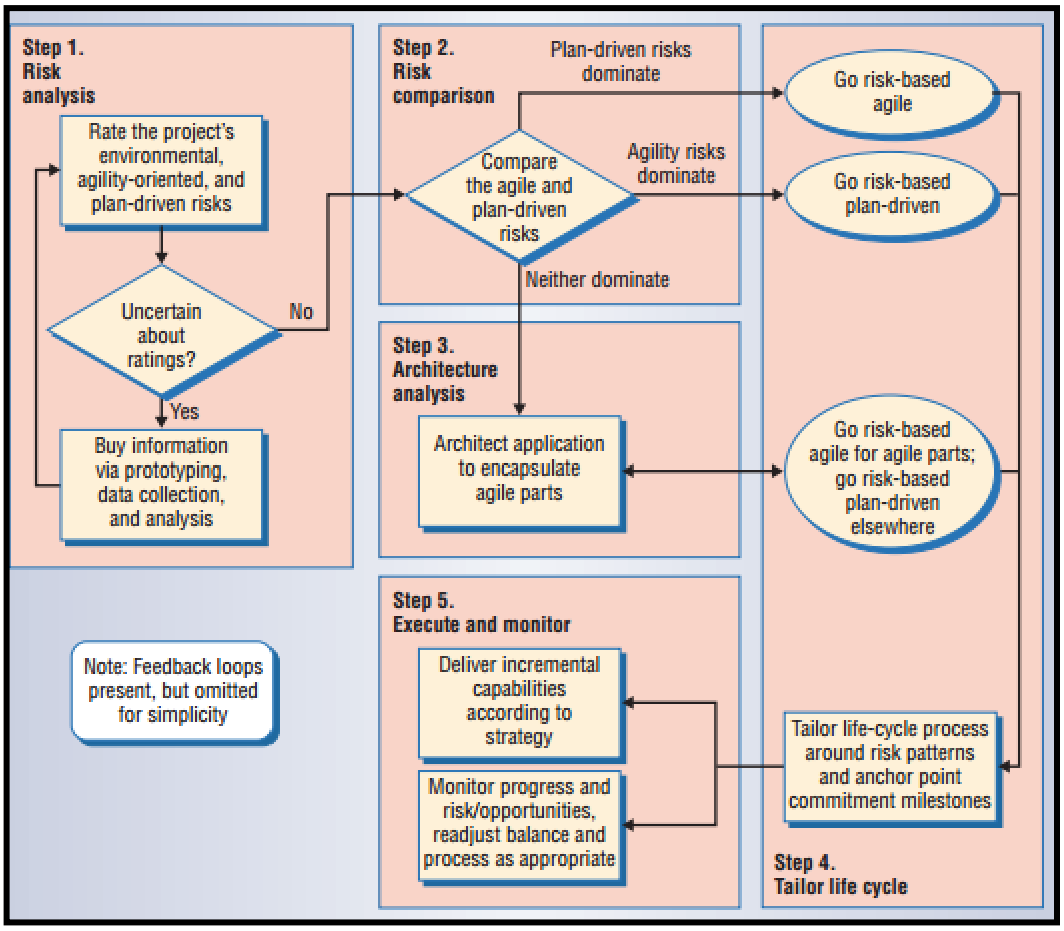

Step 1.
Risk
analysis
Rate the project's environmental, agility-oriented, and plan-driven risks
Uncertain about ratings?
No
Yes
Buy information via prototyping, data collection, and analysis
Step 2.
Risk
comparison
Plan-driven risks dominate
Compare the agile and plan-driven risks
Agility risks dominate
Neither dominate
Go risk-based agile
Go risk-based plan-driven
Step 3.
Architecture
analysis
Architect application to encapsulate agile parts
Go risk-based agile for agile parts; go risk-based plan-driven elsewhere
Note: Feedback loops present, but omitted for simplicity
Step 5.
Execute and monitor
Deliver incremental capabilities according to strategy
Monitor progress and risk/opportunities, readjust balance and process as appropriate
Tailor life-cycle process around risk patterns and anchor point commitment milestones
Step 4.
Tailor life cycle

Figure 23. Five-Step risk-based approach (Barry Boehm & Turner, 2003)

Tailoring XP for large and complex projects

Agile XP practices are suitable for large-scale, complex software development (Cao, Mohan, Xu, & Ramesh, 2004). Having as basis a set of agile practices, namely Accept multiple valid approaches, Accommodate requirements change, Engage the customer, Build on successful experience, Develop good teamwork, Effective software development conforms to project environment constraints, and Prepare for unexpected consequences from innovation in software processes, it presents a set of 7 XP practices suited for large-scale agile projects:

- Practice 1: Designing upfront. It combines designing upfront with agile practices such as short release, pair programming, and refactoring.
- Practice 2: Short Release cycles with a layered approach. The system continuously accommodates requirements changes. The delivered functionalities suit the customer need rather than focusing on documenting detailed specifications.
- Practice 3: Surrogate customer engagement. This practice is a modified version of XP practice “on site customer”.
- Practice 4: Flexible pair programming. Contrary to “always paired” in XP, developers are paired in analysis, design and testing. Coding is performed by solo programming. The combination of solo programming and pair programming overcomes some shortfalls of pair programming (*e.g.*, developer’s resistance), while still benefiting from it where feasible.
- Practice 5: Identifying and managing developers. People factor is more important in agile development than in traditional development. It emphasizes choosing the right people for the team and creating a collaborative environment to support teamwork. Developers’ knowledge and experiences on different aspects of a project are greatly valued.
- Practice 6: Reuse with forward refactoring. This practice maps to the principle of building on successful experience. Refactoring is used as a technique to enhance reuse. Developers usually focus only on their current need instead of building components for later reuse. However, for a large project, development of upfront architectural design and use of design patterns are critical. Functionalities of the system are developed based on design patterns. In addition, modules that have been developed to handle specific functionalities are refactored and made generic enough so that they can be tailored to handle different functionalities.
- Practice 7: Flatter hierarchies with controlled empowerment. Developers are empowered to make their own decisions. On the other side, for a large and mission-critical application, the empowerment might cause unexpected consequences such as incompatibilities among the development process and products produced by different developers.

Distributed Agile Teams: the Scrum of Scrums

Besides the use of Scrum in small organizations or in small projects, some techniques for adapting events, actors and artifacts arise in order to geographical distributed teams or multiple teams could work for the same product development (Eckstein, 2013). Distributed Scrum is classified in three distributed team models (Sutherland, Viktorov, & Blount, 2006): (1) Isolated Scrums – teams are isolated across geographies; (2) Scrum of Scrums – multiple Scrum teams working on the same product and in the same geographical space (Cristal, Wildt, & Prikladnicki, 2008); and (3) Totally Integrated Scrum – where multiple teams are geographical distributed (Paasivaara, Durasiewicz, & Lassenius, 2008b). The Scrum methodology was also tested in projects involving different organizations trying to implement the same product (Dingsøyr, Hanssen, Dybå, Anker, & Nygaard, 2006).

Additionally, another scaled framework from Scrum is Large-Scale Scrum (LeSS) (Larman & Vodde, 2016). This framework is one-team oriented for scaled projects within Scrum practices. LeSS includes a single Product Backlog; one Definition of Done for all teams; one Potentially Shippable Product Increment at the end of each Sprint; one Product Owner; many complete cross-functional teams (with no single-specialist teams); and a common Sprint for all Teams, every Sprint. The roles, events and artifacts of LeSS are represented in Figure 16 in Section 2.4.

Scrum uses structured meetings such as the daily Scrum meeting, the daily Scrum of Scrums meeting, the Sprint planning meeting, and the Sprint review meeting. These meetings are key components of the Scrum method and they should be adjusted to the distributed working environment (Cho, 2007). Information and knowledge-sharing issues were the most important issues in the company due to its geographically distributed working environment, where also coordination, communication, control, training, and trust and confidence issues hinder developers from being efficient (Cho, 2007). Paasivaara describes adaptation to meetings as well (Paasivaara, Durasiewicz, & Lassenius, 2008a).

Each team's parallel work within the same product development must be coordinated, but breakthroughs and progress within the distributed Scrum teams are slow and hard to achieve (Begel, Nagappan, Poile, & Layman, 2009).

Scrum roles and events can be easily adapted to such dependencies between teams since the agile methodologies provide this kind of flexibility. These adaptations allow distributed teams (geographical distributed or not) to work in parallel and at the same time, and are mandatory in order to prevent an increase of the time to market, that could endanger the project execution. The product backlog should be aligned by collaborating product owners (Leffingwell, 2007).

The role of a Scrum master should be adjusted. Since the Scrum master is the key person for the success of Scrum, the Scrum master should be engaged more actively in projects as a coordinator and a controller in distributed Scrum environment. As a coordinator/controller, the Scrum master needs to ensure share of information and knowledge well between the sites and that the tasks are divided and assigned well through the Sprint planning meeting. In addition, the Scrum master needs to have the authority to motivate developers to work hard and to take ownership of the projects. These tasks can be eased by using various tools such as Wiki, VersionOne, and JIRA (Cho, 2007).

The various Scrum meetings should be adjusted due to the geographical distance. For example, in daily Scrum meetings which are held every day for less than 15 minutes, team members come to talk about what tasks have been done since the last meeting, what tasks will be done before the next meeting, and what the issues and challenges are imposed on the tasks. Scrum team members come to daily Scrum meetings to communicate with other team members to find out what is going on. However, the effectiveness of communication between the sites is severely limited compare to face-to-face conversation. To mitigate the problem, the daily Scrum meeting and other Scrum meetings should be held with good communication devices. Among many different multi-media devices, a video conferencing system between the sites is recommended as the best way to communicate. Other tools including remote desktop, an email system, an instant message system, and a phone system can mitigate the communication problems too (Cho, 2007).

Regarding Sprints and events, it is normal that not all of team's Sprints are synchronized relating its start date. The Scrum Master and the Product Owner should have total availability to work with all teams equitably. Besides, if problem reports arise from a team in a Sprint Review or a Sprint Retrospective, there is still enough time to re-schedule aspects in other teams or re-allocate resources at the end of the other team's Sprints. Additionally, another issue that can be considered is that one element of each team may participate in other team's Sprint Review, and all elements should participate instead of always the same element participating in those Sprint Reviews, so all team elements have the opportunity to know other teams' work.

A Hybrid Method using RUP with Scrum

Scrum and RUP can be combined by embedding Scrum ceremonies (Daily Scrum meeting and Sprint meeting) and roles (Scrum Master, Team, Product Owner), and artifacts (product backlog, sprint backlog, and burndown chart) within RUP phases (Cho, 2009) (see Figure 24). In this work, the business modeling discipline is the main player in the inception phase. The analysis and the design disciplines are mostly utilized in the elaboration phase. The implementation and testing disciplines focus on the construction phase, whereas, the deployment and configuration disciplines are in the transition phase.

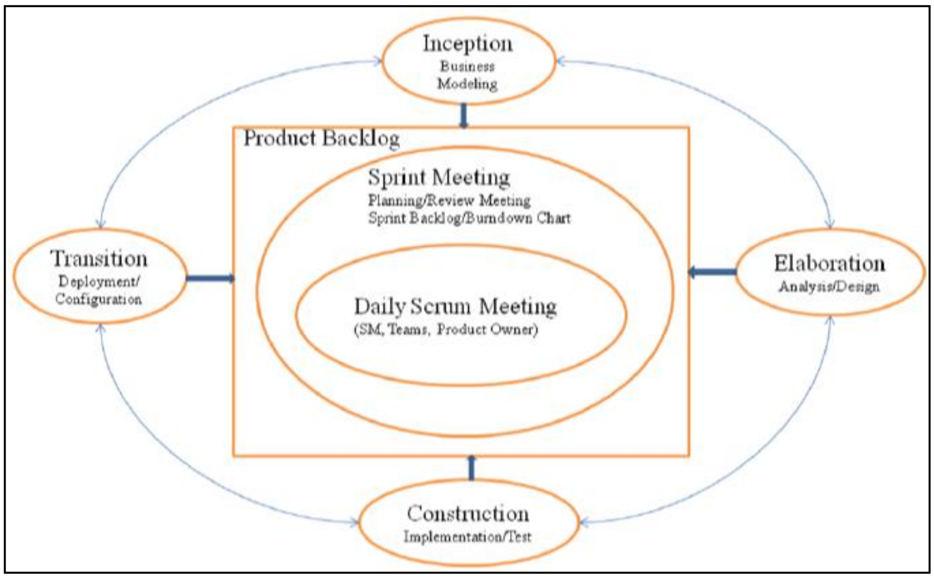


Figure 24. Hybrid model combining RUP phases and Scrum ceremonies (Cho, 2009)

The daily Scrum meeting, the daily Scrum of Scrums, the Sprint planning meeting, and the Sprint review meeting can be conducted iteratively in each RUP phase (Figure 25). The product owner can create the product log as a part of the business modeling discipline. The Scrum Master also can plays the usual role defined in the Scrum process. The tasks defined in the product backlog and the Sprint backlog can be accomplished and monitored through the daily Scrum meeting and the Sprint meeting.

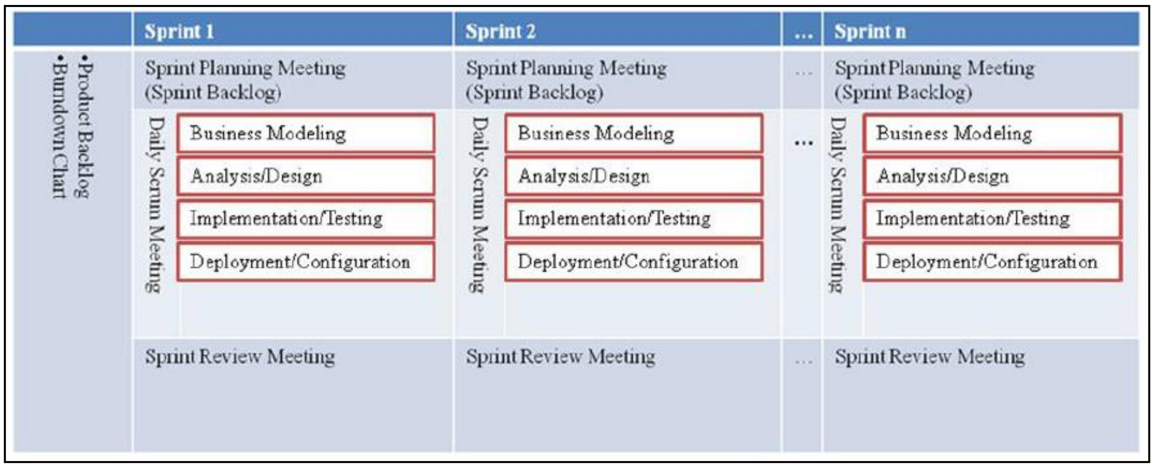


Figure 25. A Typical Phase of Hybrid Model (Cho, 2009)

2.5 Conclusions

The reviewed literature encompass applicability of software architecture methods within any of the phases in an agile project, from sprint planning, user stories, backlogs, development, testing, etc. Additionally, the aforementioned approaches relate to a diversity of agile methodologies, from XP to Crystal, FDD or Scrum (as shown in Table 3).

The aforementioned approaches showed that there is an opportunity to improve software development while maintaining a balance between agility and the architectural approach. Therefore, several approaches to integrate and embed software architecture and agile methods are proposed. These works all have in common the fact that architecture methods must perform in parallel with common agile methodologies, and the architecture itself must possess agility and flexibility enough to respond to changes rapidly.

Table 3. Applicability of the reviewed approaches

Reference	Phase	Architecture-driven Method	Agile Framework
(R. L. Nord & Tomayko, 2006)	Planning and Stories, Designing, Analysis and Testing	QAW, ADD, ATAM/CBAM	XP
(Jeon et al., 2011)	Planning and stories, and Designing	QAW, ADD	XP, Scrum
(Farhan et al., 2009)	Analysis and Testing	ATAM	XP, Crystal
(Sharifloo et al., 2008)	Planning and stories, before upcoming iteration		XP
(Kanwal et al., 2010)	All phases (Develop an Overall Model, Build a Features List, Plan by Feature, Design by Feature, Build by Feature)	QAW, ATAM, ARID	FDD
(Madison, 2010)	up-front planning, storyboarding, Sprint, and working software	communication, quality attributes, and design patterns	N/A
(Díaz et al., 2014)	Planning and stories, Sprints	Change Impact Analysis	Scrum

These approaches do not include thorough requirements specification and a logical architecture able to be used as basis for the development like 4SRS does. Additionally, the initial backlog should include both functional and quality (non-functional) requirements (typically quality ones only emerge during development), where 4SRS supports their identification by using the Model/View/Controller (MVC) pattern. It is expected that the proposed approach uses these "strengths" of 4SRS and adapt them to agile context, but also to include concerns that the presented approaches provide (change impact analysis, architecture review and assessment, and others).

This chapter essentially focused in presenting existing research regarding architecture design and large-scale agile (LSA). After a brief contextualization of how the architectural design discipline changed and coexisted with the adoption of ASD, it described existing approaches that used architecture design methods in specific stages of ASD processes and described how practices may require some change in scaled (*i.e.*, LSA) settings. The presented works allowed depicting that architecture has a specific role depending in the development stage where it is applied and, foremost, that the design is evolutionary and that the work is performed in a continuous way. The adoption of "*continuous*"-oriented approaches leads to specific concerns towards defining practices for modeling requirements and architecture. Such concerns are described in Chapter 3.

References

Abrahamsson, P., Babar, M. A., & Kruchten, P. (2010). Agility and architecture: Can they coexist? *IEEE Software*, *27*(2), 16–22. https://doi.org/10.1109/MS.2010.36

Agile Alliance. (2001). Manifesto for agile software development.

Ambler, S. (2005). The agile unified process (aup). *Ambysoft*.

Ambler, S., & Lines, M. (2012). *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise*. IBM Press.

Anderson, D. (2010). *Kanban: successful evolutionary change for your technology business*. Blue Hole Press.

Avgeriou, P., Grundy, J., Hall, J. G., Lago, P., & Mistrík, I. (2011). *Relating Software Requirements and Architectures*.

Beck, K., & Andres, C. (2004). *Extreme programming explained: embrace change*. Addison-Wesley Professional.

Begel, A., Nagappan, N., Poile, C., & Layman, L. (2009). Coordination in large-scale software teams. In *Workshop on Cooperative and Human Aspects on Software Engineering (ICSE)* (pp. 1–7). IEEE Computer Society.

Bellomo, S., Kruchten, P., Nord, R., & Ozkaya, I. (2014). How to Agilely Architect an Agile Architecture. *Cutter IT Journal*.

Bjarnason, E., Wnuk, K., & Regnell, B. (2011). A case study on benefits and side-effects of agile practices in large-scale requirements engineering. In *1st Workshop on Agile Requirements Engineering* (p. 3). ACM.

Boehm, B. (2002). Get ready for agile methods, with care. *Computer*, *35*(1), 64–69.

Boehm, B., Gray, T., & Seewaldt, T. (1984). Prototyping versus specifying: a multiproject experiment. *IEEE Transactions on Software Engineering*, *SE-10*(3), 290–303.

Boehm, B., & Turner, R. (2003). Using risk to balance agile and plan-driven methods. *Computer*, *36*(6), 57–66.

Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, *21*(5), 61–72.

Brown, N., Nord, R., & Ozkaya, I. (2010). *Enabling agility through architecture*. DTIC Document.

Cao, L., Mohan, K., Xu, P., & Ramesh, B. (2004). How extreme does extreme programming have to be? Adapting XP practices to large-scale projects. In *37th Annual Hawaii International Conference on System Sciences* (p. 10 pp.). IEEE.

Cho, J. (2007). Distributed Scrum for large-scale and mission-critical projects. In *Americas Conference on Information Systems (AMCIS)* (p. 235).

Cho, J. (2009). A hybrid software development method for large-scale projects: rational unified process with scrum. *Issues in Information Systems*, *10*(2).

Clements, P., Ivers, J., Little, R., Nord, R., & Stafford, J. (2003). *Documenting Software Architectures in an Agile World*. DTIC Document.

Cockburn, A. (2004). *Crystal clear: a human-powered methodology for small teams*. Pearson Education.

Coplien, J. O., & Bjørnvig, G. (2011). *Lean architecture: for agile software development*. John Wiley & Sons.

Cristal, M., Wildt, D., & Prikladnicki, R. (2008). Usage of Scrum practices within a global company. In *nternational Conference on Global Software Engineering (ICGSE)* (pp. 222–226). IEEE. https://doi.org/10.1109/ICGSE.2008.34

Daneva, M., Van Der Veen, E., Amrit, C., Ghaisas, S., Sikkel, K., Kumar, R., ... Wieringa, R. (2013). Agile requirements prioritization in large-scale outsourced system projects: An empirical study. *Journal of Systems and Software*, *86*(5), 1333–1353.

Díaz, J., Pérez, J., & Garbajosa, J. (2014). Agile product-line architecting in practice: A case study in smart grids. *Information and Software Technology*, *56*(7), 727–748.

https://doi.org/10.1016/j.infsof.2014.01.014

Dingsøyr, T., Fægri, T. E., & Itkonen, J. (2014). What is Large in Large-Scale? A Taxonomy of Scale for Agile Software Development. In *Product-Focused Software Process Improvement* (pp. 273–276). Springer.

Dingsøyr, T., Hanssen, G. K., Dybå, T., Anker, G., & Nygaard, J. O. (2006). Developing software with scrum in a small cross-organizational project. In *Software Process Improvement* (pp. 5–15). Springer.

Dingsøyr, T., & Moe, N. B. (2013). Research challenges in large-scale agile software development. *ACM SIGSOFT Software Engineering Notes*, *38*(5), 38–39.

Dingsøyr, T., & Moe, N. B. (2014). Towards Principles of Large-Scale Agile Development: A Summary of the workshop at XP2014 and a revised research agenda.

Dingsøyr, T., Moe, N. B., & Olsson, H. H. (2018). Towards an Understanding of Scaling Frameworks and Business Agility: A Summary of the 6th International Workshop at XP2018 (in print). In *XP2018 Scientific Workshops*. Porto, Portugal: ACM.

Eckstein, J. (2013). *Agile software development with distributed teams: Staying agile in a global world*. Addison-Wesley.

Erdogmus, H. (2009). Architecture meets agility. *IEEE Software*, *26*(5), 2–4. https://doi.org/10.1109/MS.2009.121

Falessi, D., Cantone, G., Sarcià, S., Calavaro, G., D'Amore, C., & Subiaco, P. (2010). Peaceful coexistence: agile developer perspectives on software architecture. *IEEE Software*, *27*(2), 23–25.

Farhan, S., Tauseef, H., & Fahiem, M. A. (2009). Adding agility to architecture tradeoff analysis method for mapping on crystal. In *WRI World Congress on Software Engineering (WCSE'09) - Volume 04* (Vol. 4, pp. 121–125). IEEE. https://doi.org/10.1109/WCSE.2009.405

Fernandes, J. M., & Machado, R. J. (2016). *Requirements in Engineering Projects*. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-18597-2

Ferreira, N., Santos, N., Machado, R. J., & Gasevic, D. (2013). Aligning Domain-related Models for Creating Context for Software Product Design. (G. Berlin Heidelberg, Ed.), *SWQD'13*.

Vienna, Austria: Springer-Verlag.

Fischer, G., & Schneider, M. (1984). Knowledge-based communication processes in software engineering. In *7th international conference on Software engineering* (pp. 358–368). IEEE Press.

Forsberg, K., & Mooz, H. (1991). The relationship of system engineering to the project cycle. *INCOSE International Symposium*, *1*(1), 57–65. https://doi.org/10.1002/j.2334-5837.1991.tb01484.x

Gilb, T. (1985). Evolutionary Delivery versus the "waterfall model." *ACM SIGSOFT Software Engineering Notes*, *10*(3), 49–61. https://doi.org/10.1145/1012483.1012490

Grau, B. R., & Lauenroth, K. (2014). Requirements engineering and agile development - collaborative , just enough , just in time , sustainable. *International Requirements Engineering Board (IREB)*.

Grundy, J. (2013). Foreword - Architecture vs Agile: competition or cooperation? In *Agile Software Architecture: Aligning Agile Processes and Software Architectures* (pp. xxi–xxvi). Elsevier.

Highsmith, J. A. (2000). *Adaptive Software Development*. *Addison-Wesley*.

Hossain, E., Babar, M. A., & Paik, H. (2009). Using scrum in global software development: a systematic literature review. In *Fourth IEEE International Conference on Global Software Engineering (ICGSE)* (pp. 175–184). IEEE.

IIBA. (2015). *A Guide to the Business Analysis Body of Knowledge® (BABOK® Guide) Version 3*.

IIBA. (2017). *Agile Extension to the BABOK Guide v2*. International Institute of Business Analysis.

IREB. (2018). *IREB Certified Professional for Requirements Engineering - Advanced Level RE@Agile*.

Jeon, S., Han, M., Lee, E., & Lee, K. (2011). Quality attribute driven agile development. In *9th International Conference on Software Engineering Research, Management and Applications (SERA)* (pp. 203–210). IEEE. https://doi.org/10.1109/SERA.2011.24

Kanwal, F., Junaid, K., & Fahiem, M. A. (2010). A hybrid software architecture evaluation method for fdd-an agile process model. In *International Conference on Computational Intelligence and Software Engineering (CiSE)* (pp. 1–5). IEEE.

https://doi.org/10.1109/CISE.2010.5676863

Kazman, R. (2013). Foreword - Bringing the Two Together: Agile Architecting or Architecting for Agile? In *Agile Software Architecture: Aligning Agile Processes and Software Architectures* (pp. xxix–xxx). Elsevier.

Kirikova, M. (2017). Continuous Requirements Engineering. In *International Conference on Computer Systems and Technologies - CompSysTech'17*. Ruse, Bulgaria: ACM. https://doi.org/https://doi.org/10.1145/3134302.3134304

Kniberg, H., & Ivarsson, A. (2012). *Scaling agile@ spotify*.

Kruchten, P. (2004). *The rational unified process: an introduction*. Addison-Wesley Professional.

Kruchten, P. (2007). Voyage in the agile memeplex. *Queue*, *5*(5), 38. https://doi.org/10.1145/1281881.1281893

Larman, C., & Vodde, B. (2016). Large-Scale Scrum: More with LeSS.

Leffingwell, D. (2007). *Scaling software agility: best practices for large enterprises*. Pearson Education.

Leffingwell, D. (2016). *SAFe® 4.0 Reference Guide: Scaled Agile Framework® for Lean Software and Systems Engineering*. Scaled Agile, Inc.

Madison, J. (2010). Agile architecture interactions. *IEEE Software*, *27*(2), 41–48. https://doi.org/10.1109/MS.2010.35

Mancl, D., Fraser, S., Opdyke, B., Hadar, E., & Hadar, I. (2009). Architecture in an agile world. *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. Orlando, Florida, USA: ACM. https://doi.org/10.1145/1639950.1639981

Martin, J. (1991). *Rapid application development*. Macmillan Publishing Co., Inc.

Moe, N. B., & Dingsøyr, T. (2017). Emerging Research Themes and updated Research Agenda for Large-Scale Agile Development: A Summary of the 5th International Workshop at XP2017. In *Proceedings of the XP '17 Workshops*.

Moe, N. B., Olsson, H. H., & Dingsøyr, T. (2016). Trends in Large-Scale Agile Development: : A

Summary of the 4th Workshop at XP2016. In *Proceedings of the Scientific Workshop Proceedings of XP2016 on - XP '16 Workshops* (pp. 1–4). New York, New York, USA: ACM Press. https://doi.org/10.1145/2962695.2962696

Nord, R. L., & Tomayko, J. E. (2006). Software architecture-centric methods and agile development. *IEEE Software*, *23*(2), 47–53. https://doi.org/10.1109/MS.2006.54

Nord, R., Ozkaya, I., & Kruchten, P. (2014). Agile in distress: architecture to the rescue. In T. Dingsøyr & N. B. Moe (Eds.), *International Conference on Agile Software Development (XP'14)* (pp. 43–57). Springer Verlag. https://doi.org/10.1007/978-3-319-14358-3_5

Paasivaara, M., Durasiewicz, S., & Lassenius, C. (2008a). Distributed agile development: Using Scrum in a large project. In *IEEE International Conference on Global Software Engineering (ICGSE)* (pp. 87–95). IEEE. https://doi.org/10.1109/ICGSE.2008.38

Paasivaara, M., Durasiewicz, S., & Lassenius, C. (2008b). Using scrum in a globally distributed project: a case study. *Software Process: Improvement and Practice*, *13*(6), 527–544.

Palmer, S., & Felsing, M. (2001). *A practical guide to feature-driven development*. Pearson Education.

Pohl, K. (2010). *Requirements Engineering*. Springer.

Poppendieck, M., & Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit. Addison-Wesley*.

Prause, C. R., & Durdik, Z. (2012). Architectural design and documentation: Waste in agile development? In *2012 International Conference on Software and System Process (ICSSP)* (pp. 130–134). IEEE. https://doi.org/10.1109/ICSSP.2012.6225956

Raatikainen, M., Rautiainen, K., Myllärniemi, V., & Männistö, T. (2008). Integrating product family modeling with development management in agile methods. In *Proceedings of the 1st international workshop on Software development governance* (pp. 17–20). ACM.

Reifer, D. J., Maurer, F., & Erdogmus, H. (2003). Scaling agile methods. *IEEE Software*, *20*(4), 12–14.

Royce, W. W. (1970). Managing the development of large software systems. In *IEEE WESCON* (Vol. 26). Los Angeles.

Scheerer, A., Hildenbrand, T., & Kude, T. (2014). Coordination in Large-Scale Agile Software Development: A Multiteam Systems Perspective. In *47th Hawaii International Conference on System Sciences (HICSS)* (pp. 4780–4788). IEEE. https://doi.org/10.1109/HICSS.2014.587

Schwaber, K. (1997). Scrum development process. In *Business Object Design and Implementation* (pp. 117–134). Springer. https://doi.org/10.1007/978-1-4471-0947-1_11

Schwaber, K. (2015). *Nexus Guide*. *Scrum.org*.

Sharifloo, A. A., Saffarian, A. S., & Shams, F. (2008). Embedding architectural practices into Extreme Programming. In *9th Australian Conference on Software Engineering (ASWEC)* (pp. 310–319). IEEE. https://doi.org/10.1109/ASWEC.2008.4483219

Stapleton, J. (1997). *DSDM, dynamic systems development method: the method in practice*. Cambridge University Press.

Sutherland, J. (2018). *The Scrum@Scale Guide - The Definitive Guide to Scrum@Scale: Scaling that Works, Version 1.0*.

Sutherland, J., Viktorov, A., & Blount, J. (2006). Adaptive Engineering of Large Software Projects with Distributed/Outsourced Teams. In *Proc. International Conference on Complex Systems, Boston, MA, USA* (pp. 25–30).

Uludag, O., Kleehaus, M., Caprano, C., & Matthes, F. (2018). Identifying and Structuring Challenges in Large-Scale Agile Development Based on a Structured Literature Review. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)* (pp. 191–197). IEEE. https://doi.org/10.1109/EDOC.2018.00032

Uludag, O., Kleehaus, M., Xu, X., & Matthes, F. (2017). Investigating the Role of Architects in Scaling Agile Frameworks. In *2017 IEEE 21st International Enterprise Distributed Object Computing Conference (EDOC)* (pp. 123–132). IEEE. https://doi.org/10.1109/EDOC.2017.25

Vaidya, A. (2014). Does DAD know best, is it better to do LeSS or just be SAFe? Adapting scaling agile practices into the enterprise. In *Pacific NW Software Quality Conference (PNSQC)*. PNSQC.org.

Van de Ven, A. H., Delbecq, A. L., & Koenig Jr, R. (1976). Determinants of coordination modes within organizations. *American Sociological Review*, 322–338.

Waterman, M., Noble, J., & Allan, G. (2012). How much architecture? Reducing the up-front effort. In *AGILE India* (pp. 56–59). IEEE. https://doi.org/10.1109/AgileIndia.2012.11

Xu, P. (2011). Coordination in large agile projects. *Review of Business Information Systems (RBIS)*, *13*(4).

Xu, P., & Ramesh, B. (2007). Software process tailoring: an empirical investigation. *Journal of Management Information Systems*, *24*(2), 293–328.

Chapter 3 - Logical Architectures within Agile and “Continuous” Approaches

This chapter describes how the design is evolutionary and that the work addresses specific concerns towards defining practices for modeling requirements and architecture within the “continuous” paradigm. This chapter starts by presenting the software architecture lifecycle. Afterwards, it describes the “continuous” practices, starting with continuous software engineering approaches, then continuous architecting approaches and architecture management and debt. For easing the continuous architecting and management, this chapter also describes microservices architectures and their modeling. This chapter ends with the conclusions of the previously presented works.

Chapter 3 - Logical Architectures within Agile and "Continuous" Approaches

"*Meaningful architecture is a living, vibrant process of deliberation, design, & decision, not just documentation.*"

***Grady Booch**, co-author of UML*

3.1. Introduction

If, on one hand, the previous section focused on approaches for delivering products or services using agile practices supported by architectural methods, on the other hand this section points out that the product delivery is a continuous process.

While there are frameworks for "delivering the product right" (e.g., Scrum, XP, etc.) in short cycles, ASD is about getting feedback, learning and adapting. It is also about "delivering the right product". It is in this sense that approaches like "Lean Startup" (Ries, 2011) and "Lean Six Sigma" (George & George, 2003), by using tools like "Plan - Do - Check - Act or Adjust" (PDCA) and "Define, Measure, Analyze, Improve and Control" (DMAIC) cycles, suggest that product companies must get feedback and learn from customers after a product is deployed in the marketplace. Such feedback is conducted under controlled experiments, such as "A/B testing" (Kohavi & Longbotham, 2017). The GROWS method (Hunt, 2015) is another example of learning-oriented ASD.

Modern Agile[2] framework was created by a community of practitioners that aim to modernize ASD practices. It also presents revised principles of the ones from the Agile Manifesto, namely (Figure 26) (Kerievsky, 2016): "Make people awesome" (i.e., "Customer Obsession" by figuring out what's holding them back and making essential changes to help them achieve awesome results); "Make safety a prerequisite" (i.e., learn blamelessly from failures and quickly improve); "Experiment and learn rapidly" (fail fast and quickly move on to new experiments in order to achieve continuous improvement); and "Deliver value continuously" (a safe, continuous deployment pipeline lowers stress by making releasing an automated event).

[2] http://modernagile.org/

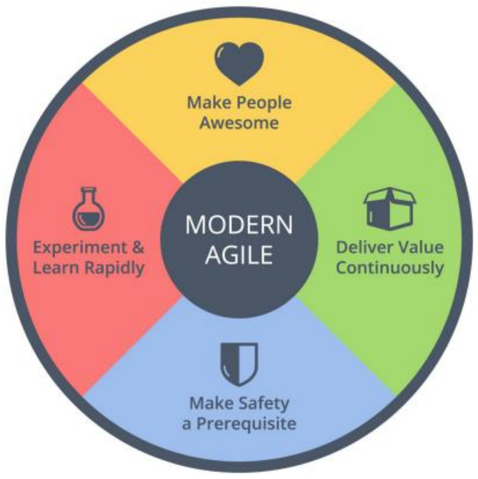

Make People Awesome
Experiment & Learn Rapidly
MODERN AGILE
Deliver Value Continuously
Make Safety a Prerequisite

Figure 26. Modern Agile

Heart of Agile[3] was created by Alistair Cockburn, one of the authors of the Agile Manifesto, in an attempt to go back to the essence of the Agile Manifesto, based in four actions: Collaborate; Deliver; Reflect; Improve.

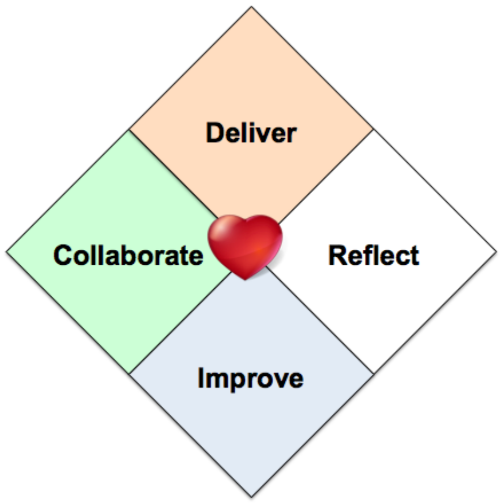


Figure 27. Heart of agile

The four actions can be further expanded using the Japanese concept "Shu-Ha-Ri" of skill progression in training and learning (Novack, 2016)., The diagram extends more specific actions to complement the four primary actions (*Learning and Income* extending *Deliver*, *Insights and Improvements* extending *Reflect*, *Collaboration and Trust* extending *Collaborate*, and *Experiment*

[3] http://heartofagile.com

and Change extending *Improve*). In addition, it highlights the need to “return to center” where the heart of agile resides (the fourth stage “Kokoro”, meaning "heart" in Japanese).

Heart of Agile and Modern Agile have some similarities, since both concepts (Ageling, 2018):

- have a solid foundation in the original Agile Manifesto
- are lightweight
- are based on four principles
- are easy to understand, with a lot behind them
- are going back to the core

There are differences though. While Heart of Agile is about returning to the essence of the Agile Manifesto, Modern Agile claims to be an evolved version of it. Additionally, Heart of Agile guides on doing and mastering things, while Modern Agile’s principles are primarily to reflect on the present and the future.

Heart of Agile and Modern Agile both are inspirational concepts, bringing “Agile” back to a core. Both with a different perspective. And they are not mutually exclusive (Ageling, 2018). Modern Agile principles can help identifying topics to improve or enhance and Heart of Agile can guide this journey of improvement.

This paradigm has required the software teams to adopt “continuous”-oriented approaches for delivering software products, i.e., “continuous software engineering” (CSE) (Bosch, 2014). This way of working originated from the adoption of continuous integration (CI) and continuous deployment (CD) practices. Performing both practices properly – CI and CD – allows companies to have a continuous delivery environment. Inside the CSE, a widely used approach towards continuous delivery is DevOps (Loukides, 2012), linking software “development” to “operations” (or maintenance) after deployment to the marketplace. Similar linkage between “development” and the “business analysis” is proposed by BizDev (Fitzgerald & Stol, 2017), and the three disciplines are linked in BizDevOps (Gruhn & Schäfer, 2015).

Following the “continuous”-oriented paradigm, in order to define properly support CI and CD practices, practices such as continuous requirements engineering (CRE) (Kirikova, 2017) and continuous architecture (Erder & Pureur, 2015) arose. Together, they allow a “full-cycle” support of continuous practices. Fitzgerald and Stol called it “Continuous ∗” (i.e. Continuous Star)

(Fitzgerald & Stol, 2017). CRE and CA are discussed in this section, since the scope of this thesis focuses on these requirements and architecture discipline, rather than CI or CD, for instance.

Regarding RE, Pohl identifies two perspectives on continuity of requirements engineering (Pohl, 2010): continuous changes in the business environment, and requirements engineering activities in each phase of the systems development lifecycle. Specifically within the second one, RE is performed as a sub-function of future representation, reality representation, fulfilment Engineering, design and implementation, operations, and management (Kirikova, 2017). Of course, it mainly relates to performing RE continuously rather than only in an initial upfront effort (Grau & Lauenroth, 2014). The effort of avoiding upfront RE, from practices and principles proposed by entities like International Institute of Business Analysis (IIBA, 2017) and International Requirements Engineering Board (IREB, 2018), has been presented in chapter 2.

In terms of architecture and design, the "continuous" paradigm is called Continuous architecture (CA) (Erder & Pureur, 2015). CA is an architectural approach that can encompass continuous delivery, providing it with a broader architectural perspective. The CA principle recommends delaying design decisions until they are absolutely necessary (Erder & Pureur, 2015). The developed system should be architected to enable changes, leveraging "The Power of Small". Moreover, the systems should be architected with a special focus on the build, test, and deploy phases.

Finally, the CA principle also suggests following Conway's law (Conway, 1968), modeling the organization of the development teams after the design of the system they are working on. Migration to microservices (Newman, 2015) is one of the most common situations when companies adopt continuous architecting processes (Davide Taibi, Lenarduzzi, & Pahl, 2017). In an era where software solutions are more and more cloud-based, microservices architectures provide many benefits in CA and CI/CD (and DevOps). Thus, continuously architect microservices is within the scope of this thesis.

3.2. Architecture lifecycle and viewpoints

Software architecture lifecycle overview

Architecture design includes from conceptual level to more refined one (Fernandes & Machado, 2016). Such argument is in line with the design process proposed by Douglass: architectural, mechanistic, and detailed (Douglass, 1999). Architectural design defines the strategic decisions that affect most or all the software components, such as concurrency model and the distribution of components across processor nodes. Mechanistic design elaborates individual collaborations by adding "glue" objects to bind the mechanism together and optimize its functionality. Such objects include containers, iterators, and smart pointers. Detailed design defines the internal structure and behavior of individual classes. This includes internal data structuring and algorithm details. These three levels of design are depicted in Figure 28.

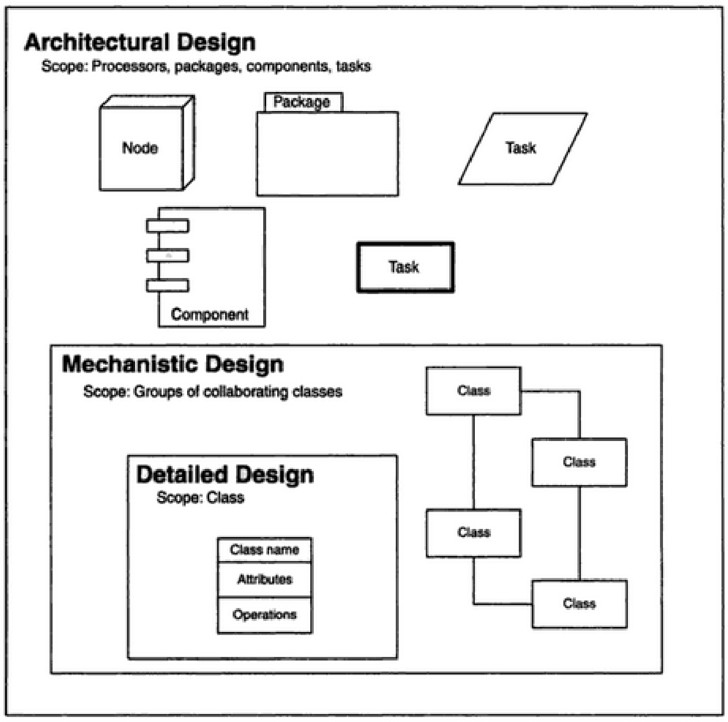


Figure 28. Three levels of architectural design (Douglass, 1999)

During a SDLC, the architecture aims different inputs, target-users and viewpoints at each stage. Other authors, like Kazman, Nord and Klein explore the use of architecture during SDLC stages (Kazman, Nord, & Klein, 2003) - Business needs and constraints, Requirements, Architecture design, Detailed design, Implementation, Testing, Deployment, and Maintenance (Table 4). Within these stages, they described the different goals of architecture-based activity and how architecture-centric methods, namely the Architecture Tradeoff Analysis Method (ATAM), the Quality Attribute Workshop (QAW), the Cost-Benefit Analysis Method (CBAM), Active Reviews for Intermediate Designs (ARID), and the Attribute-Driven Design (ADD) method contribute to those stages (Table 5).

An architecture has a particular scope. It may relate from software, hardware, organization or information, the overall system which encompasses all four, or the enterprise that hosts the system (or will host a future system to be developed) (Eeles & Cripps, 2009). Within an organization or development project, many architectural viewpoints are defined, each more suitable for a given user but all related to each other. The "4+1" View Model (Kruchten, 1995) is one of the widely known architecture model, which presents the logical, process, physical, development and scenarios views. Other views like Siemens' Five-view Model (Soni, Nord, & Hofmeister, 1995), Reference Model of Open Distributed Processing (RM-ODP) (ISO, 1998), NIST Enterprise Architecture Model (Fong & H., 1989), Department of Defense Architecture Framework (DoDAF) (DoD, 2009) or the *Zachman Framework*™ (Zachman, 2011) present relations between these viewpoints. Urbaczewski and Mrdalj compare some of these frameworks (Table 6) in order to provide context for their suitability (Urbaczewski & Mrdalj, 2006).

Table 4. Architecture-based activities within a SDLC *(Kazman et al., 2003)*

Life-Cycle Stage	Architecture-Based Activity
Business needs and constraints	• Create a documented set of business goals: issues/environment, opportunities, rationale, and constraints using a business presentation template.
Requirements	• Elicit and document six-part quality attribute scenarios using general scenarios, utility trees, and scenario brainstorming.
Architecture design	• Design the architecture using ADD. • Document the architecture using multiple views. • Analyze the architecture using some combination of the ATAM, ARID, or CBAM.
Detailed design	• Validate the usability of high-risk parts of the detailed design using an ARID view.
Implementation	
Testing	
Deployment	
Maintenance	• Update the documented set of business goals using a business presentation template. • Collect use case, growth, and exploratory scenarios using general scenarios, utility trees, and scenario brainstorming. • Design the new architectural strategies using ADD. • Augment the collected scenarios with a range of response and associated utility values (creating a utility-response curve); determine the costs, expected benefits, and ROI of all architectural strategies using the CBAM. • Make decisions among architectural strategies based on ROI, using the CBAM results.

Table 5. Use of architecture-centric methods within a SDLC *(Kazman et al., 2003)*

Life-Cycle Stage	QAW	ADD	ATAM	CBAM	ARID
Business needs and constraints	Input	Input	Input	Input	
Requirements	Input; Output	Input	Input; Output	Input; Output	
Architecture design		Output	Input; Output	Input; Output	Input
Detailed design					Input; Output
Implementation					
Testing					
Deployment					
Maintenance				Input; Output	

Table 6. A comparison of enterprise architecture frameworks (Urbaczewski & Mrdalj, 2006)

SLDC Phase/ Framework	Planning	Analysis	Design	Implementation	Maintenance
Zachman	Yes	Yes	Yes	Yes	No
DoDAF	Yes	Yes	Yes	Describes final products	No
FEAF	Yes	Yes	Yes	Yes	Detailed Subcontractor's view
TEAF	Yes	Owner's Analysis	Yes	Yes	No
TOGAF		Principles that support decision making across enterprise; provide guidance of IT resources; support architecture principles for design and implementation			

A software architecture, just like a software project, has a lifecycle. The Software Architecture Development Life Cycle (SADLC) (Reddy, Govindarajulu, & Naidu, 2007) has inputs from the business architecture or from software development life cycle for performing its architecture analysis and design. Then, the SADLC follows Software Architecture Analysis, Architecture design, Evaluation of design and ending in the Implementation of the Architecture (Figure 29). In the SADLC, the control moves from spiral model to the architectural issues area with design (SDLC) information and resolve all architectural issues (Figure 30).

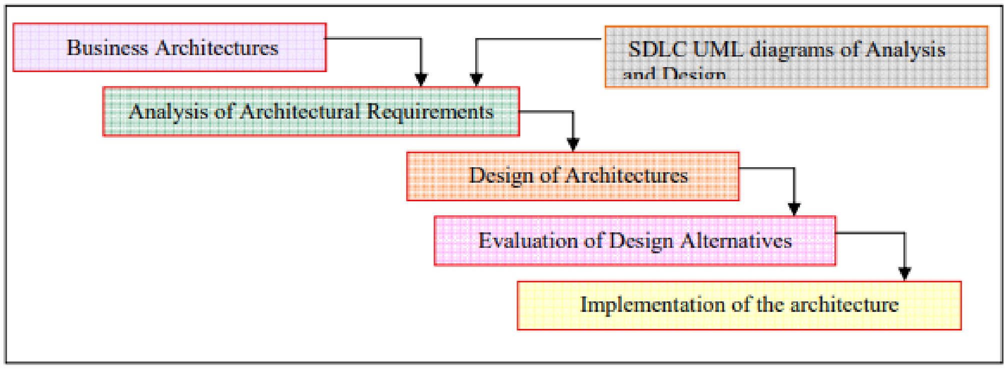

Business Architectures
SDLC UML diagrams of Analysis and Design
Analysis of Architectural Requirements
Design of Architectures
Evaluation of Design Alternatives
Implementation of the architecture

Figure 29. The Software Architecture Development Life Cycle (SADLC) ***(Reddy et al., 2007)***

The OMG modeling infrastructure, or Four-Layer Architecture, comprises a hierarchy of model levels just in compliance with the foundations of MDD (Model-Driven Development) (Atkinson & Kuhne, 2003). Each model in the Four-Layer Architecture (except for the one at the highest level) is an instance of the one at the higher level, which range from M0 to M3. The first level (*user data*), i.e., M0, refers to the data manipulated by software. Models of user data - one level above - are called *user concepts* models, i.e., M1. Models of user concepts models are *language concepts* models, i.e., M2. These are models of models and so are called metamodels. A metamodel is a model of a modeling language. It is also a model whose elements are types in another model. It describes the structure of the different models that are part of it, the elements that are part of those models and their respective properties. The *language concepts metamodels* are at the highest level of the modeling infrastructure. The hierarchy of models is as follows: M3 – Language concepts metamodels: M2 – Language concepts: M1 – User concepts: M0 – User data:

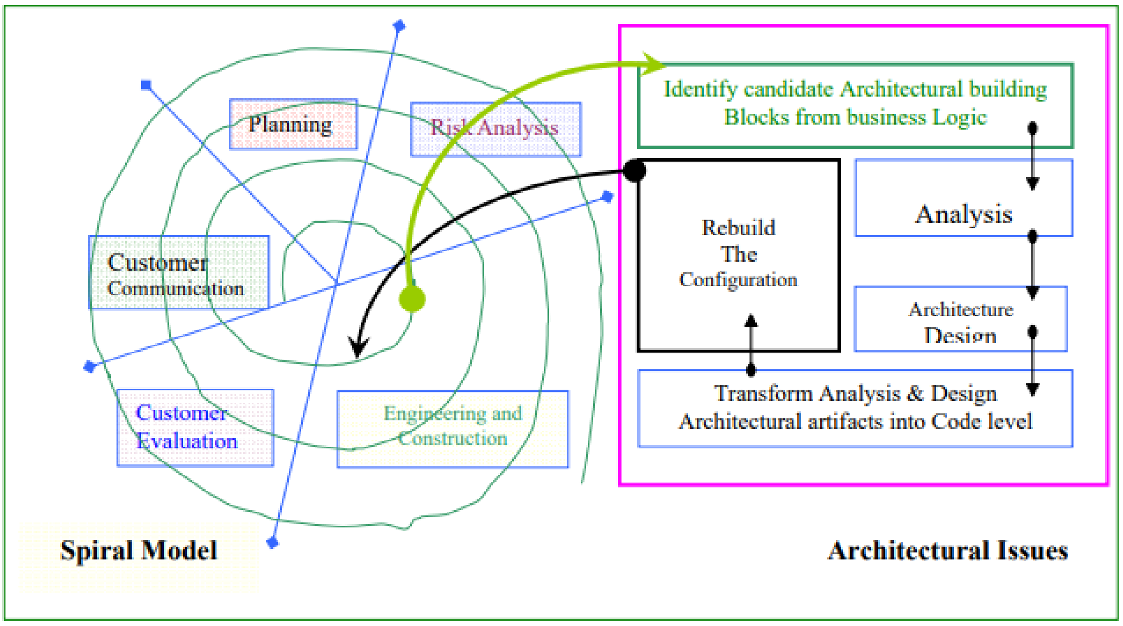

Planning
Risk Analysis
Customer Communication
Customer Evaluation
Engineering and Construction
Spiral Model
Identify candidate Architectural building Blocks from business Logic
Rebuild The Configuration
Analysis
Architecture Design
Transform Analysis & Design Architectural artifacts into Code level
Architectural Issues

Figure 30. Proposing architectural issues within the Spiral-based SADLC ***(Reddy et al., 2007)***

In comparison, the Model-driven Architecture (MDA) (OMG, 2003) proposes a hierarchical structure for model abstraction, namely:

- Computation-independent model (CIM)
- Platform-independent model (PIM)
- Platform-specific model (PSM)

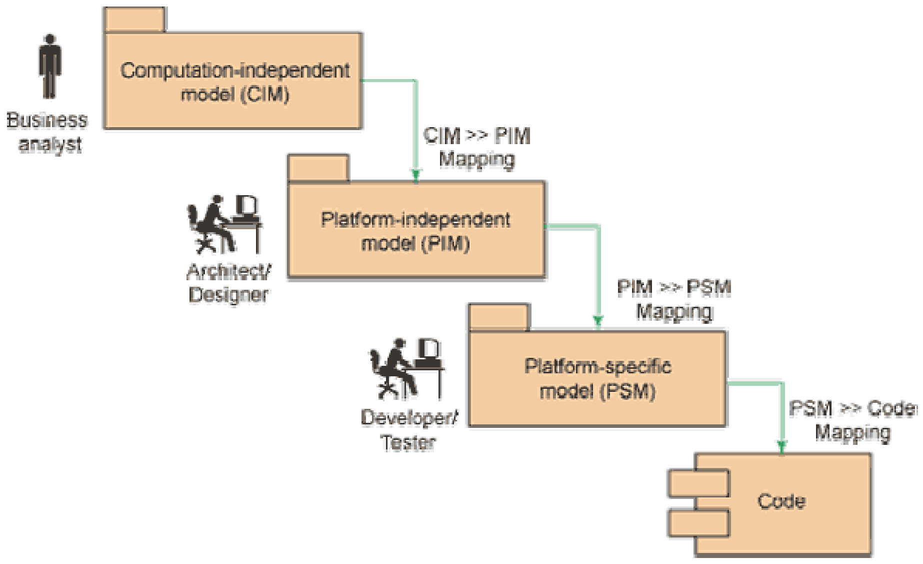


Figure 31. MDA-based model abstraction ***(Dodani, 2006)***

Software architecture methods

There are approaches that support assure an alignment of the architecture with other relevant information. Some known approaches are Reuse-Driven Software Engineering Business (RSEB) (Jacobson, Griss, & Jonsson, 1997), Family-oriented Abstraction Specification and Translation (FAST) (Weiss, 1999), Feature-Oriented Reuse Method (FORM) (Kang et al., 1998), *Komponentenbasierte Anwendungsentwicklung* (KobrA – that is German for "component-based application development") (Bayer, Muthig, & Göpfert, 2001), Quality-driven Architecture Design (QADA) (Matinlassi, Niemelä, & Dobrica, 2002), Product Line Software Engineering (PulSE) (Bayer, Flege, & Knauber, 1999). All the methods previously stated have the objective of supporting the design of a software architecture based in elicited information related to software requirements.

Software architecture classification levels

In this section, we propose a classification schema for architecture viewpoints, were the viewpoints are described under a set of proposed dimensions for the schema (Figure 32), namely the viewpoint framework, the software development phase and the level of abstraction.

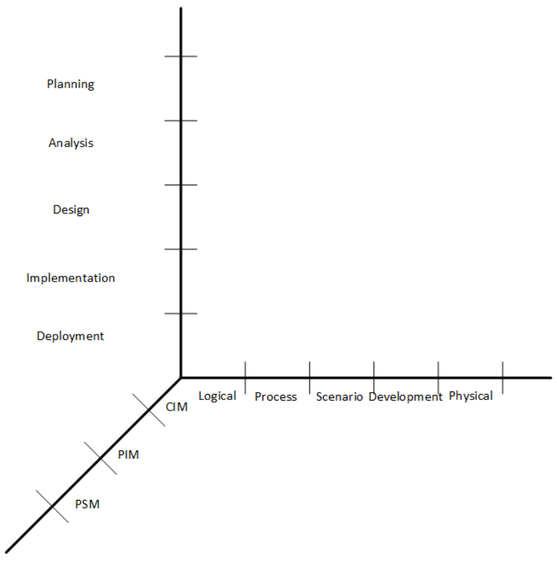


Figure 32. Classification schema

For the viewpoint framework, the classification schema uses Kruchten's 4+1 framework (Kruchten, 1995). The adoption of this framework over the remaining is due to its widely known.

However, for a better classification, the architecture levels are also fitted in other viewpoint frameworks: NIST EA Model (Fong & H., 1989), Siemens Five-view Model (Soni et al., 1995), SEI's viewpoint and target audience framework (P. C. Clements & Northrop, 1996), ISO RM-ODP (ISO, 1998), Rozanski & Woods Viewpoint Catalog (Rozanski & Woods, 2005), DoDAF 2.0 (DoD, 2009) and Zachman Enterprise Ontology (Zachman, 2011).

The classification is proposed by analyzing its fitting within the schema dimensions based in found definitions in literature. We propose a set of architecture levels. These levels refer to the architecture's scope, as well as context of use. The proposal was based in a comparison and likeness of the viewpoints from the presented frameworks, depicted in **Table 7**. Additionally, the levels were grouped in

Table 8): Concepts, Information systems, Software systems, and Infrastructure.

Table 7. Architecture viewpoints categories

Concepts	**Inf. Systems**	**Software Systems**	**Infrastructure**
Conceptual architecture Reference architecture	Enterprise architecture Process architecture Information system architecture	Logical architecture Component architecture Data models / Classes Technical architecture	Deployment architecture Physical architecture

Table 8. Comparison and likeness of architecture viewpoints

NIST EA Model	**Kruchten 4+1 Model**	**Siemens Five-view Model**	**SEI's viewpoint and target audience**	**ISO RM-ODP** (ISO, 1998)	**Viewpoint Catalog**	**DoDAF 2.0**	**Zachman Enterprise Ontology**
Business	Scenarios	Conceptual	Conceptual	Enterprise	Context	All	Scope
Information		Module	Module	Information	Functional	Capability	Concepts
	Logical						Logic
Information Systems	Process		Process		Information	Data and Information	
Data					Concurrency	Operational	Physics
Delivery Systems				Engineering		Project	Technology
	Development	Code		Computation	Development	Services	
		Execution			Deployment	Standards	Product
	Physical	Hardware	Physical	Technical	Operational	Systems	

Level 1 – conceptual level

The conceptual architecture is derived from business requirements and are understood and supported by senior management (D. Chen, Doumeingts, & Vernadat, 2008). With this definition in mind, the conceptual architecture can be classified as:

Conceptual – **Phase**: Planning; **4+1**: Logical; **Abstraction**: CIM. (Others: **NIST EA Model**: Business; **Siemens Five-view Model**: Conceptual; **SEI’s viewpoint and target audience:** Conceptual; **ISO-RM ODP**: Information; **Viewpoint Catalog**: Context; **DoDAF 2.0**: Capability; **Zachman**: Concepts.)

A reference architecture is a framework in which system related concepts are organized (Zwegers, 1998). It is also defined as a set of coherent engineering and design principles used in a specific domain. It aims at structuring the design of a specific system architecture by defining a unified terminology, the structure of the system, responsibilities of system components, by providing standard (template) components, by giving examples, etc. (Brussel, Wyns, Valckenaers, Bongaerts, & Peeters, 1998). With this definition in mind, the reference architecture can be classified as:

Reference – **Phase**: Planning; **4+1**: Logical; **Abstraction**: CIM. (Others: **NIST EA Model**: N/A; **Siemens Five-view Model**: Conceptual; **SEI’s viewpoint and target audience:** Conceptual; **ISO-RM ODP**: Information; **Viewpoint Catalog**: Functional; **DoDAF 2.0**: Standards; **Zachman**: Scope.)

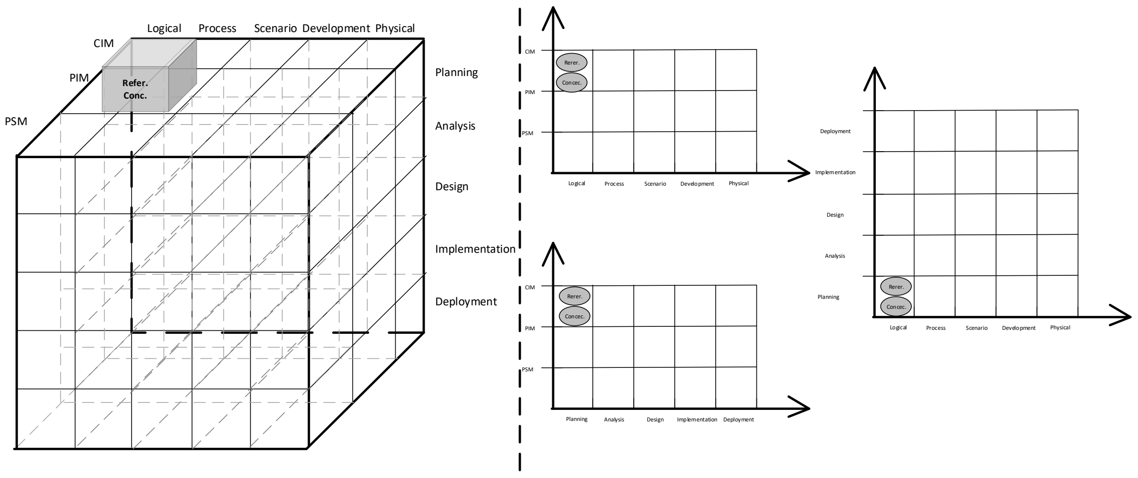


Figure 33. Viewpoints classifications at Level 1

Level 2 – Information systems level

An enterprise architecture tries to describe and control an organization's structure, processes, applications, systems and techniques in an integrated way (Lankhorst, 2009). Enterprise architecture provides a way to enable cross-functional, cross-discipline collaboration essential to articulating and implementing strategic business requirements. A coherent description of enterprise architecture provides insight, enables communication among stakeholders and guides complicated change processes (H. Chen, 2008). With this definition in mind, the enterprise architecture can be classified as:

Enterprise – **Phase**: Analysis; **4+1**: Process; **Abstraction**: CIM. (Others: **NIST EA Model**: Business; **Siemens Five-view Model**: Conceptual; **SEI's viewpoint and target audience:** Conceptual; **ISO-RM ODP**: Enterprise; **Viewpoint Catalog**: Context; **DoDAF 2.0**: Capability; **Zachman**: Scope.)

The process architecture represents the fundamental organization of service development, service creation, and service distribution in the relevant enterprise context (Winter & Fischer, 2006). A process architecture can also be defined as an arrangement of the activities and their interfaces in a process (Browning & Eppinger, 2002). Process architecture ensures that all the relevant information, which consists of the foundation and guidelines for the process review and improvement, are made explicit and can be referred to (Jeston & Nelis, 2008). With this definition in mind, the process architecture can be classified as:

Process – **Phase**: Analysis; **4+1**: Process; **Abstraction**: CIM. (Others: **NIST EA Model**: Information Systems; **Siemens Five-view Model**: N/A; **SEI's viewpoint and target audience:** Process; **ISO-RM ODP**: N/A; **Viewpoint Catalog**: N/A; **DoDAF 2.0**: Capability; **Zachman**: Concepts.)

An information system architecture is about the logical constructs for controlling and defining interfaces and for integrating the components that compose the system, which for information systems relates to an entire enterprise (Zachman, 1987). It must represent the concepts in the real world that are part of the information system and its development (Sowa & Zachman, 1992). Additionally, it must provide a linkage between information strategy and business strategy (Zachman, 1987). Ferreira et al. use an information system logical architecture, where its logical components relate to activities performed within the information system (Ferreira, Santos, Machado, & Gašević, 2012). With this definition in mind, the information systems architecture can be classified as:

Inf. System – **Phase**: Design; **4+1**: Logical; **Abstraction**: PIM. (Others: **NIST EA Model**: Business; **Siemens Five-view Model**: N/A; **SEI's viewpoint and target audience:** Process; **ISO-RM ODP**: Information; **Viewpoint Catalog**: Information; **DoDAF 2.0**: Data and Information; **Zachman**: N/A.)

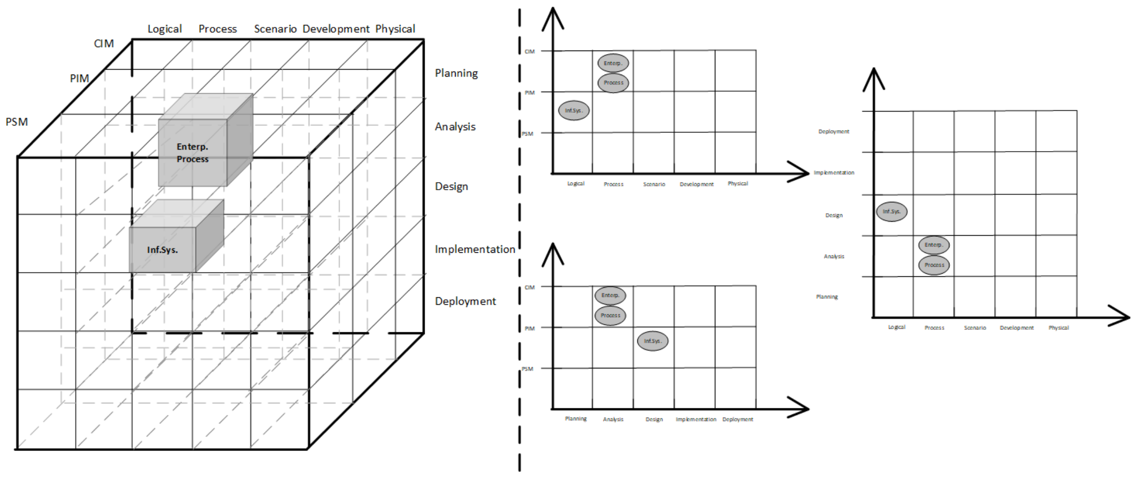


Figure 34. Viewpoints classifications at Level 2

Level 3 – Software features

A system logical architecture can be viewed as a constructed set of the system's design decisions (Ferreira, Santos, Machado, Fernandes, & Gasević, 2014), a view of a system composed of a set of problem-specific abstractions supporting functional requirements and suiting the purpose of identifying common design elements across the different parts of a system (Kruchten, 1995), a module view representing the static structure of the software system (the system's functional blocks) (P Clements, Garlan, Little, Nord, & Stafford, 2003). It is a design artifact representing a functionality-based structure of the system being designed (Azevedo, 2014), represented as objects or object classes (Kruchten, 1995) or as components (Azevedo, 2014). With this definition in mind, the logical architecture can be classified as:

Logical – **Phase**: Design; **4+1**: Logical; **Abstraction**: PIM. (Others: **NIST EA Model**: N/A; **Siemens Five-view Model**: Module; **SEI's viewpoint and target audience:** Module; **ISO-RM ODP**: Information; **Viewpoint Catalog**: Functional; **DoDAF 2.0**: Operational; **Zachman**: Logic.)

Component is a modular unit with well-defined Interfaces that is replaceable within its environment (UML, 2011). With this definition in mind, the component architecture can be classified as:

Component – **Phase**: Design; **4+1**: Development; **Abstraction**: PIM. (Others: **NIST EA Model**: Data; **Siemens Five-view Model**: Development; **SEI's viewpoint and target audience:** Code; **ISO-RM ODP**: Computation; **Viewpoint Catalog**: Development; **DoDAF 2.0**: Services; **Zachman**: Technology.)

A class architecture provides a classification of objects and to specify the Features that characterize the structure and behavior of those objects (UML, 2011). With this definition in mind, the class architecture can be classified as:

Data / Class – **Phase**: Design; **4+1**: Development; **Abstraction**: PIM. (Others: **NIST EA Model**: Data; **Siemens Five-view Model**: Development; **SEI's viewpoint and target audience:** Code; **ISO-RM ODP**: Computation; **Viewpoint Catalog**: Concurrency; **DoDAF 2.0**: Standards; **Zachman**: Technology.)

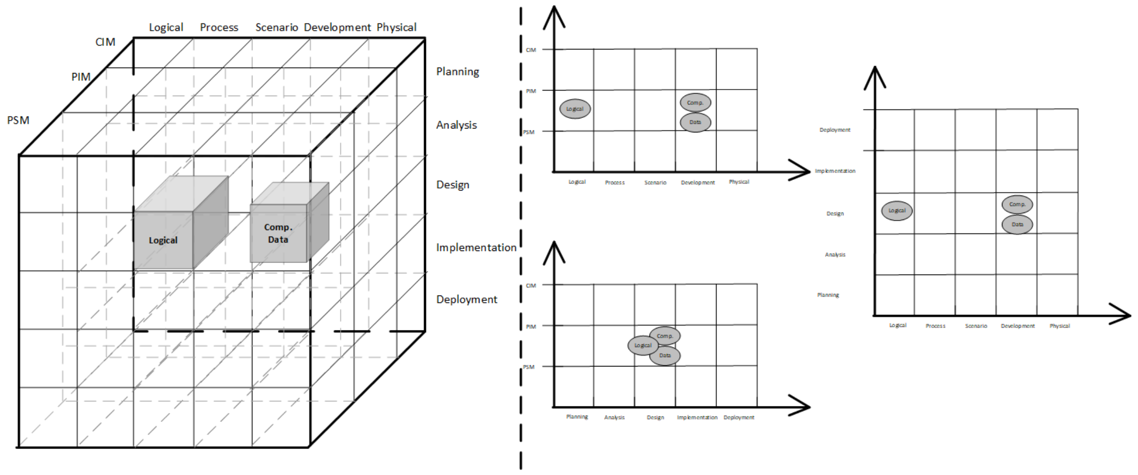


Figure 35. Viewpoints classifications at Level 3

Level 4 – IT infrastructures

The technical architecture provides the technical components that enable the business strategies and functions (D. Chen et al., 2008). It describes the capabilities that are required to support the deployment of business, data, and application services, in terms of IT infrastructure, middleware, networks, communications, processing, standards, and so on (Booch, 2010). With this definition in mind, the technical architecture can be classified as:

Technical – **Phase**: Implementation; **4+1**: Development; **Abstraction**: PSM. (Others: **NIST EA Model**: Delivery Systems; **Siemens Five-view Model**: Code; **SEI’s viewpoint and target audience:** N/A; **ISO-RM ODP**: Technical; **Viewpoint Catalog**: Development; **DoDAF 2.0**: Systems; **Zachman**: Technology.)

A deployment architecture defines the execution architecture of systems and the assignment of software artifacts to system elements (UML, 2011). With this definition in mind, the deployment architecture can be classified as:

Deployment – **Phase**: Deployment; **4+1**: Physical; **Abstraction**: PSM. (Others: **NIST EA Model**: Delivery Systems; **Siemens Five-view Model**: Execution; **SEI’s viewpoint and target audience:** Physical; **ISO-RM ODP**: Computation; **Viewpoint Catalog**: Deployment; **DoDAF 2.0**: Product; **Zachman**: Concepts.)

A physical architecture show a system's physical layout, revealing which pieces of software run an what pieces of hardware (Fowler, 2004). With this definition in mind, the physical architecture can be classified as:

Physical – **Phase**: Deployment; **4+1**: Physical; **Abstraction**: PSM. (Others: **NIST EA Model**: Delivery Systems; **Siemens Five-view Model**: Hardware; **SEI’s viewpoint and target audience:** Physical; **ISO-RM ODP**: Technical; **Viewpoint Catalog**: Operational; **DoDAF 2.0**: Systems; **Zachman**: Technology.)

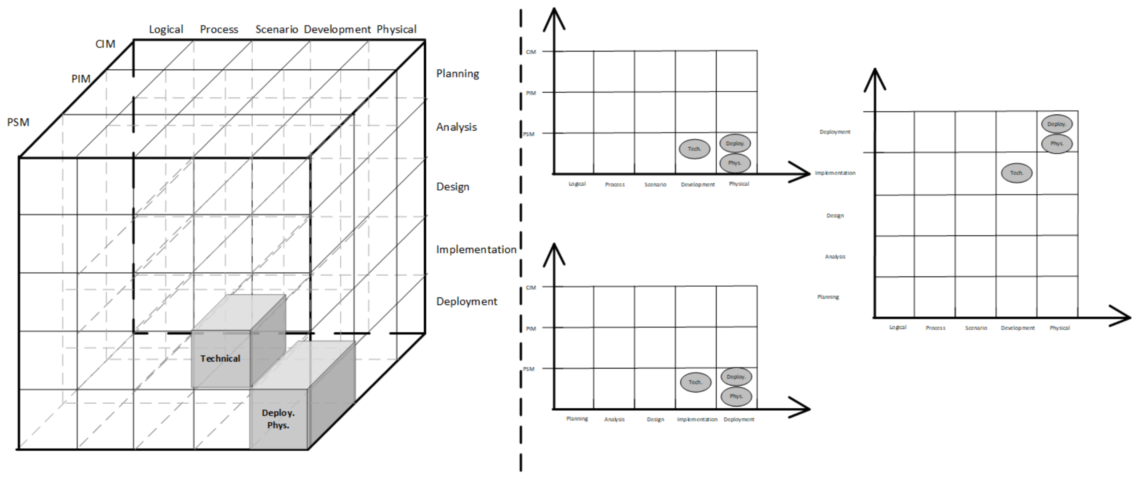


Figure 36. Viewpoints classifications at Level 4

3.3. Modeling approaches within the "continuous" paradigm

Continuous software engineering

While agile practices have succeeded in involving the customer in the development cycle, there is an urgent need to learn from customer usage of software also after delivering and deployment of the software product. Among the plethora of available practices that a company may use towards "being agile", Olsson, Alahyari and Bosch propose a pathway for stages that a company should embrace, called "Stairway to Heaven" (Helena Holmstrom Olsson, Alahyari, & Bosch, 2012; Helena Holmström Olsson & Bosch, 2014). It is composed by five stages, as depicted in Figure 37: (i) Traditional development; (ii) Agile R&D organization; (iii) Continuous integration; (iv) Continuous deployment; and (v) R&D as an experiment system.

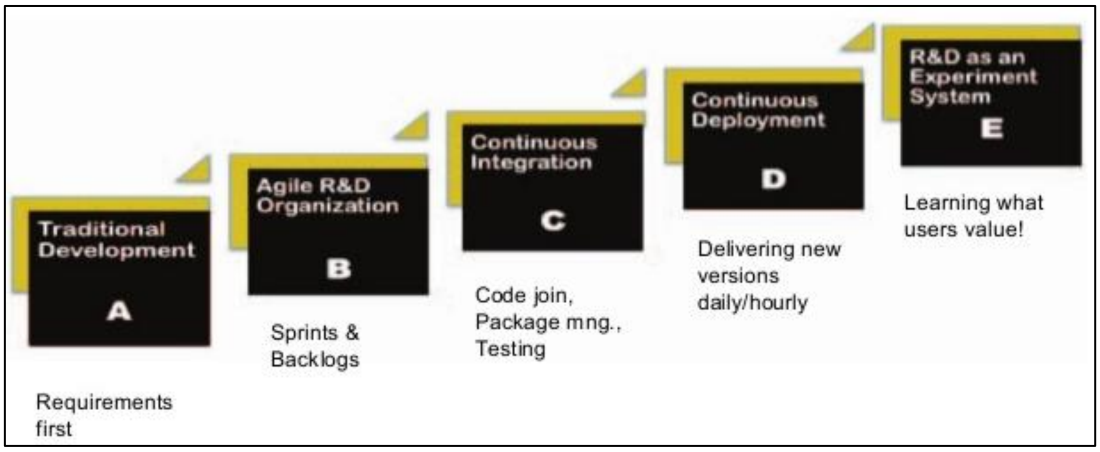


Figure 37. "Stairway to Heaven" ***(Helena Holmstrom Olsson et al., 2012; Helena Holmström Olsson & Bosch, 2014)***

Traditional development

Traditional development is typically the starting point for most companies. Traditional development is a software development approach characterized by slow development cycles, sequential phases (waterfall-style), and a rigorous planning phase in which requirements are frozen upfront (Sommerville, 2007). Projects adopting this development approach suffer from long feedback cycles and difficulties to integrate customer feedback into the product development process (Helena Holmstrom Olsson et al., 2012; Sommerville, 2007). Typically, software delivery takes place in the end of the project life cycle, and it is then that customers can provide feedback.

Agile R&D organization

Adopting agile development practices is typically for overcoming the challenges from overcoming the customer’s long feedback cycles. Agile practices are characterized by small cross-functional development teams, short development Sprints resulting in working software, and continuous planning in which the customer is involved to allow for continuous customer feedback (Highsmith, 2002). In agile organizations, however, product management and system verification still work according to the traditional development approach (Helena Holmstrom Olsson et al., 2012).

Continuous integration (CI)

This step relates to the establishment of practices that allow for frequent integration of work, daily builds, and fast commit of changes (e.g., automated builds and automated testing). At this point, both product development organization and test and verification organization work according to agile practices with short feedback cycles and continuous integration of work. Work is integrated frequently, leading to multiple integrations per day (Humble & Farley, 2011).

Continuous deployment (CD)

CD implies the continuous push out of changes to the code instead of doing large builds and having planned releases of large chunks of functionality. This allows for continuous customer feedback, the ability to learn from customer usage data, and to eliminate work that does not produce value for the customer. At this point, R&D, product management, and customers are all involved in a rapid, agile development cycle in which response time is short (Helena Holmstrom Olsson et al., 2012).

R&D as an experiment system

The final step in the “Stairway to Heaven” model relates to the ability of the organization to respond based on instant customer feedback, where actual deployment of software functionality is seen as a way of validating functionality. Customers are exposed to partial implementation of a functionality and the organization uses their feedback for determining the value of that particular functionality (Bosch, 2012).

Olsson and Bosch state the following when organizations evolve from one step to another (Helena Holmström Olsson & Bosch, 2014):

- From Traditional to Agile R&D: requires a careful introduction of agile practices into the organization, a shift to small development teams, and a focus on features rather than components
- From Agile to Continuous Integration: requires an automated test suite, a main branch to which code is continually delivered, and a modularized architecture
- From Continuous Integration to Continuous Deployment: requires internal and external stakeholders to be fully involved and a proactive customer with whom to explore the concept
- From Continuous Deployment to R&D as an "Innovation System": requires careful ecosystem management in order to align internal business strategies with the dynamics of a competitive business ecosystem.

Finally, a characteristic for all transitions is the critical alignment of internal and external processes in order to maximize the benefits as provided by the business ecosystem of which a company is part.

A research conducted by Fitzgerald and Stol showed that software engineering has evolved with the adoption of agile practices and CI, Lean Startup and Lean Thinking approaches (Fitzgerald & Stol, 2017). They also discuss the need for a holistic approach that also refers the adoption of agile approaches in other organizational functions, like Enterprise Agility (Overby, Bharadwaj, & Sambamurthy, 2005) and Beyond Budgeting (Bogsnes, 2008). Finally, the approach is complemented by integrating development and operations (DevOps) (Loukides, 2012), and integrating business strategy and development (BizDev) (Fitzgerald & Stol, 2017). Hence, they propose a holistic approach encompassing all these concepts and emphasizing in CSE, called "Continuous ∗" (i.e. Continuous Star) (Fitzgerald & Stol, 2017). Continuous ∗ considers the entire software life cycle, with three main sub- phases (Figure 38): Business Strategy & Planning, Development, and Operations.

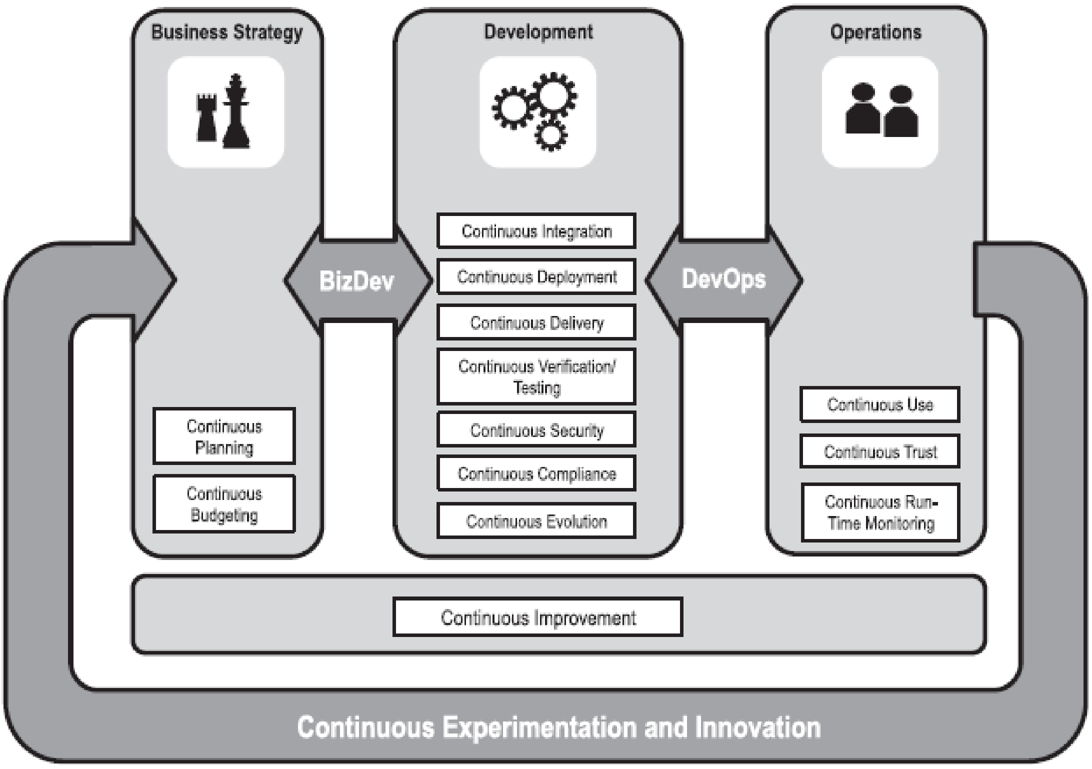


Figure 38. Continuous* framework (Fitzgerald & Stol, 2017)

Each activity from Continuous * is briefly explained:

- **Continuous planning:** endeavor involving multiple stakeholders from business and software functions, where plans are dynamic open-ended artifacts that evolve in response to changes in the business environment;
- **Continuous budgeting:** budgeting (organization’s investments, revenue and expense outlook) becomes a continuous activity to facilitate changes during the year;
- **Continuous integration:** a typically automatically triggered process comprising inter-connected steps such as compiling code, running unit and acceptance tests, validating code coverage, checking coding standard compliance and building deployment packages;
- **Continuous delivery:** the practice of continuously deploying good software builds automatically to some environment, but not necessarily to actual users;
- **Continuous deployment:** the practice of ensuring that the software is continuously ready for release and deployed to actual customers;

- **Continuous verification:** Adoption of verification activities including formal methods and inspections throughout the development process rather than relying on a testing phase towards the end of development;
- **Continuous testing:** a process typically involving some automation of the testing process, or prioritization of test cases, to help reduce the time between the introduction of errors and their detection;
- **Continuous compliance:** seeks to satisfy regulatory compliance standards on a continuous basis;
- **Continuous security:** security being treated as a key concern throughout all phases of the development lifecycle and even post deployment, supported by a smart and lightweight approach to identifying security vulnerabilities;
- **Continuous evolution:** technical debt is incurred when an architecture is unsuitable to facilitate new requirements;
- **Continuous use:** Recognizes that the initial adoption versus continuous use of software decisions are based on different parameters, and that customer retention can be a more effective strategy than trying to attract new customers;
- **Continuous trust:** Trust developed over time as a result of interactions based on the belief that a vendor will act cooperatively to fulfill customer expectations without exploiting their vulnerabilities;
- **Continuous run-time monitoring:** run-time behaviors of all kinds must be monitored to enable early detection of quality-of-service problems, such as performance degradation, and also the fulfillment of service level agreements (SLAs);
- **Continuous improvement:** Based on lean principles of data-driven decision-making and elimination of waste, which lead to small incremental quality improvements;
- **Continuous innovation:** a sustainable process that is responsive to evolving market conditions and based on appropriate metrics across the entire lifecycle of planning, development and run-time operations;
- **Continuous experimentation:** software development based on experiments with stakeholders consisting of repeated Build-Measure-Learn cycles.

Continuous Architecture

Continuous Architecture (CA) is a set of principles and tools targeted at addressing the gap between the Agile delivery and architecture practices (Erder & Pureur, 2015). It brings together the work of the agile developer to start building and the enterprise architect that will look at a 5-year plan. CA is the response required from the adoption of Continuous Delivery by software development teams.

The main objective of Continuous Delivery is to respond quickly to business needs by frequently delivering high-quality software in rapid cycles. Unlike traditional software delivery approaches that emphasize the importance of delivering various documents such as requirements, architecture, and design specifications, the overall goal of Continuous Delivery is to produce production-quality software rapidly in an incremental manner. Instead of validating various artifacts produced as part of the Software Development Life Cycle (SDLC), quality is enforced by systematically testing the software components using automated tests (Erder & Pureur, 2015).

Continuous architecting has a set of specific goals (Erder & Pureur, 2015):

- To create an architecture that can evolve with applications, that is testable , that can respond to feedback and in fact is driven by feedback
- To make Enterprise Architecture real
- To make solution architecture sustainable
- To create real world, actionable, useful strategies

CA is characterized by the following principles (Erder & Pureur, 2015):

1. Architect Products – not solutions for Projects
2. Focus on Quality Attributes – not on Functional Requirements
3. Delay Design Decisions Until They Are Absolutely Necessary To Keep The Architecture Manageable
4. Leverage "The Power Of Small" To Architect For Change
5. Architect for Build, Test and Deploy To Deliver Capabilities Continuously
6. Model The Organization Of Your Teams After The Design Of The System To Promote Interoperability

As part of CA, especially within large-scale agile (LSA) settings, (Martini, Pareto, & Bosch, 2014) identifies the following practices:

- Risk Management;
- architectural decisions and changes;
- Pattern Distillation;
- communication of architecture:
    - providing architectural knowledge (communication output),
    - monitor the current status of the system (communication input);
- Inter-features Architecting;
- Architecting for Testability;
- Controlling Erosion.

The CA practices require focus in the architecture role, as well as in architecting tasks within developing and maintaining software architecture. Martini and Bosch present Continuous Architecting Framework For Embedded software and Agile (CAFFEA) (Martini, Pareto, & Bosch, 2015), an organizational framework oriented for architecture governance. The frameworks presents roles, challenges and tasks within architecture teams and governance teams viewpoints (Figure 39). Roles are described as belonging to the Architects and to the Team. The existing roles for Architects are Chief Architect, Governance Architect and Team Architect. Chief architects are responsible for the whole overall portfolio architecture, which might include more products and more than one system; Governance architects are responsible for areas of the architecture, related to single products or systems or sub-systems, but not related to only one team; and Team architects are usually most experienced developer in a team who have the most knowledge about the architecture and support/lead the team on such area (Martini et al., 2014). Within the Teams, within this framework they have roles responsible for coordination and cooperation practices, which are complementary to the typical agile (and feature-oriented) teams. Roles for Teams are Runway Team, Architecture Teams, Governances Teams.

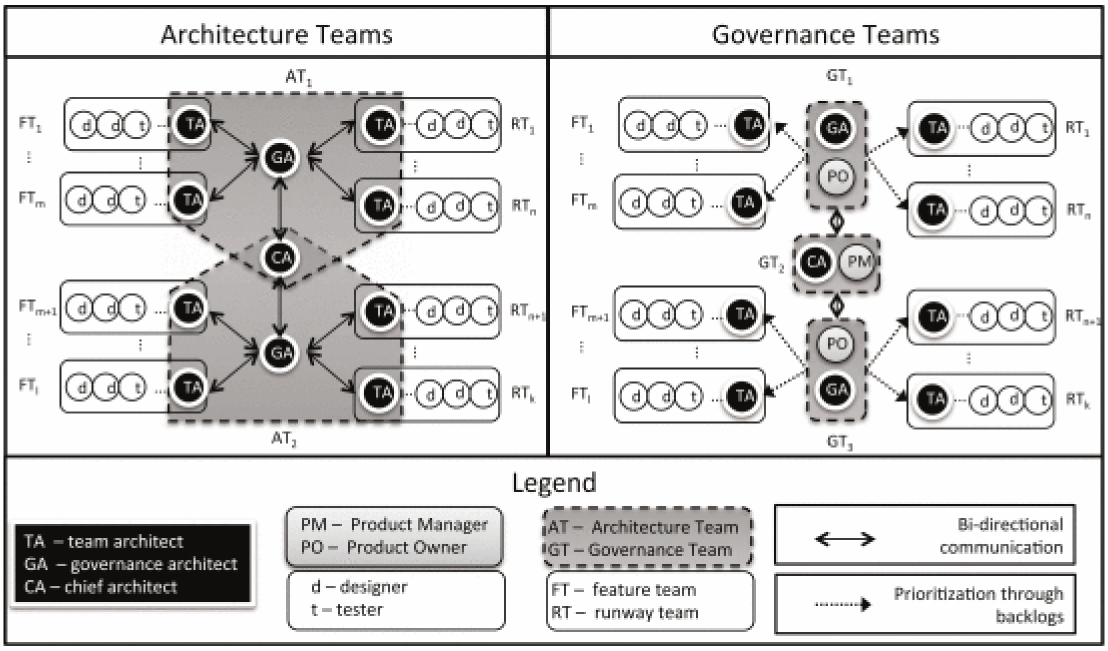


Figure 39. The Continuous Architecting Framework For Embedded software and Agile (CAFFEA) *(Martini et al., 2015)*

Architectural Management, Evolution, Change, and Debt

The architecture is not static either during the project, after either the project ended (and the software is delivered and "in production"). Architectural changes as the architecture evolves, because of enhancement and maintenance requirements, are addressed as architecture management (Babar, 2013). Changes come in different flavours, such as redefining or adding requirements, changing infrastructure and technology, or causing changes by bugs and wrong decisions. to avoid design erosion, software architects need to embrace change by systematically alternating design activities with iterative architecture assessment and refactoring (Stal, 2014).

Architecture maintenance is performed by architectural methods that support evaluation of developed features during cycles (Kanwal, Junaid, & Fahiem, 2010). At this point, the major concern is to accommodate the required changes without damaging the architectural integrity. Prior design decisions are reassessed for the potential impact of the required changes and new decisions are made (Babar, 2013).

Any change that will influence the system's safety requirements after we have finished the safety analysis and safety planning for development of safety-critical software will require a change impact analysis. Using agile development we may add new requirements, change existing requirements and make current requirements more detailed both in the product backlog and in the Sprint backlogs (Stålhane, 2014).

Change impact analysis (CIA) is important for software maintenance and is closely related to traceability in two ways (Stålhane, 2014):

(1) *From code to requirements* – which requirements are affected if we change this code? This also gives us information on which tests that need to be re-run;

(2) *From requirements to code* – what code must be changed if this requirement is changed?

Steps in a typical impact analysis process are (Wiegers, 2014):

- Identify the sequence in which the tasks must be performed and how they can be interleaved with currently planned tasks.
- Determine whether the change is on the project's critical path. If a task on the critical path slips, the project's completion date will slip. Every change consumes resources, but if you can plan a change to avoid affecting tasks that are currently on the critical path, the change will not cause the entire project to slip.
- Estimate the impact of the proposed change on the project's schedule and cost.
- Evaluate the change's priority by estimating the relative benefit, penalty, cost, and technical risk compared to other discretionary requirements.
- Report the impact analysis results to all stakeholders so that they can use the information to help them decide whether to approve or reject the change request.

A change in a system may have effects that have to be determined, which leads to use CIA techniques (Arnold, 1996). In agile architecting, CIA is used in decision-making process of adding or changing features, focusing in affected dependencies with earlier design decisions, rationale, constraints, and risks (Pérez, Díaz, Garbajosa, & Yagüe, 2014).

It is crucial to perform a CIA if the change affects an architecturally significant requirement (ASR) (L. Chen, Ali Babar, & Nuseibeh, 2013). ASRs are requirements that play an important role in determining the architecture of the system (Paul Clements & Bass, 2010). This concept mostly arose from the need to differentiate the quality requirements (i.e., non-functional requirements

(Sommerville, 2007), which are commonly related to the architecture of the system) that do not have great impact to the system design as the ones that actually do have importance in design decisions (L. Chen et al., 2013).

These techniques may arise as a way of managing the technical debt (Kruchten, Nord, & Ozkaya, 2012; Tom, Aurum, & Vidgen, 2013). Technical debt may relate from architecture, source code, testing, among others (Martini, Besker, & Bosch, 2018). Architectural debt relates to a group of architecturally connected files that incur high maintenance costs over time due to their flawed connections (L. Xiao, Cai, Kazman, Mo, & Feng, 2016).

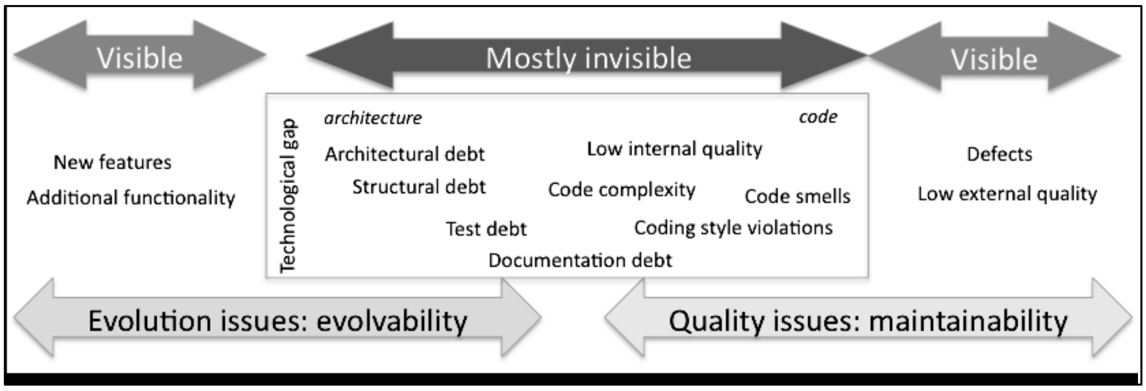


Figure 40. Types of Technical debt (Kruchten, Nord, & Ozkaya, 2012)

Software architecture denotes a potential area for refactoring activities due to its continuous growth and evolution. Software architecture assessment and refactoring should happen regularly, in all iterations (Stal, 2014).

Refactoring activities should be conducted iteratively in a systematic way, towards a process for continuous architecture improvement, which can include (Stal, 2014):

- Architecture assessment: Identify architecture smells and design problems;
- Prioritization: Prioritize all identified architectural issues by determining the priority of the affected requirements;
- Select appropriate refactoring patterns;
- Quality assurance: For each refactoring application, check whether it changes the semantics of the system.

3.4. Microservices architectures

Microservices are an architectural style oriented towards modularization, where the idea is to split the application into smaller, interconnected services, running as a separate process (Figure 41) that can be independently deployed, scaled and tested (Thönes, 2015).

They also extend from the 'design-stage architecture' into deployment and operations as a continuous development style (Pahl & Jamshidi, 2016). That is why, when working with microservices, considerations range from architecture to deployment (or DevOps) issues (Aderaldo, Mendonca, Pahl, & Jamshidi, 2017).

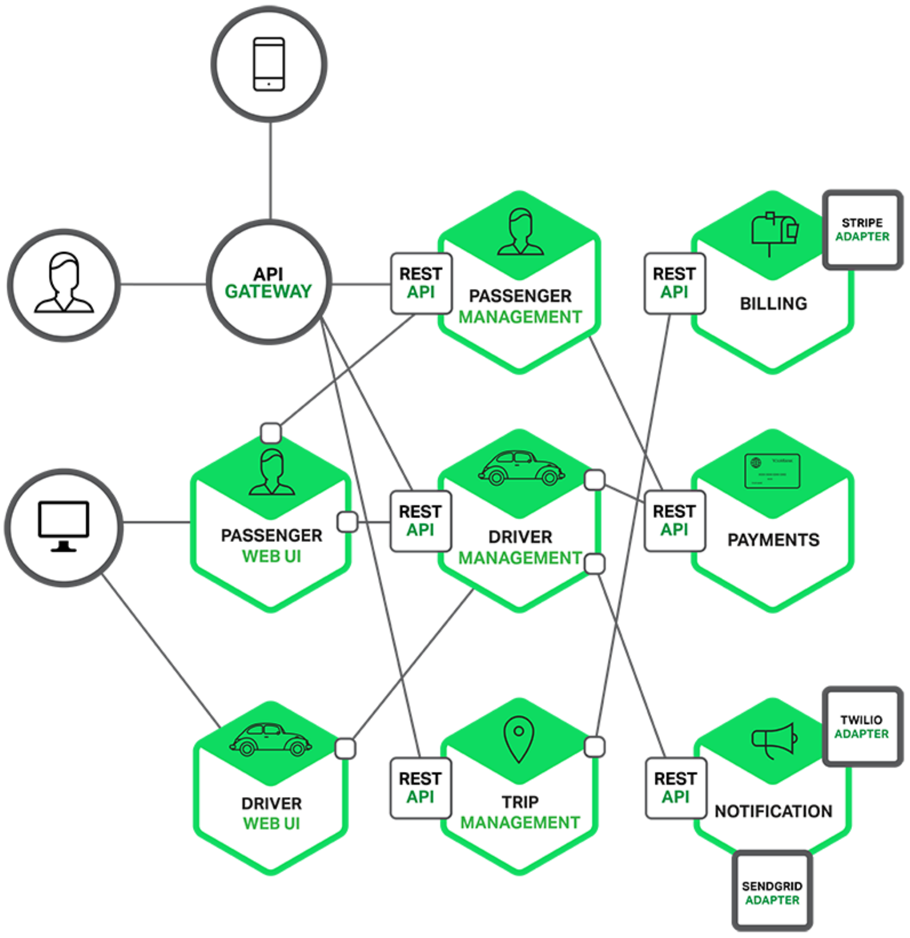


Figure 41. A pictorial representation of a microservices architecture[4]

The main difference when dealing with a monolithic application and a microservices architecture is that a monolithic application puts all its functionality into a single process and scales by replicating the monolith on multiple servers, while a microservices architectures puts

[4] https://www.nginx.com/blog/introduction-to-microservices/

each element of functionality into a separate service and scales by distributing these services across servers, replicating as needed (J. Lewis & Fowler, 2014).

A microservices system encompasses all of the things about your organization that are related to the application it produces. For that reason, Nadareishvili, Mitra, McLarty, and Amundsen present a microservice design model comprised of five parts (Figure 42) (Nadareishvili, Mitra, McLarty, & Amundsen, 2016): Service, Solution, Process and Tools, Organization, and Culture.

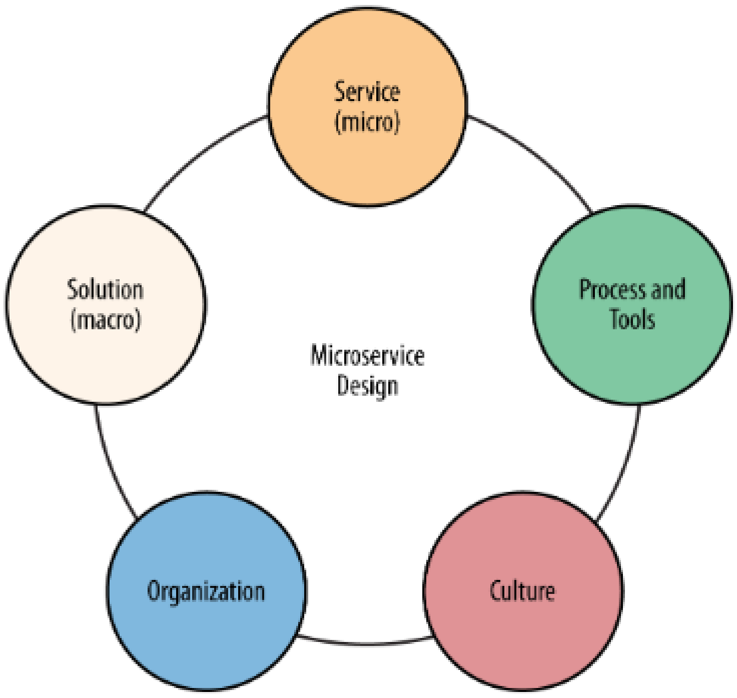


Figure 42. Microservices main characteristics

The main characteristics of microservices are (J. Lewis & Fowler, 2014):

- Componentization via Services: Software is broken up into multiple services that are independently replaceable and upgradeable and communicate by means of inter-process communication facilities using an explicit component-published-interface.
- Organized Around Business Capabilities: Microservices are implemented around business areas, in which services include a user-interface, storage, and any external collaborations.
- Products not Projects: Development teams own a product throughout its entire lifetime, taking full responsibility for the software in production.
- Smart Endpoints and Dumb Pipes: Simple messaging or a lightweight messaging bus is used to provide communication among microservices.

- Decentralized Governance: teams use the right tool (or programming language) for a given situation, instead of a single tool for the entire solution. It is also referred as polyglot programming.
- Decentralized Data Management: Microservices prefer letting each service manage its own database, either different instances of the same database technology, or entirely different database systems. It is also referred as polyglot persistence.
  - Infrastructure Automation: Many of the products or systems being built with microservices use infrastructure automation techniques, based in automated testing and deployment, towards continuous delivery.
  - Design for Failure: applications need to be designed so that they can tolerate the failure of services.
  - Evolutionary design.

The development of microservices follow the following principles (Newman, 2015):

1. "Model around business concepts", to be represented as bounded contexts and domain models according to Domain-Driven Design (DDD) patterns (Evans, 2004).
2. "Adopt a culture of automation" in testing and deployment; practice continuous delivery.
3. "Hide internal implementation details" such as databases; define technology-agnostic Application Programming Interfaces (APIs).
4. "Decentralize all the things": e.g., apply shared governance, prefer service choreography over orchestration, and use dumb middleware but smart endpoints.
5. Make services "independently deployable", e.g., let versioned (service) endpoints co-exist; deploy only one service per (virtual) host.
6. "Isolate failure", e.g. introduce circuit breakers to make services robust.
7. Be "highly observable", e.g. via semantic monitoring with data aggregation.

Although these characteristics and principles overlap, they also differ substantially (Table 9) (Zimmermann, 2017):

Table 9. Microservices characteristics and principles (Zimmermann, 2017)

Characteristics by Lewis/Fowler (J. Lewis & Fowler, 2014)	Relationship	Newman’s Principles (Newman, 2015)
Componentization via services	(similar to)	Hide internal implementation details
Organized around business capabilities	(matches)	Model around business concepts
Products not projects	(no pendant)	
Smart endpoints and dumb pipes	(included in)	Decentralize all the things
Decentralized governance	(superset of)	
Decentralized data management	(superset of)	
Infrastructure automation	(superset of)	Adopt a culture of automation
Design for failure	(subset of)	Isolate failure
Evolutionary design	(no pendant)	
	(no pendant)	Highly observable
Independently deployable (not formally listed as a characteristic, but described as a definition	(matches)	Independently deployable

Also, in the same work, Zimmerman summarizes common tenets on microservices (Zimmermann, 2017):

1. *Fine-grained interfaces* to single-responsibility units that encapsulate data and processing logic are exposed remotely, typically via RESTful HTTP resources or asynchronous message queues. These remote units constitute services that can be deployed, changed, substituted, and scaled independently of each other.
2. Business-driven development practices and pattern languages such as *domain-driven design* (DDD) (Evans, 2004) are employed to identify and conceptualize services.
3. Cloud-native application design principles are followed, e.g., as summarized in *IDEAL* (isolated state, distribution, elasticity, automated management and loose coupling)

(Haberle, Charissis, Fehling, Nahm, & Leymann, 2015) or the twelve app factors in Heroku's method (Wiggins, 2012).

4. Multiple computing paradigms (such as functional and imperative) and storage paradigms (e.g., relational databases and several types of NoSQL stores) are leveraged in a *polyglot programming and persistence* strategy. Some of these polyglot services only guarantee eventual rather than strong consistency.
5. *Lightweight containers* are used to deploy services. Docker and Dropwizard are frequently mentioned as two related options (although these two technologies do not reside on the same level of abstraction and have rather different scopes, operating system virtualization vs. code library assembly).
6. *Decentralized continuous delivery* is practiced during service development (which requires/promotes a high degree of automation and autonomy).
7. *DevOps* Lean, but holistic and largely automated approaches to configuration, performance and fault management are employed, which extend agile practices and include service monitoring.

Finally, Zimmerman summarizes the analysis by positioning the seven tenets (T-x), the nine characteristics from Lewis and Fowler (LFy), and Newman's seven principles (N-z) in Kruchten's 4+ 1 viewpoint scheme (Kruchten, 1995). Figure 43 shows consensus and/or complementary positions in three viewpoints (scenario, development, and process) and little focus on the remaining two (logical, physical); one tenet, five L/F characteristics and two N principles deal with cross-cutting concerns that span multiple viewpoints (e.g., decentralized governance).

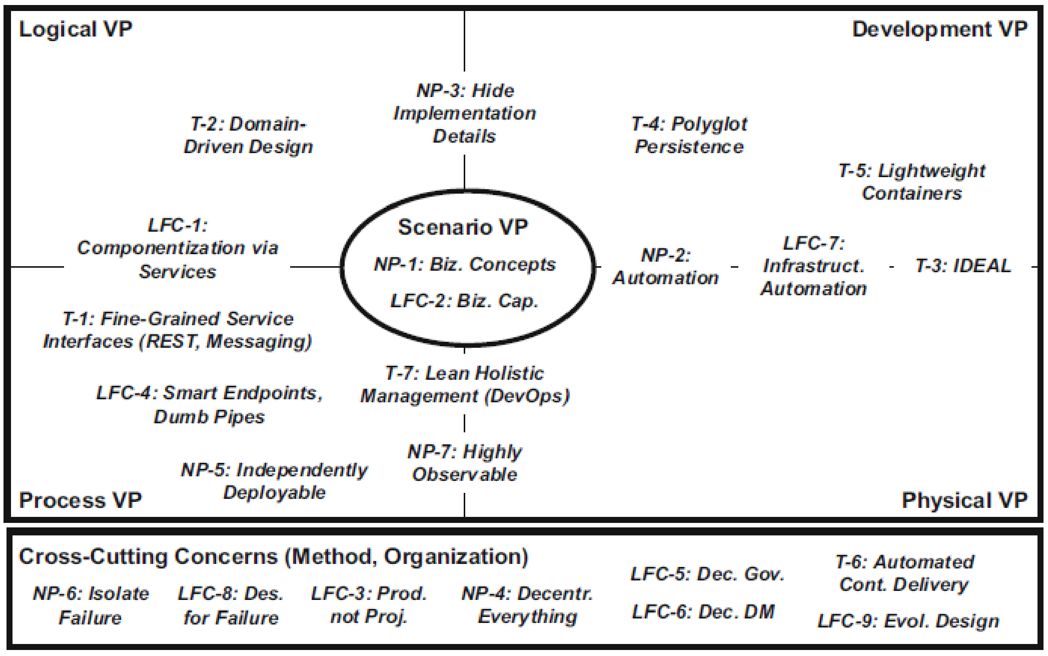

Logical VP
Development VP
Process VP
Physical VP
Scenario VP
NP-1: Biz. Concepts
LFC-2: Biz. Cap.
NP-3: Hide Implementation Details
T-2: Domain-Driven Design
T-4: Polyglot Persistence
T-5: Lightweight Containers
LFC-1: Componentization via Services
NP-2: Automation
LFC-7: Infrastruct. Automation
T-3: IDEAL
T-1: Fine-Grained Service Interfaces (REST, Messaging)
LFC-4: Smart Endpoints, Dumb Pipes
T-7: Lean Holistic Management (DevOps)
NP-7: Highly Observable
NP-5: Independently Deployable
Cross-Cutting Concerns (Method, Organization)
NP-6: Isolate Failure
LFC-8: Des. for Failure
LFC-3: Prod. not Proj.
NP-4: Decentr. Everything
LFC-5: Dec. Gov.
LFC-6: Dec. DM
T-6: Automated Cont. Delivery
LFC-9: Evol. Design

Figure 43. Positioning the microservices tenets (Zimmermann, 2017)

They enable companies to increase the deployment frequency of new releases as one crucial part within the Continuous Delivery (CD) pipeline (Armin Balalaie, Heydarnoori, & Jamshidi, 2016a; L. Chen, 2018; O’Connor, Elger, & Clarke, 2017). They affect the way teams are structured, source code is organized and continuously built/packed, and software products are continuously deployed (Familiar, 2015).

There is some discussion about similarities and differences to SOA. Microservices are not entirely new, but qualify as “SOA done right”, comprising an organic implementation approach to SOA (Zimmermann, 2017). Common characteristics include business orientation, polyglot programming in multiple paradigms and languages, and design for failure; decentralization and automation are emphasized specifically in the microservices implementation approach. An important microservices property is that services can be deployed independently of each other, which requires services to communicate with each other via remoting protocols such as HTTP and asynchronous message queues (Zimmermann, 2017).

Typically there are two types of patterns to define the required microservices (Richardson, 2018): decomposition by business capability or by domains. The second one is highly adopted (Newman, 2015; Pautasso, Zimmermann, Amundsen, Lewis, & Josuttis, 2017; Steinegger, Giessler, Hippchen, & Abeck, 2017), making use of DDD (Evans, 2004) approach.

Independent of whether an MSA-based system is designed greenfield, *i.e.*, from scratch, or brownfield, *i.e.*, by decomposing a monolith, microservices need to be identified (Newman, 2015). There is not a common standard on modeling the architectures (Francesco, Malavolta, & Lago, 2017). In fact, 'modeling' is not highly considered in microservice-related research (Cerny, Donahoo, & Trnka, 2018), although its provided benefits in terms of abstraction, model transformation, code generation, modeling viewpoints and languages (Rademacher, Sorgalla, Wizenty, Sachweh, & Zündorf, 2018). There seems to be a tendency to use languages used to describe service-based architectures (Di Francesco, 2017) (SoaML, SOMA, SOADL, CAML, CloudML and StratusML), however UML is suitable to model services and operations as well (Alshuqayran, Ali, & Evans, 2016; Rademacher, Sachweh, & Zündorf, 2018a). Modeling approaches in SOA is more mature, namely for service design and interfaces, and their applicability in MSA have many similarities (Rademacher, Sachweh, & Zundorf, 2017). Modeling in MSA may be based in integrating API and SOA styles of service design and delivery, together with a service model that contains both styles of service and articulates their relationships (Z. Xiao, Wijegunaratne, & Qiang, 2016).

Models may be used in different abstraction levels (OMG, 2003). In microservices, domain models are used when adopting DDD for identifying the services, where afterwards may be used additional models – intermediate and deployment – for specifying service interfaces, deployment, etc. (Rademacher, Sorgalla, & Sachweh, 2018). Additionally, these different models may be used within different languages, like UML for domains and SoaML for interfaces (Rademacher, Sorgalla, & Sachweh, 2018).

Microservices modeling

DDD is always the basis for defining a microservices architecture, allowing to decompose a problem in subdomains that a microservice may tackle, and also assuring the microservice complies with the Single Responsibility Principle (SRP) (Indrasiri & Siriwardena, 2018). Rademacher, Sachweh and Zündorf present a UML Profile for identifying microservices, namely when adopting DDD (Rademacher, Sachweh, & Zündorf, 2018b).

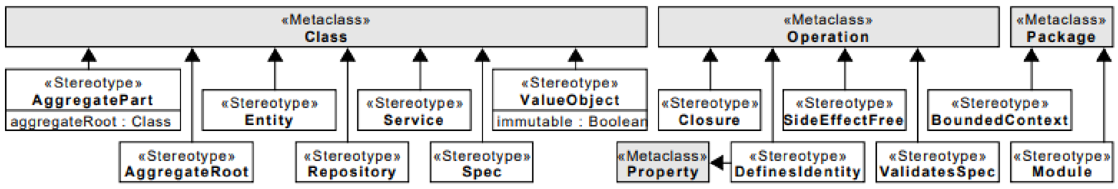


Figure 44. UML profile for microservices design *(Rademacher, Sachweh, & Zündorf, 2018b)*

Kharbuja uses DDD for identifying bounded contexts within requirements modeled in Use Cases, defining steps to derive a domain model for a microservice, namely (Kharbuja, 2016):

- **Step 1:** The initial analysis of the case study produces use case model
- **Step 2:** For each use case, task trees are generated listing the functionalities needed to accomplish the desired goal of the respective use cases.
- **Step 3:** The initial task trees created for each use case at Step 2 are analyzed. The tasks are categorized as are either functionally independent from their corresponding use cases or common in multiple use cases.
- **Step 4:** The use case model obtained in Step 3 is analyzed again for further refactoring.
- **Step 5:** The use cases obtained in step 4 are used to identify the service candidates. The final use cases obtained in Step 4 have appropriate level of granularity and cohesive functionalities.

An example of a UML Use Case model usage for candidate service is depicted in Figure 45.

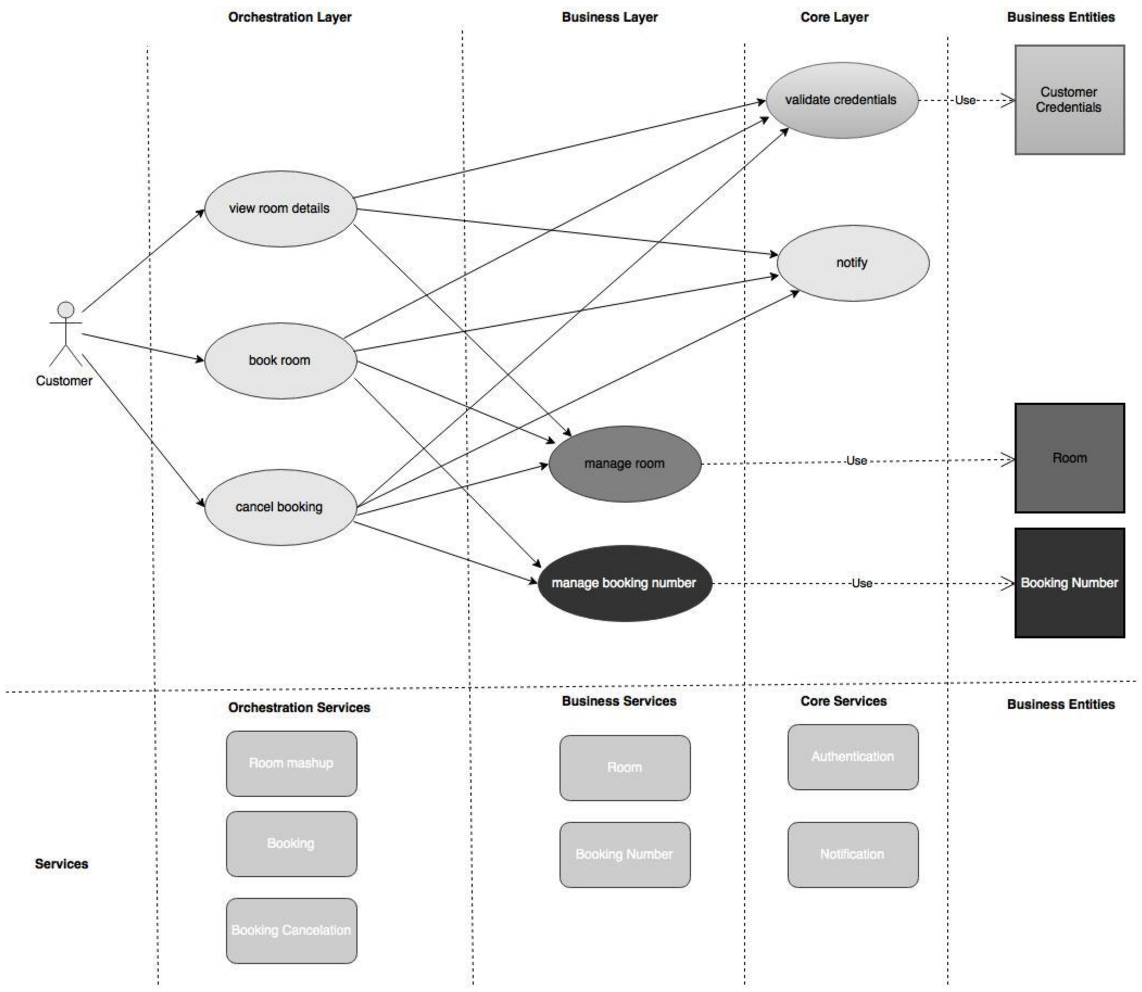
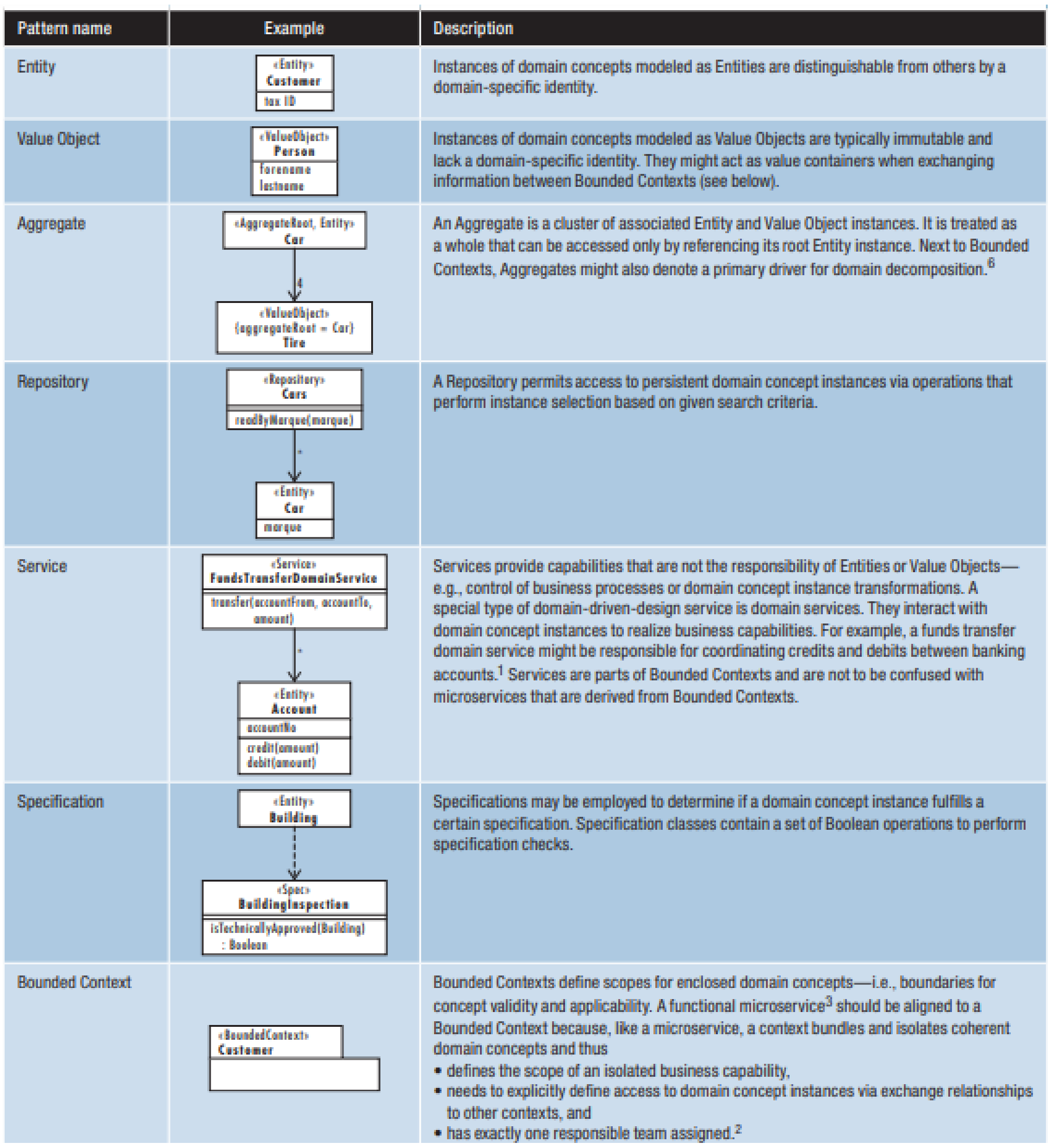


Figure 45. Use Case Model for identification of service candidates (Kharbuja, 2016)

Rademacher *et al.* present some patterns on modeling microservices domains using DDD (Rademacher, Sorgalla, & Sachweh, 2018). Figure 46 depicts DDD modeling patterns for microservice design, using UML notation.



Pattern name	Example	Description
Entity	«Entity» Customer / tax ID	Instances of domain concepts modeled as Entities are distinguishable from others by a domain-specific identity.
Value Object	«ValueObject» Person / forename lastname	Instances of domain concepts modeled as Value Objects are typically immutable and lack a domain-specific identity. They might act as value containers when exchanging information between Bounded Contexts (see below).
Aggregate	«AggregateRoot, Entity» Car → 4 → «ValueObject» {aggregateRoot = Car} Tire	An Aggregate is a cluster of associated Entity and Value Object instances. It is treated as a whole that can be accessed only by referencing its root Entity instance. Next to Bounded Contexts, Aggregates might also denote a primary driver for domain decomposition.[6]
Repository	«Repository» Cars / readByMarque(marque) → * → «Entity» Car / marque	A Repository permits access to persistent domain concept instances via operations that perform instance selection based on given search criteria.
Service	«Service» FundsTransferDomainService / transfer(accountFrom, accountTo, amount) → * → «Entity» Account / accountNo / credit(amount) debit(amount)	Services provide capabilities that are not the responsibility of Entities or Value Objects—e.g., control of business processes or domain concept instance transformations. A special type of domain-driven-design service is domain services. They interact with domain concept instances to realize business capabilities. For example, a funds transfer domain service might be responsible for coordinating credits and debits between banking accounts.[1] Services are parts of Bounded Contexts and are not to be confused with microservices that are derived from Bounded Contexts.
Specification	«Entity» Building ⇢ «Spec» BuildingInspection / isTechnicallyApproved(Building) : Boolean	Specifications may be employed to determine if a domain concept instance fulfills a certain specification. Specification classes contain a set of Boolean operations to perform specification checks.
Bounded Context	«BoundedContext» Customer	Bounded Contexts define scopes for enclosed domain concepts—i.e., boundaries for concept validity and applicability. A functional microservice[3] should be aligned to a Bounded Context because, like a microservice, a context bundles and isolates coherent domain concepts and thus • defines the scope of an isolated business capability, • needs to explicitly define access to domain concept instances via exchange relationships to other contexts, and • has exactly one responsible team assigned.[2]

Figure 46. DDD patterns for domain-driven microservice design *(Rademacher, Sorgalla, & Sachweh, 2018)*

The Extended Increment Architecture approach (Zúñiga-Prieto, Insfran, & Abrahao, 2016) lengthens the SoaML metamodel in order to handle microservices architecture design. Within this approach, a Participant may refer to:

(i) a microservice to be integrated;

(ii) a microservice/component already existing in the current architecture with which the microservice(s) to be integrated will interoperate; and

(iii) a microservice/component to be created in order to consume microservice services or provide it with services. In addition, the Services Architecture diagram allows depicting how parts of a microservice work together to play the owning microservice role(s). Finally, a reference to a Service Contract that describes interoperation among Participants is described, as well as the integration logic.

In order to allow these languages properly address microservice specific characteristics (mainly in comparison to SOA), some model-driven development (MDD) works propose metamodels that enable microservices architectures modeling (Düllmann & Van Hoorn, 2017; Rademacher, Sorgalla, Sachweh, & Zündorf, 2018), allowing to instantiate data, service and operations of microservices. The DDD application can be enriched by defining semantics in OWL (Diepenbrock, Rademacher, & Sachweh, 2017).

An agile approach for service design and service engineering relies on early understanding of user needs and service touchpoints for rapid adaptation to emerging user needs (Berre, 2012). Model-based development approaches, properly combined with agile practices, are useful for service design and engineering, relating value models, process models, user interface and interaction flow models, and service architectures and service contract models (Berre, 2012).

Defining service boundaries

Decomposing an existing monolith to microservices has many challenges (Di Francesco, 2017; D Taibi, Lenarduzzi, Pahl, & Janes, 2017). Typically, the decomposition starts by developing services for a given business process (Lenarduzzi & Taibi, 2018), making use of simplified microservices patterns (Davide Taibi et al., 2017; Davide Taibi, Lenarduzzi, & Pahl, 2018). These patterns are used until the architectures emerges to a complexity that requires new decisions DDD approach, on data driven as the database-is-the-service pattern (Messina, Rizzo, Storniolo, Tripiciano, & Urso, 2016), on approaches as SMART (G. Lewis, Morris, Simanta, Smith, & Wrage, 2007) or ENTICE (Kecskemeti & Marosi, 2016), etc.

Some works have researched on how to extract these services from monoliths (Gysel, Kölbener, Giersche, & Zimmermann, 2016; Mazlami, Cito, & Leitner, 2017; Quiroz, Kim, Parashar, Gnanasambandam, & Sharma, 2009). Decomposing into microservices have impact on the source code, but concerns like multi-tenancy, statefulness and data consistency must be taken in consideration (Furda, Fidge, Zimmermann, Kelly, & Barros, 2018), while a new infrastructure may be developed (Armin Balalaie et al., 2016a).

These works have followed the microservices architectures as they have evolved in complexity, starting by deploying the individual services in lightweight container technologies, then introducing discovery services and reusable fault-tolerant communication libraries, service proxies, or sidecars, and ultimately serverless architectures (Jamshidi, Pahl, Mendonca, Lewis, & Tilkov, 2018), usually guided by the reference proposal in (Yale Yu, Silveira, & Sundaram, 2016).

Microservices patterns

Although recent, developing microservices has had such good acceptance that some patterns have been already identified. Taibi, Lenarduzzi and Pahl described architectural patterns categorized by Orchestration and Coordination, Deployment, Data (Davide Taibi et al., 2018). Issues such as data consistency, security, communication, deployment, and other patterns (Krause, 2014; Namiot & Sneps-Sneppe, 2014; Richardson, 2018; Davide Taibi et al., 2018) have also been addressed.

The patterns from these works often overlap, so for simplicity reasons, further it is presented a set of widely accepted microservices patterns, proposed by Richardson (Richardson, 2018) (Figure 47). The patterns are classified in: (1) Application patterns; (2) Application Infrastructure patterns; and (3) Infrastructure patterns. Additionally, the patterns are divided, following a division structure as listed in Table 10. This pattern catalogue prescribe a set of development approaches for MSA projects. Inside each category, patterns may be exclusive, complimentary or dependent between them. It is thus possible to depict how a MSA project development process may be organized.

Any migration of an application’s architecture to microservices brings challenges that make this migration a non-trivial task. Balalaie, Herdarnoori and Jamshidi proposed migration steps, after analyzing the architecture before the migration and the target architecture (A Balalaie, Heydarnoori, Jamshidi, Tamburri, & Lynn, 2015). Migrating the system towards the target architecture should be done incrementally and in several steps without affecting the end-users of the system. Furthermore, as the number of services is growing, there is a need of a mechanism for automating the delivery process.

By describing an experience report of their migration process, a set of migration steps may be generalized as follows (Armin Balalaie, Heydarnoori, & Jamshidi, 2016b):

- Preparing the Continuous Integration Pipeline;
- Expose legacy functionalities as a REST API;
- Introducing Continuous Delivery practices (e.g., separate the source code, the configuration, and the environment specification);
- Introducing Edge Server (minimize the impact of internal changes on end-users);
- Introducing Dynamic Service Collaboration (Service Discovery, Load Balancer and Circuit Breaker);
- Introducing the Resource Manager;
- Introducing additional services to complete the target architecture;
- Clusterization.

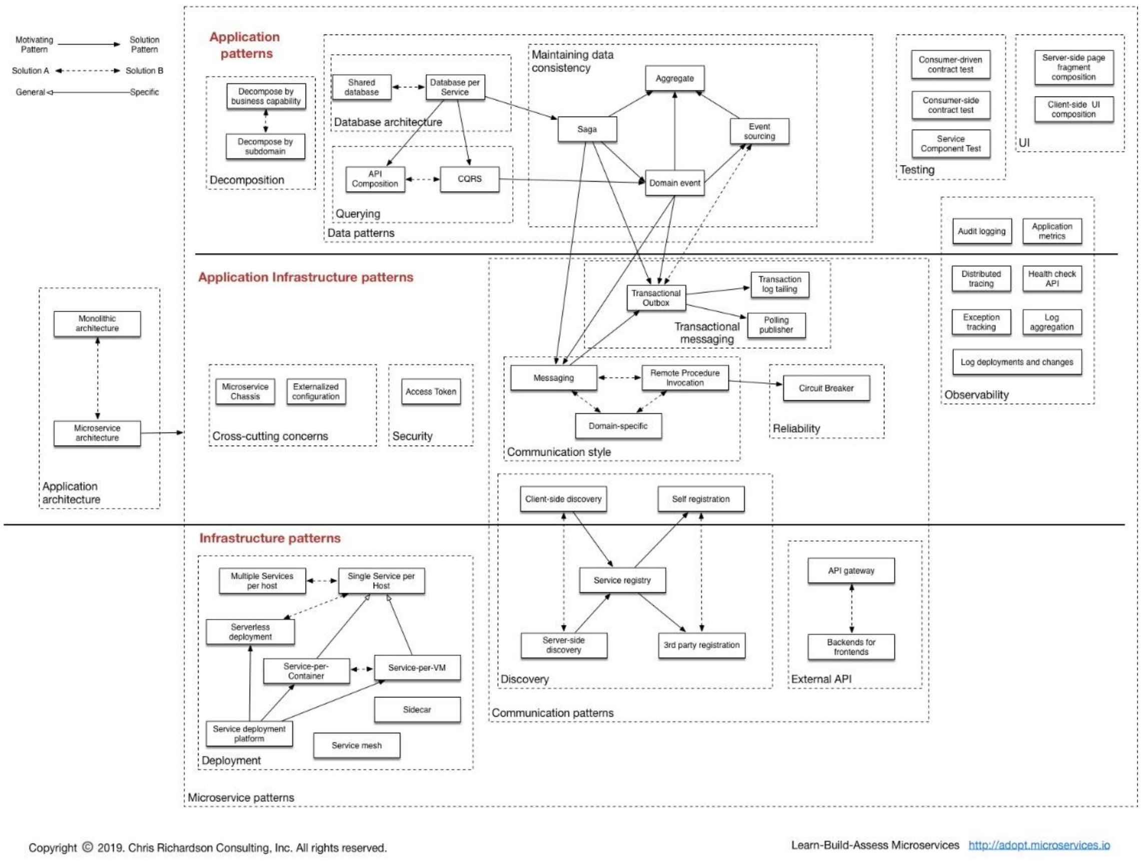


Figure 47. Microservices architecture patterns[5]

Driven by the common "alliance" between DevOps culture and microservices architectures, these authors also include the following cross-cutting steps for these migrations (Armin Balalaie et al., 2016a):

- Filling the Gap Between the Dev and Ops via Continuous Monitoring;
- Changing Team Structures (small cross-functional teams for each new service constructed).

[5] List of patterns from http://microservices.io/patterns/index.html, accessed in 28/08/2017

Table 10. Microservices patterns and categories

Core patterns	Decomposition	Security	UI patterns
• Monolithic architecture • Microservice architecture	• Decompose by business capability • Decompose by subdomain	• Access Token	• Server-side page fragment composition • Client-side UI composition
Cross cutting concerns	Testing	Observability	Deployment patterns
• Microservice chassis • Externalized configuration	• Service Component Test • Consumer-driven contract test • Consumer-side contract test	• Log aggregation • Application metrics • Audit logging • Distributed tracing • Exception tracking • Health check API • Log deployments and changes	• Multiple service instances per host • Service instance per host • Service instance per VM • Service instance per Container • Serverless Deployment • Service Deployment platform

Data management		Communication		
Database architecture	Maintaining data consistency	Communication style		Service discovery
• Database per Service • Shared database	• Saga • Event sourcing • Domain event • Agregate	• Remote Procedure Invocation • Messaging • Domain-specific protocol		• Client-side discovery • Server-side discovery • Service registry • Self registration • 3rd party registration
Querying		Reliability	External API	Transactional messaging
• API Composition • CQRS		• Circuit Breaker	• API gateway • Backend for front-end	• Transactional outbox • Transaction log tailing • Polling publisher

These steps were afterwards introduced as migration patterns, as depicted in Table 11.

Table 11. Microservices migration patterns *(Armin Balalaie et al., 2016a)*

Pattern name		
Enable the Continuous Integration	Recover the Current Architecture	Decompose the Monolith
Decompose the Monolith Based on Data Ownership	Change Code Dependency to Service Call	Introduce Service Discovery
Introduce Service Discovery Client	Introduce Internal Load Balancer	Introduce External Load Balancer
Introduce Circuit Breaker	Introduce Configuration Server	Introduce Edge Server
Containerize the Services	Deploy into a Cluster and Orchestrate Containers	Monitor the System and Provide Feedback

Along with adopting patterns, also bad practices (architectural bad smells) are identified (Davide Taibi & Lenarduzzi, 2018). Splitting a monolith, including splitting the connected data and libraries, is the most critical issue, resulting in potential maintenance issues when the cuts are not done properly. Moreover, the conversion to a distributed system increases the system's complexity, especially when dealing with connected services that need to be highly decoupled from any point of view, including communication and architecture (namely Hard-Coded Endpoints, Not Having an API Gateway, Inappropriate Service Intimacy, and Cyclic Dependency) (Davide Taibi & Lenarduzzi, 2018). The list of bad smells presented by Taibi and Lenarduzzi is depicted in Table 12.

Table 12. List of microservices bad smells (Davide Taibi & Lenarduzzi, 2018)

Microservices bad smells			
API Versioning	Hard-Coded Endpoints	Not Having an API Gateway	Too Many Standards
Cyclic Dependency	Inappropriate Service Intimacy	Shared Libraries	Wrong Cuts
ESB Usage	Microservice Greedy	Shared Persistency	

3.5. Conclusions

This chapter addressed the architecture design discipline, in terms of its lifecycle, the evolutionary design and the architecting as a continuous practice.

The works presented in Section 3.2 propose different inputs, target-users and viewpoints of architectures at each stage of the software development life cycle (SDLC). This has also led to proposals for proper usage of specific architecture methods depending on the stage of the SDLC.

Mainly because the stages of the SDLC initially require architectures that include information modeled at a higher-level of abstraction. As the SDLC evolves, typically the level of abstraction decreases.

Software architecture design, when performed in context of ASD, sometimes referred as “*agile architecting*”, promotes the emerging and incremental design of the architectural artifact, in a sense of avoiding “big design upfront” (BDUF). There is a lack of a pathway that guides agile architecting in an end-to-end approach (from business requirements to deployment). This research proposes in Chapter 5 a pathway that includes architecture design from software development life-cycle (SDLC) stages of software development that use ASD approaches, where two main artifacts are considered: a candidate logical architecture and a refined logical architecture.

If, in agile architecting, architecture evolves throughout the SDLC with the “just-enough” architecture for preventing BDUF and is oriented towards the “continuous” paradigm, the design must have as input only the “just-enough” requirements. Chapter 4 introduces processes for agile requirements, which later are used for the logical architecture design.

References

Aderaldo, C. M., Mendonca, N. C., Pahl, C., & Jamshidi, P. (2017). Benchmark Requirements for Microservices Architecture Research. In *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)* (pp. 8–13). IEEE. https://doi.org/10.1109/ECASE.2017.4

Ageling, W.-J. (2018). *Heart of Agile vs Modern Agile How does it compare?* Agile Insights.

Alshuqayran, N., Ali, N., & Evans, R. (2016). A systematic mapping study in microservice architecture. In *Service-Oriented Computing IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. https://doi.org/10.1109/SOCA.2016.15

Arnold, R. S. (1996). *Software change impact analysis*. IEEE Computer Society Press.

Atkinson, C., & Kuhne, T. (2003). Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5), 36–41.

Azevedo, S. M. F. (2014). *Refinement and variability techniques in model transformation of software requirements*. University of Minho.

Babar, M. A. (2013). Making Software Architecture and Agile Approaches Work Together: Foundations and Approaches. In *Agile Software Architecture: Aligning Agile Processes and Software Architectures*. Elsevier. https://doi.org/10.1016/B978-0-12-407772-0.00001-0

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016a). Microservices Architecture Enables DevOps: An Experience Report on Migration to a Cloud-Native Architecture. *IEEE Software*, 33(3), 42–52.

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016b). Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. In A. Celesti & P. Leitner (Eds.), *Advances in Service-Oriented and Cloud Computing. ESOCC Workshops 2015. Communications in Computer and Information Science* (Vol. 567, pp. 201–215). Springer, Cham. https://doi.org/10.1007/978-3-319-33313-7_15

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D. A., & Lynn, T. (2015). Microservices migration patterns. *Software: Practice and Experience*, 48(11), 2019–2042. https://doi.org/10.1002/spe.2608

Bayer, J., Flege, O., & Knauber, P. (1999). PuLSE: a methodology to develop software product lines. In *Proceedings of the 1999 symposium on Software reusability*. ACM.

Bayer, J., Muthig, D., & Göpfert, B. (2001). *The library system product line. A KobrA case study*. Fraunhofer IESE.

Berre, A. J. (2012). An Agile Model-Based Framework for Service Innovation for the Future Internet. In *International Conference on Web Engineering*. (pp. 1–4). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-35623-0_1

Bogsnes, B. (2008). *Implementing Beyond Budgeting: Unlocking the Performance Potential*. Wiley.

Booch, G. (2010). Enterprise Architecture and Technical Architecture. *IEEE Software*, 27(2), 96–96. https://doi.org/10.1109/MS.2010.42

Bosch, J. (2012). Building Products as Innovation Experiment Systems. In *Software Business. Proceedings of the International Conference of Software Business (ICSOB 2012)* (pp. 27–39). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-30746-1_3

Bosch, J. (2014). *Continuous Software Engineering*. Springer Cham.

Browning, T. R., & Eppinger, S. D. (2002). Modeling impacts of process architecture on cost and schedule risk in product development. *IEEE Transactions on Engineering Management*, 49(4), 428–442.

Brussel, H. Van, Wyns, J., Valckenaers, P., Bongaerts, L., & Peeters, P. (1998). Reference architecture for holonic manufacturing systems: PROSA. *Computers in Industry*, 37(3), 255–274. https://doi.org/10.1016/S0166-3615(98)00102-X

Cerny, T., Donahoo, M. J., & Trnka, M. (2018). Contextual understanding of microservice architecture. *ACM SIGAPP Applied Computing Review*, 17(4), 29–45. https://doi.org/10.1145/3183628.3183631

Chen, D., Doumeingts, G., & Vernadat, F. (2008). Architectures for enterprise integration and interoperability: Past, present and future. *Computers in Industry*, 59(7), 647–659. https://doi.org/10.1016/j.compind.2007.12.016

Chen, H. (2008). Towards service engineering: service orientation and business-IT alignment. In *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS)*. IEEE. https://doi.org/10.1109/HICSS.2008.462

Chen, L. (2018). Microservices: Architecting for Continuous Delivery and DevOps. In *IEEE International Conference on Software Architecture (ICSA)*. Seattle, USA: IEEE.

Chen, L., Ali Babar, M., & Nuseibeh, B. (2013). Characterizing Architecturally Significant Requirements. *IEEE Software*, 30(2), 38–45. https://doi.org/10.1109/MS.2012.174

Clements, P., & Bass, L. (2010). *Relating Business Goals to Architecturally Significant Requirements for Software Systems*. CMU/SEI-2010-TN-018.

Clements, P. C., & Northrop, L. M. (1996). *Software Architecture: An Executive Overview*. CMU/SEI-96-TR-003; ADA305470.

Clements, P., Garlan, D., Little, R., Nord, R., & Stafford, J. (2003). *Documenting software architectures: views and beyond*. IEEE.

Conway, M. E. (1968). How Do Committees Invent? *Datamation*, 28–31.

Di Francesco, P. (2017). Architecting microservices. In *IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*. https://doi.org/10.1109/ICSAW.2017.65

Diepenbrock, A., Rademacher, F., & Sachweh, S. (2017). An Ontology-based Approach for Domain-driven Design of Microservice Architectures. In *INFORMATIK2017*. https://doi.org/10.18420/in2017_177

DoD. (2009). *DoD Architecture Framework Version 2.0: Volume 2 - Architectural Data and Models*.

Dodani, M. H. (2006). A Picture is Worth a 1000 Words? *Journal of Object Technology*, 5(2), 35–40.

Douglass, B. (1999). *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns*. Addison-Wesley Professional.

Düllmann, T. F., & Van Hoorn, A. (2017). Model-driven Generation of Microservice Architectures for Benchmarking Performance and Resilience Engineering Approaches *. In *ICPE '17 Companion*. L'Aquila, Italy: ACM. https://doi.org/10.1145/3053600.3053627

Eeles, P., & Cripps, P. (2009). *The process of software architecting*. Pearson Education.

Erder, M., & Pureur, P. (2015). *Continuous architecture: Sustainable architecture in an agile and cloud-centric world*. Morgan Kaufmann.

Evans, E. (2004). *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley.

Familiar, B. (2015). *Microservices, IoT and Azure: Leveraging DevOps and Microservice Architecture to deliver SaaS Solutions*. Apress.

Fernandes, J. M., & Machado, R. J. (2016). *Requirements in Engineering Projects*. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-18597-2

Ferreira, N., Santos, N., Machado, R., Fernandes, J. E., & Gasević, D. (2014). A V-Model Approach for Business Process Requirements Elicitation in Cloud Design. In A. Bouguettaya, Q. Z. Sheng, & F. Daniel (Eds.), *Advanced Web Services* (pp. 551–578). Springer New York. https://doi.org/10.1007/978-1-4614-7535-4_23

Ferreira, N., Santos, N., Machado, R. J., & Gašević, D. (2012). Derivation of process-oriented logical architectures: An elicitation approach for cloud design. In *International Conference on Product Focused Software Process Improvement* (pp. 44–58). Springer.

Fitzgerald, B., & Stol, K.-J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123, 176–189. https://doi.org/10.1016/J.JSS.2015.06.063

Fong, E. N., & H., G. (1989). *Information Management Directions: The Integration Challenge*.

Fowler, M. (2004). *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional.

Francesco, P. Di, Malavolta, I., & Lago, P. (2017). Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In *IEEE International Conference on Software Architecture (ICSA)* (pp. 21–30). IEEE. https://doi.org/10.1109/ICSA.2017.24

Furda, A., Fidge, C., Zimmermann, O., Kelly, W., & Barros, A. (2018). Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency. *IEEE Software*, 35(3), 63–72. https://doi.org/10.1109/MS.2017.440134612

George, M. L., & George, M. (2003). *Lean six sigma for service*. New York, NY: McGraw-Hill.

Grau, B. R., & Lauenroth, K. (2014). *Requirements engineering and agile development - collaborative, just enough, just in time, sustainable*. International Requirements Engineering Board (IREB).

Gruhn, V., & Schäfer, C. (2015). BizDevOps: Because DevOps is Not the End of the Story. In Fujita H. & Guizzi G. (Eds.), SoMeT 2015: Intelligent Software Methodologies, Tools and

Techniques. *Communications in Computer and Information Science* (pp. 388–398). Springer, Cham. https://doi.org/10.1007/978-3-319-22689-7_30

Gysel, M., Kölbener, L., Giersche, W., & Zimmermann, O. (2016). Service cutter: A systematic approach to service decomposition. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. https://doi.org/10.1007/978-3-319-44482-6_12

Haberle, T., Charissis, L., Fehling, C., Nahm, J., & Leymann, F. (2015). The Connected Car in the Cloud: A Platform for Prototyping Telematics Services. *IEEE Software*, 32(6), 11–17. https://doi.org/10.1109/MS.2015.137

Highsmith, J. A. (2002). *Agile software development ecosystems* Addison-Wesley. Boston, MA.

Humble, J., & Farley, D. (2011). *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.

Hunt, A. (2015). *GROWS Overview*. Retrieved from http://growsmethod.com/grows_overview.html

IIBA. (2017). *Agile Extension to the BABOK Guide v2*. International Institute of Business Analysis.

Indrasiri, K., & Siriwardena, P. (2018). *Microservices for the enterprise: Designing, Developing, and Deploying*. Apress.

IREB. (2018). *IREB Certified Professional for Requirements Engineering - Advanced Level RE@Agile*.

ISO. (1998). I*SO/IEC 10746-1 Information technology – Open Distributed Processing – Reference Model: Overview*.

Jacobson, I., Griss, M., & Jonsson, P. (1997). *Software Reuse: Architecture, Process and Organization for Business Success*. Addison Wesley Longman.

Jamshidi, P., Pahl, C., Mendonca, N. C., Lewis, J., & Tilkov, S. (2018). Microservices: The Journey So Far and Challenges Ahead. *IEEE Software*, 35(3), 24–35. https://doi.org/10.1109/MS.2018.2141039

Jeston, J., & Nelis, J. (2008). *Business process management : practical guidelines to successful implementations*. Elsevier/Butterworth-Heinemann.

Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., & Huh, M. (1998). FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*.

Kanwal, F., Junaid, K., & Fahiem, M. A. (2010). A hybrid software architecture evaluation method for fdd-an agile process model. In *International Conference on Computational Intelligence and Software Engineering (CiSE)* (pp. 1–5). IEEE. https://doi.org/10.1109/CISE.2010.5676863

Kazman, R., Nord, R., & Klein, M. (2003). *A life-cycle view of architecture analysis and design methods*. Retrieved from http://www.dtic.mil/docs/citations/ADA421679

Kecskemeti, G., & Marosi, A. (2016). The ENTICE approach to decompose monolithic services into microservices. In *International Conference on High Performance Computing & Simulation (HPCS)* (pp. 591–596). IEEE. https://doi.org/10.1109/HPCSim.2016.7568389

Kerievsky, J. (2016). *An Introduction to Modern Agile*.

Kharbuja, R. (2016). *Designing a Business Platform using Microservices*. Technische Universität München.

Kirikova, M. (2017). Continuous Requirements Engineering. In *International Conference on Computer Systems and Technologies - CompSysTech'17*. Ruse, Bulgaria: ACM. https://doi.org/https://doi.org/10.1145/3134302.3134304

Kohavi, R., & Longbotham, R. (2017). Online Controlled Experiments and A/B Tests. In *Encyclopedia of machine learning and data mining* (pp. 922–929). Springer US.

Krause, L. (2014). *Microservices: Patterns and Applications - Designing Fine-grained Services by Applying Patterns*. microservicesbook.io.

Kruchten, P. (1995). The 4+1 View Model of Architecture. *IEEE Software*, 12(6), 42–50. https://doi.org/10.1109/52.469759

Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*, 29(6), 18–21. https://doi.org/10.1109/MS.2012.167

Lankhorst, M. (2009). Enterprise Architecture at Work: Modelling, Communication and Analysis. *The Enterprise Engineering Series.* Springer.

Lenarduzzi, V., & Taibi, D. (2018). Microservices, Continuous Architecture, and Technical Debt Interest: An Empirical Study. In *44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* . Prague, Czech republic: IEEE.

Lewis, G., Morris, E., Simanta, S., Smith, D., & Wrage, L. (2007). SMART: Analyzing the Reuse Potential of Legacy Components in a Service-Oriented Architecture Environment. In *AIAA Infotech@Aerospace 2007 Conference and Exhibit*. Reston, Virigina: American Institute of Aeronautics and Astronautics. https://doi.org/10.2514/6.2007-2865

Lewis, J., & Fowler, M. (2014). *Microservices: a definition of this new architectural term*. Retrieved March 24, 2017, from https://martinfowler.com/articles/microservices.html

Loukides, M. (2012). *What is DevOps?*

Martini, A., Besker, T., & Bosch, J. (2018). Technical Debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. *Science of Computer Programming*, 163, 42–61. https://doi.org/10.1016/J.SCICO.2018.03.007

Martini, A., Pareto, L., & Bosch, J. (2014). Role of Architects in Agile Organizations. In J. Bosch (Ed.), *Continuous Software Engineering*. Springer Cham.

Martini, A., Pareto, L., & Bosch, J. (2015). Towards Introducing Agile Architecting in Large Companies: The CAFFEA Framework. In *Agile Processes in Software Engineering and Extreme Programming. Proceedings of the International Conference on Agile Software Development (XP2015)* (pp. 218–223). Springer, Cham. https://doi.org/10.1007/978-3-319-18612-2_20

Matinlassi, M., Niemelä, E., & Dobrica, L. (2002). *Quality-driven architecture design and quality analysis method, A revolutionary initiation approach to a product line architecture*. VTT Technical Research Centre of Finland.

Mazlami, G., Cito, J., & Leitner, P. (2017). Extraction of Microservices from Monolithic Software Architectures. In *2017 IEEE International Conference on Web Services (ICWS)* (pp. 524–531). IEEE. https://doi.org/10.1109/ICWS.2017.61

Messina, A., Rizzo, R., Storniolo, P., Tripiciano, M., & Urso, A. (2016). The Database-is-the-Service Pattern for Microservice Architectures. In *International Conference on Information Technology in Bio- and Medical Informatics (ITBAM)* (pp. 223–233). Springer, Cham. https://doi.org/10.1007/978-3-319-43949-5_18

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice Architecture: Aligning Principles, Practices, and Culture*. O’Reilly.

Namiot, D., & Sneps-Sneppe, M. (2014). On micro-services architecture. *International Journal of Open Information Technologies*, 2(9), 24–27.

Newman, S. (2015). *Building microservices - Designing fine-grained systems*. O’Reilly Media, Inc.

Novack, J. (2016). *Shu Ha Ri: An Agile Adoption Pattern*. SolutionsIQ.

O’Connor, R. V., Elger, P., & Clarke, P. M. (2017). Continuous software engineering-A microservices architecture perspective. *Journal of Software: Evolution and Process*, 29(11), e1866. https://doi.org/10.1002/smr.1866

Olsson, H. H., Alahyari, H., & Bosch, J. (2012). Climbing the “Stairway to Heaven” – A Multiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software. In *38th Euromicro Conference on Software Engineering and Advanced Applications* (pp. 392–399). IEEE. https://doi.org/10.1109/SEAA.2012.54

Olsson, H. H., & Bosch, J. (2014). Climbing the “Stairway to Heaven”: Evolving From Agile Development to Continuous Deployment of Software. In J. Bosch (Ed.), *Continuous Software*

Engineering (pp. 15–27). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-11283-1_2

OMG. (2003). *MDA Guide Version 1.0.1*. Object Management Group.

Overby, E., Bharadwaj, A., & Sambamurthy, V. (2005). A Framework for Enterprise Agility and the Enabling Role of Digital Options. In *Business Agility and Information Technology Diffusion. IFIP International Working Conference on Business Agility and Information Technology Diffusion (TDIT 2005)* (pp. 295–312). Boston: Springer. https://doi.org/10.1007/0-387-25590-7_19

Pahl, C., & Jamshidi, P. (2016). Microservices: A Systematic Mapping Study. In *6th International Conference on Cloud Computing and Services Science (CLOSER)* (Vol. 1, pp. 137–146). SCITEPRESS - Science and and Technology Publications. https://doi.org/10.5220/0005785501370146

Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., & Josuttis, N. (2017). Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software*, 34(1), 91–98. https://doi.org/10.1109/MS.2017.24

Pérez, J., Díaz, J., Garbajosa, J., & Yagüe, A. (2014). Bridging User Stories and Software Architecture: A Tailored Scrum for Agile Architecting. In A. W. . Ali Babar, Muhammad ; Brown & I. Mistrik (Eds.), *Agile Software Architecture - Aligning Agile Processes and Software Architectures* (pp. 215–241). Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-407772-0.00008-3

Pohl, K. (2010). *Requirements Engineering*. Springer.

Quiroz, A., Kim, H., Parashar, M., Gnanasambandam, N., & Sharma, N. (2009). Towards autonomic workload provisioning for enterprise Grids and clouds. In *2009 10th IEEE/ACM International Conference on Grid Computing* (pp. 50–57). IEEE. https://doi.org/10.1109/GRID.2009.5353066

Rademacher, F., Sachweh, S., & Zundorf, A. (2017). Differences between Model-Driven Development of Service-Oriented and Microservice Architecture. In *IEEE International Conference on Software Architecture Workshops (ICSAW)* (pp. 38–45). IEEE. https://doi.org/10.1109/ICSAW.2017.32

Rademacher, F., Sachweh, S., & Zündorf, A. (2018a). *Analysis of Service-oriented Modeling Approaches for Viewpoint-specific Model-driven Development of Microservice Architecture*. ArXiv Preprint ArXiv:1804.09946.

Rademacher, F., Sachweh, S., & Zündorf, A. (2018b). Towards a UML Profile for Domain-Driven Design of Microservice Architectures. In *Software Engineering and Formal Methods* (pp. 230–245). Springer. https://doi.org/10.1007/978-3-319-74781-1_17

Rademacher, F., Sorgalla, J., & Sachweh, S. (2018). Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective. *IEEE Software*, 35(3), 36–43. https://doi.org/10.1109/MS.2018.2141028

Rademacher, F., Sorgalla, J., Sachweh, S., & Zündorf, A. (2018). *Towards a Viewpoint-specific Metamodel for Model-driven Development of Microservice Architecture*.

Rademacher, F., Sorgalla, J., Wizenty, P. N., Sachweh, S., & Zündorf, A. (2018). Microservice Architecture and Model-driven Development: Yet Singles, Soon Married (?). In *Second International Workshop on Microservices: Agile and DevOps Experience (MADE18) collocated with XP18*. Porto, Portugal: ACM.

Reddy, A., Govindarajulu, P., & Naidu, M. (2007). A Process Model for Software Architecture. *International Journal of Computer Science and Network Security*, 7(4), 272–280.

Richardson, C. (2018). *Microservice Patterns (1st ed.)*. Manning.

Ries, E. (2011). *The lean startup: How today’s entrepreneurs use continuous innovation to create radically successful businesses*. Crown Books.

Rozanski, N., & Woods, E. (2005). *Software systems architecture : working with stakeholders using viewpoints and perspectives*. Addison-Wesley.

Sommerville, I. (2007). *Software Engineering*. London: Pearson/Addison Wesley.

Soni, D., Nord, R. L., & Hofmeister, C. (1995). Software architecture in industrial applications. In *Proceedings of the 17th international conference on Software engineering - ICSE ’95* (pp. 196–207). New York, New York, USA: ACM Press. https://doi.org/10.1145/225014.225033

Sowa, J. F., & Zachman, J. A. (1992). Extending and formalizing the framework for information systems architecture. *IBM Systems Journal,* 31(3), 590–616.

Stal, M. (2014). Refactoring Software Architectures. In *Agile Software Architecture - Aligning Agile Processes and Software Architectures*. Elsevier Inc.

Stålhane, T. (2014). *Change Impact Analysis in Agile Development*.

Steinegger, R. H., Giessler, P., Hippchen, B., & Abeck, S. (2017). Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications. In *Third International Conference on Advances and Trends in Software Engineering (SOFTENG'17)*. IARIA.

Taibi, D., & Lenarduzzi, V. (2018). On the Definition of Microservice Bad Smells. *IEEE Software*, 35(3), 56–62. https://doi.org/10.1109/MS.2018.2141031

Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. I*EEE Cloud Computing*, 4(5), 22–32. https://doi.org/10.1109/MCC.2017.4250931

Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural Patterns for Microservices: A Systematic Mapping Study. In *Int.Conference on Cloud Computing and Services Science, CLOSER*. INSTICC.

Taibi, D., Lenarduzzi, V., Pahl, C., & Janes, A. (2017). Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In P*roceedings of the XP2017 Scientific Workshops* (p. 23). ACM.

Thönes, J. (2015). Microservices. *IEEE Software*, 32(1), 116–116. https://doi.org/10.1109/MS.2015.11

Tom, E., Aurum, A., & Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, 86(6), 1498–1516. https://doi.org/10.1016/J.JSS.2012.12.052

UML, O. M. G. (2011). *2.4. 1 superstructure specification. document formal/2011-08-06*. Technical report, OMG.

Urbaczewski, L., & Mrdalj, S. (2006). A comparison of enterprise architecture frameworks. *Issues in Informations Systems*, 7(2), 18–23.

Weiss, D. M. (1999). *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley Professional.

Wiegers, K. (2014). *Best Practices for Change Impact Analysis* | Jama Software. Retrieved December 4, 2018, from https://www.jamasoftware.com/blog/change-impact-analysis-2/

Wiggins, A. (2012). *The Twelve-Factor App*. https://12factor.net.

Winter, R., & Fischer, R. (2006). Essential Layers, Artifacts, and Dependencies of Enterprise Architecture. *10th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW)*.

Xiao, L., Cai, Y., Kazman, R., Mo, R., & Feng, Q. (2016). Identifying and quantifying architectural debt. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16* (pp. 488–498). New York, New York, USA: ACM Press. https://doi.org/10.1145/2884781.2884822

Xiao, Z., Wijegunaratne, I., & Qiang, X. (2016). Reflections on SOA and Microservices. In *2016 4th International Conference on Enterprise Systems (ES)* (pp. 60–67). IEEE. https://doi.org/10.1109/ES.2016.14

Yale Yu, Silveira, H., & Sundaram, M. (2016). A microservice based reference architecture model in the context of enterprise architecture. In *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)* (pp. 1856–1860). IEEE. https://doi.org/10.1109/IMCEC.2016.7867539

Zachman, J. A. (1987). A framework for information systems architecture. *IBM Systems Journal*, 26(3), 276–292.

Zachman, J. A. (2011). *The Zachman Framework for Enterprise Architecture*.

Zimmermann, O. (2017). Microservices tenets. *Computer Science-Research and Development*. Retrieved from https://link.springer.com/article/10.1007/s00450-016-0337-0

Zúñiga-Prieto, M., Insfran, E., & Abrahao, S. (2016). Incremental Integration of Microservices in Cloud Applications. In *25th Int. Conf. on Information Systems Development (ISD2016).* Retrieved from http://aisel.aisnet.org/isd2014/proceedings2016/ISDMethodologies/8/

Zwegers, A. J. R. (1998). *On systems architecting : a study in shop floor control to determine architecting concepts and principles*. Technische Universiteit Eindhoven.

PART III

CONTRIBUTIONS

Chapter 4 - A Requirements Modeling Approach for Agile Settings

This chapter discusses requirements engineering (RE) in ASD processes, addressing the necessary information as stakeholders communicates their business needs or their "*minimum viable product*" (MVP). Thus, this chapter introduces both upfront and emerging approaches for RE. Regarding the latter, an approach called "*Decomposing User Agile Requirements ArTEfacts*" (DUARTE) is proposed. Both RE approaches are evaluated and discussed using three demonstration cases, two for upfront modeling and one applying DUARTE approach. This chapter ends with the conclusions.

Chapter 4 - A Requirements Modeling Approach for Agile Settings

"*Without requirements and design, programming is the art of adding bugs to an empty text file*"
***Louis Srygley**, Application Architect*

"The biggest enemy of agility (at the code level) is unnecessary complexity. .... Don't write code or future scenarios that may never occur."
***Allen Holub**, Agile Consultant*

4.1. Introduction

At the time that the 'Agile Manifesto' (Agile Alliance, 2001) was proposed, there was a big shift on focusing in delivering software and less in technical documentation and specifications. Based in one of the values of the Manifesto, '*Working software over comprehensive documentation*', specification of requirements have been reducing to as minimum as possible. Thus, the use of software models was also reduced, both in requirements and in design tasks, where only models that actually help teams develop software are used (K Schwaber & Beedle, 2001).

In plan-driven approaches (e.g., Waterfall), tasks related to Requirements Engineering (RE) discipline are traditionally managed in a phase separated in time from design and development. In change-driven approaches, like ASD, RE discipline – also called "*Agile RE*" – activities remain the same but are executed continuously (Grau & Lauenroth, 2014), and takes an iterative discovery approach (Cao & Ramesh, 2008). Elicitation, analysis, and validation are present in all ASD processes (Paetsch, Eberlein, & Maurer, 2003). Additionally, requirements modeling require an agile approach in order to prevent unnecessary efforts in "*You Aren't Gonna Need It*" (YAGNI) features, hence the need for an Agile Modeling (AM) (S Ambler, 2002) approach.

In ASD frameworks, the requirements are included in a product backlog, which then drives the development process, thus most of the RE activities are always performed earlier. ASD widely use User Stories (Cohn, 2004b) as items in the backlog for "reminders of a conversation" about a functionality. However, using only User Stories, without attached requirements specifications or models, may be insufficient to assure a common understanding or, in case of multi-teams, to clearly define inter-systems interaction.

Stakeholders must able to communicate in what way a future solution improves their business, by defining the product roadmap. A product roadmap is an initial high level project scope and direction (IIBA, 2017). Typically, a first release on a new product encompasses a product's subset able to address priority scenarios, previously identified in order to respond to market needs. In fact, many of these product releases are market-driven, where the release is deployed into the market so it is possible to get feedback from it, *i.e.*, a minimum viable product (MVP). Alongside with these requirements concerns, projects struggle to design candidate architectures for the MVP, endangering development when they conclude that the architecture requires modifications and updates. In an era where software development is more and more agile-oriented, the upfront effort is replaced by the emergence of the design throughout iterative cycles. Such efforts are in opposition to "*Big Design Upfront*" (BDUF). In ASD contexts, BDUF approaches often result in features that are disregarded after some time (YAGNI features).

From business needs to agile logical architecting, this chapter introduces requirements modeling activities and artifacts that are part of an integrated modeling roadmap (Figure 48). This roadmap is proposes a sequential order of artifacts that relate to the outputs that are used within the *Agile Modeling Process for Logical Architectures* (AMPLA), presented in Figure 49. In terms of the integrated modeling roadmap, this chapter encompasses stages 1 and 2. Requirements modeling, called "***D***ecomposing ***U***ser ***A***gile ***R***equirements Ar***TE***facts" (DUARTE) process uses ASD known techniques for deriving an UML use case diagram. Then, the candidate logical architecture will be derived (cf. Chapter 5) using as input the diagrams from DUARTE.

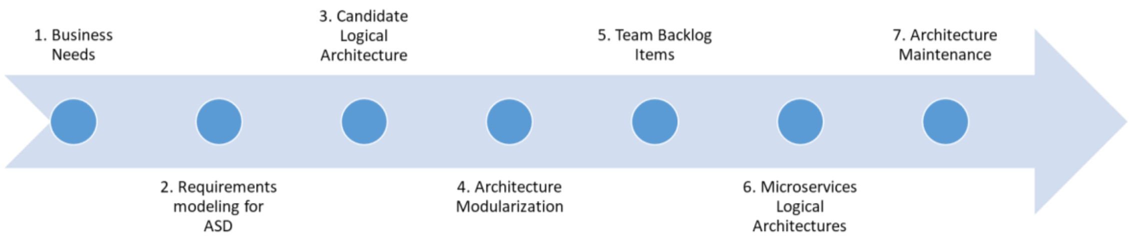

1. Business Needs
2. Requirements modeling for ASD
3. Candidate Logical Architecture
4. Architecture Modularization
5. Team Backlog Items
6. Microservices Logical Architectures
7. Architecture Maintenance

Figure 48. Integrated modeling roadmap

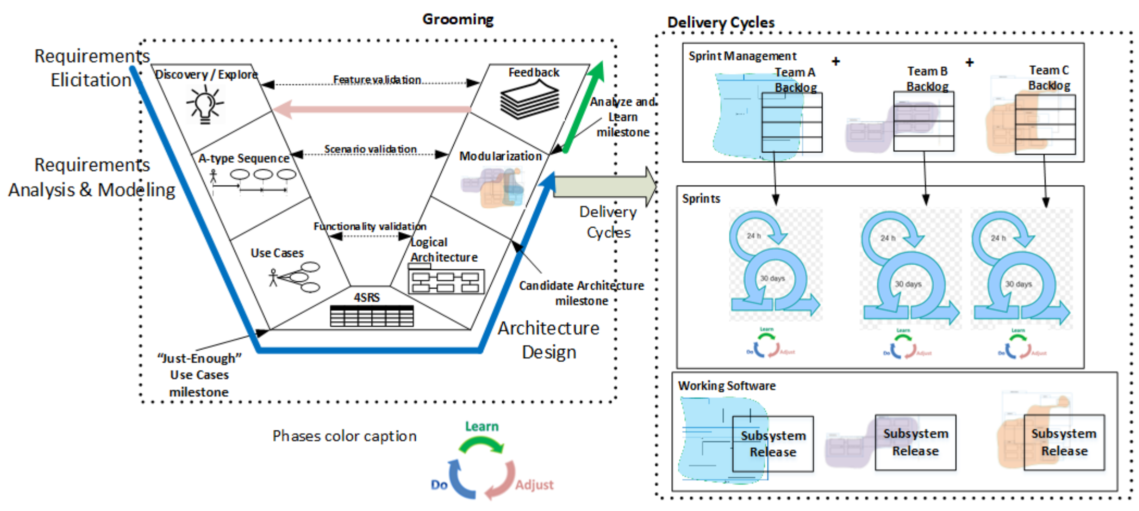


Figure 49. Overview of AMPLA

The research addressed in chapter is the result of applying RE approaches in ASD-based software research projects. Firstly, model-based approaches are hybrid – *i.e.*, first performing requirements and design in waterfall and implementation in ASD. That was the case of ISOFIN and iFloW projects. Then, in order to address situations of unknown and need to be discovered requirements, the research addressed emerging requirements, which was applied in the UH4SP project. The contributions of the projects are summarized in Table 13.

Table 13. Contributions of projects in candidate architectures

Research contribution \ demonstration case	UH4SP	iFloW	ISOFIN
Upfront Requirements modeling		X	X
Emergent Requirements modeling	X		

This chapter is is structured as follows:

- Section 4.2 discusses upfront approaches for RE;
- In opposition, Section 4.3 discusses upfront and emerging approaches for RE, where an approach called DUARTE is proposed;
- Section 4.4 describes three demonstration cases, two for upfront modeling and one applying DUARTE approach, as well as the discussions from the three cases;
- Section 4.5 presents the chapter's conclusions;
- The chapter ends with complimentary reading.

4.2. Upfront Modeling in ASD projects

When requirements are known upfront and considered stable, plan-driven approaches may be used for RE, even if further phases use change-driven approaches like ASD frameworks. Such situation is seen like a "hybrid" approach. In this case, requirements are gathered upfront.

A process that uses requirements modeling together with Scrum-based cycles (or "Sprints") is depicted in Figure 50. The process is composed of three phases: Initialization, Implementation, and Deployment.

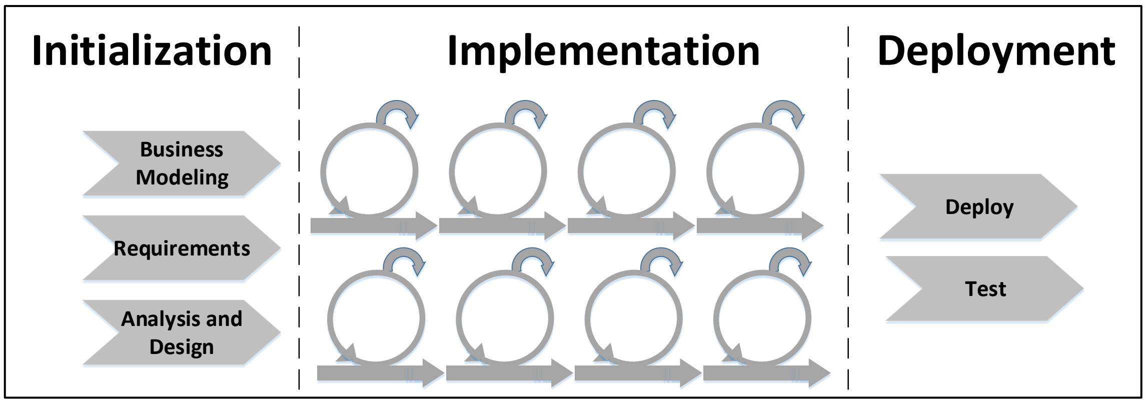


Figure 50. Hybrid ASD process with upfront requirements modeling

The initialization phase includes typical activities from domain engineering, RE and design. The implementation phase uses small iterations and incremental releases in the form of Scrum Sprints. Finally, the Deployment phase is similar to the Transition phase of RUP.

Within the initialization phase, the objective was to develop a product backlog artifact in order to start the development phase in the form of Sprints that, due to the perceived complexity of the project, was delivered together with widely accepted forms of requirements documentation.

Deriving a Use case-driven Product Backlogs

The Business Modeling results - organization's processes and current gaps - are documented in a report designated as '*As-Is report*'. Then, the requirements are elicited, formally specified in the form of UML use cases, quality (non-functional) requirements and the logical architecture (UML component diagram), documented in a report designated as '*To-Be report*' (which was constantly updated as the implementation went along). UML use cases and UML component diagram compose the '*Solution Requirements Specification*' that result in the '*To-Be Report*'. The '*To-Be*' use case models then compose the '*Product Backlog*'.

This differs from ASD frameworks. XP uses Themes, Epics and User Stories for addressing requirements. Scrum and Kanban only prescribe using work items in the Product Backlog regardless the form (however commonly these items are in form of User Stories). Test Driven Development (TDD), Acceptance Test Driven Development (ATDD), BDD and Specification by Example (SBE), use testable scenarios as input for the software development. However, approaches like the Agile Unified Process (AUP) (SW Ambler, 2005), Jacobson's "Use Case 2.0" (Jacobson, Spence, & Bittner, 2011), or others like (Cho, 2009; Durdik, 2011) use UML and use case-driven Product Backlogs.

More detail on these tasks, including the activities during the Sprint 0 event, are represented in a SPEM diagram in Figure 51. In this diagram, roles are explicitly assigned to the activities and tasks. Each task outputs a work product or a deliverable. In the case of the '*Sprint 0*', this ceremony includes specific tasks, namely prioritizing and estimating use cases, constructing the product backlog (that result in the '*Product Backlog*') and finally the planning of the following Sprints (that result in the '*Sprint Backlog*').

Then, this backlog is used to define a '*Sprint Backlog*' for every Sprint, as typically occurs in, *e.g.*, Scrum projects. In Figure 52 is depicted an example of a '*Sprint Backlog'* tracking sheet, composed by use cases and whose progress was monitored. In these type of backlogs, each use case elicited during Initialization is a Product Backlog Item (PBI). The logical architecture model is used to support development of components, but it does not have direct impact in '*Product Backlog Construction*'. Within this task, the logical architecture may be consulted in order to include in '*Sprint Backlog*' use cases where components have any kind of associations, but the construction of the backlog is a responsibility that relies uniquely the team.

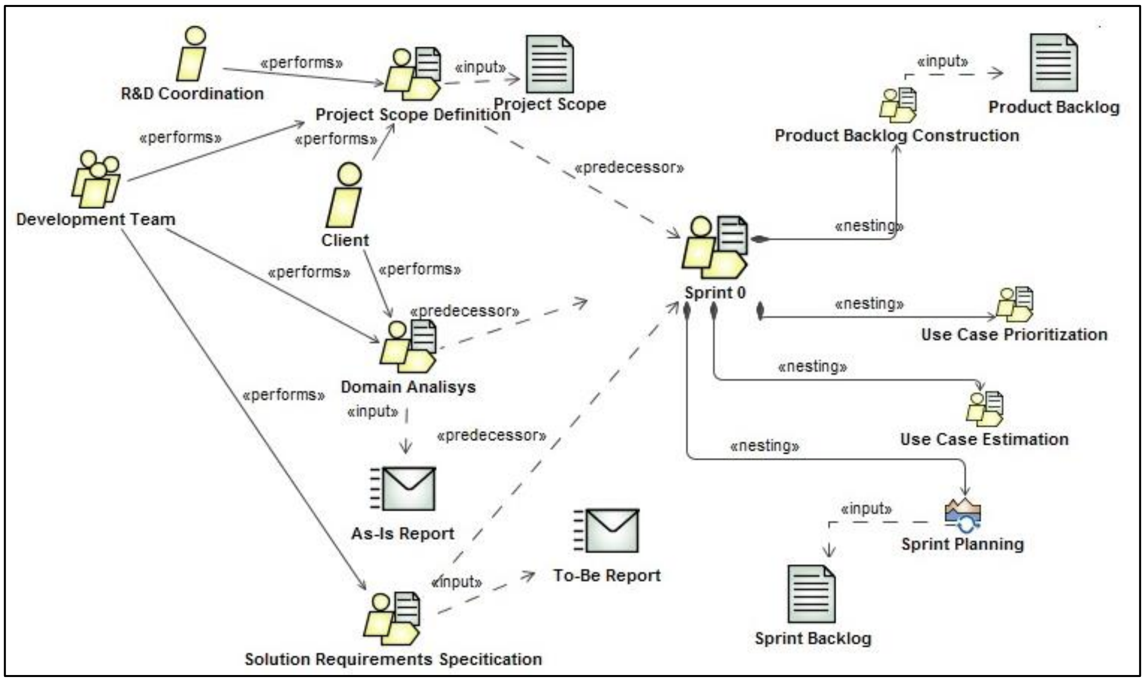
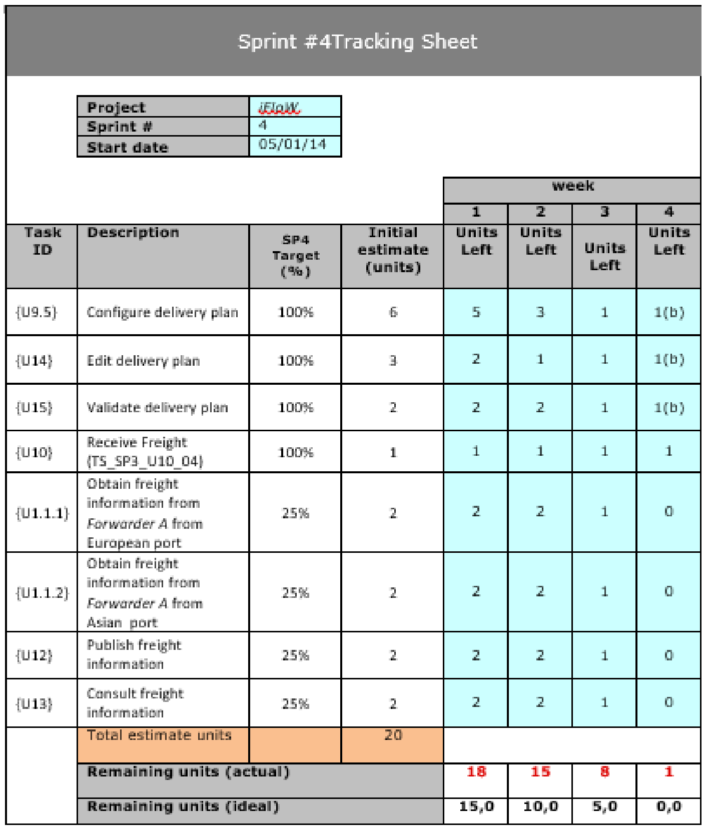


Figure 51. SPEM diagram for Initialization phase



Sprint #4Tracking Sheet

Project	iFloW
Sprint #	4
Start date	05/01/14

				week			
				1	2	3	4
Task ID	Description	SP4 Target (%)	Initial estimate (units)	Units Left	Units Left	Units Left	Units Left
{U9.5}	Configure delivery plan	100%	6	5	3	1	1(b)
{U14}	Edit delivery plan	100%	3	2	1	1	1(b)
{U15}	Validate delivery plan	100%	2	2	2	1	1(b)
{U10}	Receive Freight (TS_SP3_U10_04)	100%	1	1	1	1	1
{U1.1.1}	Obtain freight information from *Forwarder A* from European port	25%	2	2	2	1	0
{U1.1.2}	Obtain freight information from *Forwarder A* from Asian port	25%	2	2	2	1	0
{U12}	Publish freight information	25%	2	2	2	1	0
{U13}	Consult freight information	25%	2	2	2	1	0
	Total estimate units		20				
	Remaining units (actual)			18	15	8	1
	Remaining units (ideal)			15,0	10,0	5,0	0,0

Figure 52. Example of a Sprint Backlog based in Use Cases

Deriving a User story-driven Product Backlog

In the previous section, the UML use cases and UML component diagram (that compose the '*Solution Requirements Specification*') were modeled upfront and used as input in a Sprint 0. As previously mentioned, ASD frameworks like Scrum, XP or Kanban flavor using User Stories as starting points for PBI's (although only XP clearly states using User Stories). Using "INVEST" criteria ("Independent", "Negotiable", "Valuable", "Estimable", "Small", "Testable"), these frameworks claim User Stories to be better suitable for defining work items within a timebox such as Sprints rather than Use Cases.

Thus, this section presents a hybrid approach that includes an upfront requirements modeling (e.g., in UML) should use a User Story-driven Product Backlog. The use of Use Cases and User Stories regard a different context of use (Cohn, 2004a) and different granularity (Leffingwell, 2010). In order to deal with these differences, an approach that supports traceability between requirements and software is proposed.

Figure 53 proposes a traceable path, using a V-Model, between requirements in UML Use Cases, a logical architecture in UML Components and delivery of User Stories. The process uses the V-Model proposed by Ferreira *et al.* for deriving a logical architecture aligned with requirement modeled in Use Cases (Nuno Ferreira, Santos, Machado, Fernandes, & Gasević, 2014). The V-Model uses models in a successive way, where a previous model is input for a next one. Namely, the requirements elicitation included a definition of the solution's executing business processes, sequential ordering of functionalities (afterwards modeled in *A-type* sequence diagrams (Nuno Ferreira et al., 2014)) and finally modeling in Use Case diagrams.

The pathway to elicit business process needs ("Input from Business Processes") required to derive software requirements (*A-type* sequence diagrams and UML Use Cases) is described in previous works (N. Ferreira, Santos, Machado, & Gašević, 2013; N Ferreira, Santos, Soares, Machado, & Gasevic, 2012; Nuno Ferreira et al., 2014; Nuno Ferreira, Santos, Machado, & Gasevic, 2012; Nuno Ferreira, Santos, Soares, Machado, & Gašević, 2013; Santos, Ferreira, & Machado, 2017). It is not the purpose of this section to address the derivation of the UML Use Cases. Rather, what it should be retained from this process is that requirements were elicited in UML Use Cases and afterwards the architecture was designed, both performed upfront before any kind of development tasks.

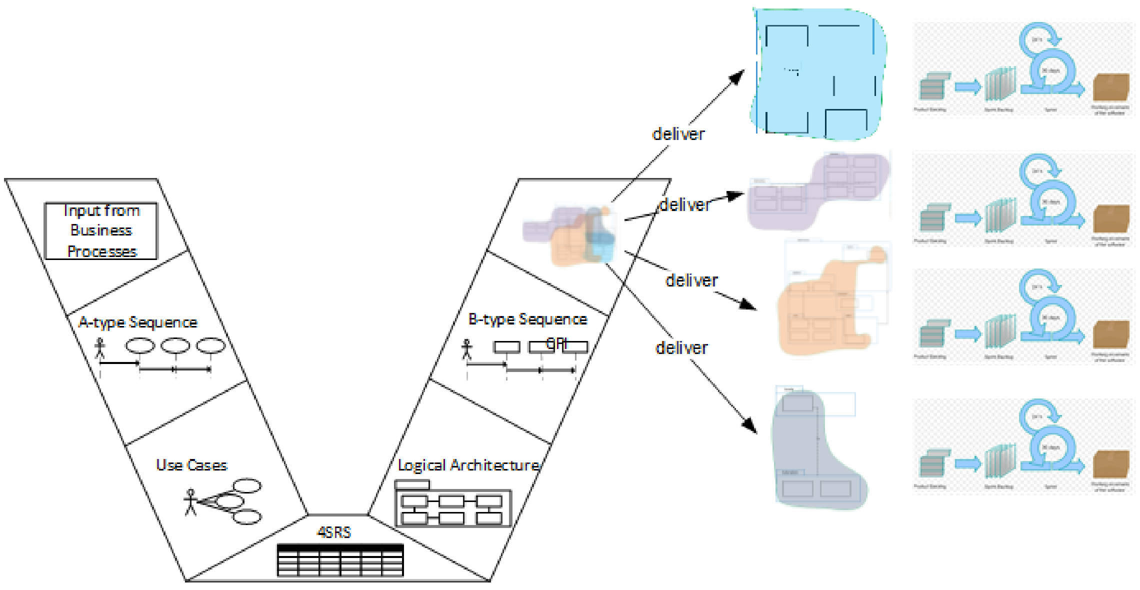


Figure 53. The result of the V-Model to be delivered to multiple Scrum teams

Some software architectures are too complex to be directly used by agile teams as their requirements input artifact. Additionally, each agile team may be responsible for implementing only some parts of the whole architecture. The software logical architecture is the artifact typically delivered to implementation teams (Scrum or other), and it regards the architecture diagram that is modularized. The components (representing software functionalities) that compose it are classified as belonging to a given software module, and thus covered by the module (Figure 54).

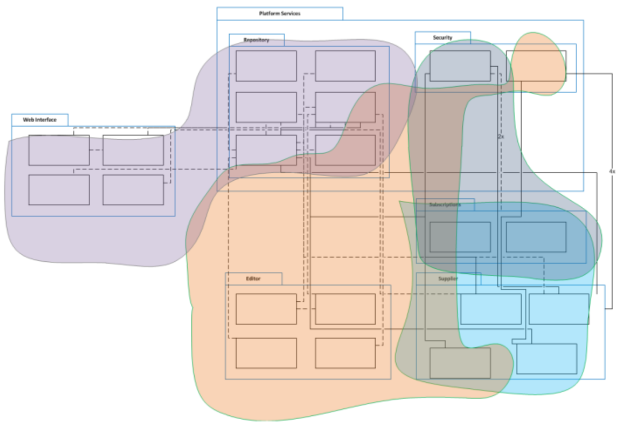


Figure 54. Architecture modularization example

After the modularization, all entities not directly connected to the module must be removed from the resulting diagram. Inside the system border defined for the given module, through the respective module coverage, the components are maintained as originally characterized. Additionally, it also allows depicting the interfaces that are outside the system border.

Due to the fact that, in the 4SRS method, the components are derived through the decomposition of Use Cases in three different types (interface, data and control), one User Story is created for each component, as depicted in Figure 55, as it complies with the greater flexibility for the Product Owner to follow the team's work.

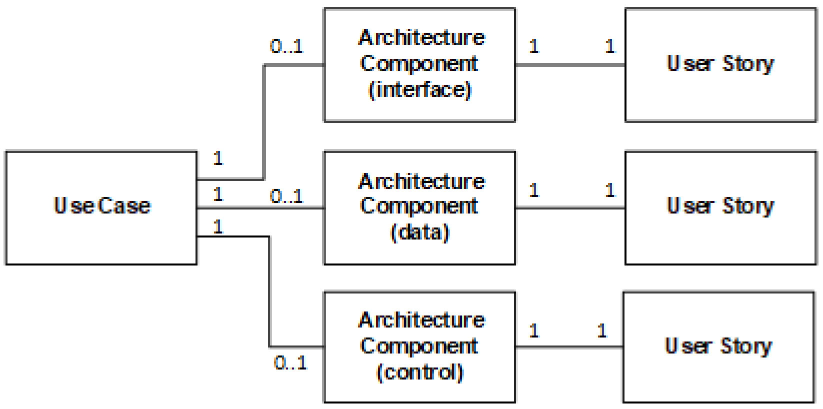

Use Case
1
1
1
0..1
0..1
0..1
Architecture Component (interface)
Architecture Component (data)
Architecture Component (control)
1
1
1
1
1
1
User Story
User Story
User Story

Figure 55. Relation between Use Cases, Components and User Stories

In short, there are two model inputs for composing the Product Backlog: the architecture components and the Use cases. The derivation of User Stories for composing the backlog is based in the components, as depicted in Figure 55, to define the size of the backlog but also using inputs from architecture components and the Use cases for defining the information about the user story. This discussion is described in detail in Chapter 6.

4.3. *Agile logical architecting with the 4SRS method*

Chapter 1 introduced Agile Modeling Process for Logical Architectures (AMPLA), a process for model derivation applicable in an Agile RE and AM context. AMPLA is a process for candidate architecture design based on successive and specific artifacts generation, which starts by discovery and exploration of user needs, *A-type sequence diagrams*, use case models, a software logical architecture diagram, feedbacks from customers and issues identification, and the consequent software delivery.

This section introduces the model derivation until the candidate architecture design (Figure 56), relating to requirements elicitation and modeling. The generated artifacts and the alignment between the explored needs and modeled software requirements can be represented by a V-Model (Figure 56). In our proposed V-Model, the artifacts are generated based on the rationale and in the information existing in previously defined artifacts, i.e., *A-type sequence diagrams* are based on discovered and explored scenarios, use case model is based on *A-type sequence diagrams*, the logical architecture is based on the use case model, and finally feedback from customers based in the logical architecture. After the feedback and consequent learning and adjustments (if needed), the approach ends with the candidate logical architecture, which is then used as input for defining the required backlog items for delivering the software. When software delivery begins, the process is performed in typical cycles, whether in Scrum, Kanban, or other frameworks.

AMPLA is composed by artifacts, phases and milestones (Figure 56). AMPLA's successive model derivation is performed in iterative cycles, easing the execution of agile feedback loops and hence contributing to the process' agility. These loops encompass phases of (1) Do; (2) Learn; and (3) Adjust. Each loop may include all phases of AMPLA, or just a subset of them.

The artifacts should be modeled incrementally, where the ideal is to have short cycles to have design prototypes ready for customer analysis and feedback. It is better to deliver small portions of models and quickly validate with customers that the right product is being developed, rather than deliver bigger portions of models and realize that they do not reflect customer needs. Hence, the path encompassing "Discovery / Explore", "*A-type sequence diagrams*", "use cases", "4SRS" and "logical architecture" relate to (1) Do phase. "Feedback" from customers relate to the (2) Learn. Finally, the (3) Adjust is reflected in new, changed or eliminated artifacts output during the (1) Do phase.

It has three established phases: (i) Requirements Elicitation; (ii) Requirements Analysis & Modeling; (iii) Architecture Design; and (iv) Delivery Cycles. For milestones, besides the checkpoints before passing from one phase to another, there are additional ones within phases that aim promoting agility. In Requirements Analysis & Modeling, the use cases refinement are validated for having "just-enough" detail before executing the 4SRS method. The execution of the 4SRS outputs a candidate version of the logical architecture, which is the first system model prototype that is presented to stakeholders for feedback. The last milestone from this phase relates to the feedback gathered from the architecture, where issues and adjustments are identified before passing on to the Delivery Cycles phase.

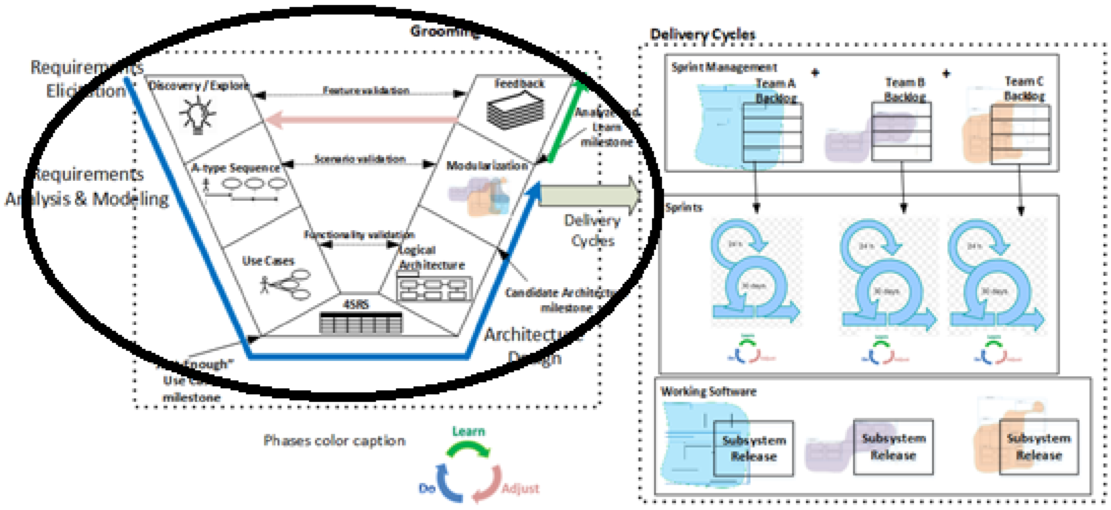


Figure 56. Candidate architecture design of AMPLA

The mashup of agile practices and industry coins (e.g., Scrum, XP, MVP, DevOps, large-scale agile, Squads/Tribes, Management 3.0, and many others) cover all software and application lifecycle. Although none of this practices relate to RE discipline, or specifically to Agile modeling (AM) (S Ambler, 2002), performing this practices into an ASD process has direct implications on how RE practices are performed and how artifacts are built.

Applying AM should start by enabling a first iteration of requirements modeling, which is then the basis for further refinements and emerges, as the software increments are being delivered throughout the Sprints. The inception, like the pregame phase or Sprint zero in Scrum, aims providing a shared understanding of the project and the required information for the development phase. In the same line of reasoning, Ambler presents an evolution and emerge-oriented approach for using models in ASD, called "Agile Model-Driven Development" (AMDD) (SW Ambler, 2003), where the starting point is "just-enough" requirements and architecture, which are updated alongside Delivery Cycles phase.

This section describes an AM process for modeling emergent user requirements towards a candidate logical architecture, called "***D****ecomposing* ***U****ser* ***A****gile* ***R****equirements* *ar****TE****facts*" (DUARTE) approach. It starts in eliciting user requirements, using ASD techniques like Lean Startup (Ries, 2011), Design Thinking (Brown, 2009), Domain-driven Design (DDD) (Evans, 2004), Behavior-driven Development (BDD) (Smart, 2015), and others, which results in modeling UML Use Cases. Requirements are modeled until they are considered as "just-enough", until used within the 4SRS method. The 4SRS allows deriving a candidate logical architecture, to be used within ASD delivery cycles.

The Decomposing User Agile Requirements arTEfacts (DUARTE) approach

In opposition to upfront and stable requirements, "*Agile RE*" is executed continuously (Grau & Lauenroth, 2014), and takes an iterative discovery approach (Cao & Ramesh, 2008). This section introduces DUARTE, a requirements-related approach performed within AMPLA, which aims at iteratively elicit and model emerging requirements that will later be used within the 4SRS method in order to derive the candidate logical architecture.

DUARTE is about eliciting and modeling requirements, promoting agility (and hence Agile RE practices) by including practices and mindsets from approaches like Lean Startup, Design Thinking, DDD, BDD, Kent Beck's 3X[6] and BizDev (Fitzgerald & Stol, 2017). Additionally, uses agile practices in order to deliver small increments (of a requirements package) and to promote continuous customer feedback (Figure 57).

Lean Startup is a hypothesis-driven approach, where a "Build-Measure-Learn" cycle is the basis for supporting product development adequate to the market. Design Thinking addresses understanding the customer need through systematic exploration, in order to understand the right product to develop. It is also worth referring that this cycle is inspired by the "Plan-Do-Check-Act" (PDCA) from Lean Manufacturing. The agile scaling framework DAD also encompasses an "Exploratory lifecycle" that uses the "Build-Measure-Learn" cycle from Lean Startup. With Lean Startup, the following concepts arose: (i) Minimal Viable Product (MVP), (ii) Minimal Marketable Feature (MMF), (iii) Minimal Marketable Release (MMR), and (iv) Minimal Marketable Product (MMP). (i) An MVP is a version of a new product that is created with the least effort possible to be used for validated learning about customers. A development team typically deploys an MVP to the market to test a new idea, to collect data about it, and thereby learn from it. (ii) An MMF is the smallest piece of functionality that can be delivered that has value to both the organization delivering it and the people using it. An MMF is a part of an MMR or MMP. (iii) An MMR is the release of a product that has the smallest possible feature set that addresses the customers' current needs. (iv) An MMP is the first deployment of a MMR.

Design Thinking addresses understanding the customer need through systematic exploration. The objective is to understand the right product to develop. This approach encompasses "Empathize", "Define", "Ideate", "Prototype" and "Test" phases.

[6] https://www.facebook.com/notes/kent-beck/the-product-development-triathlon/1215075478525314/

BDD is an agile practice that consists in defining increments of software behavior and their delivery. Similar to BDD, Test Driven Development (TDD), Acceptance Test Driven Development (ATDD) and Specification by Example (SBE) use testable scenarios as input for the software development. All these have in common to start by defining development based in scenarios and use the "Given, When, Then" (gherkin language).

DDD is an approach that proposes the division of concepts by domains, or sub-domains, if applicable. BizDev is continuous linking Business Strategy & Planning and Development. 3X relates to Kent Beck's vision of product development phases, explore, expand, extract.

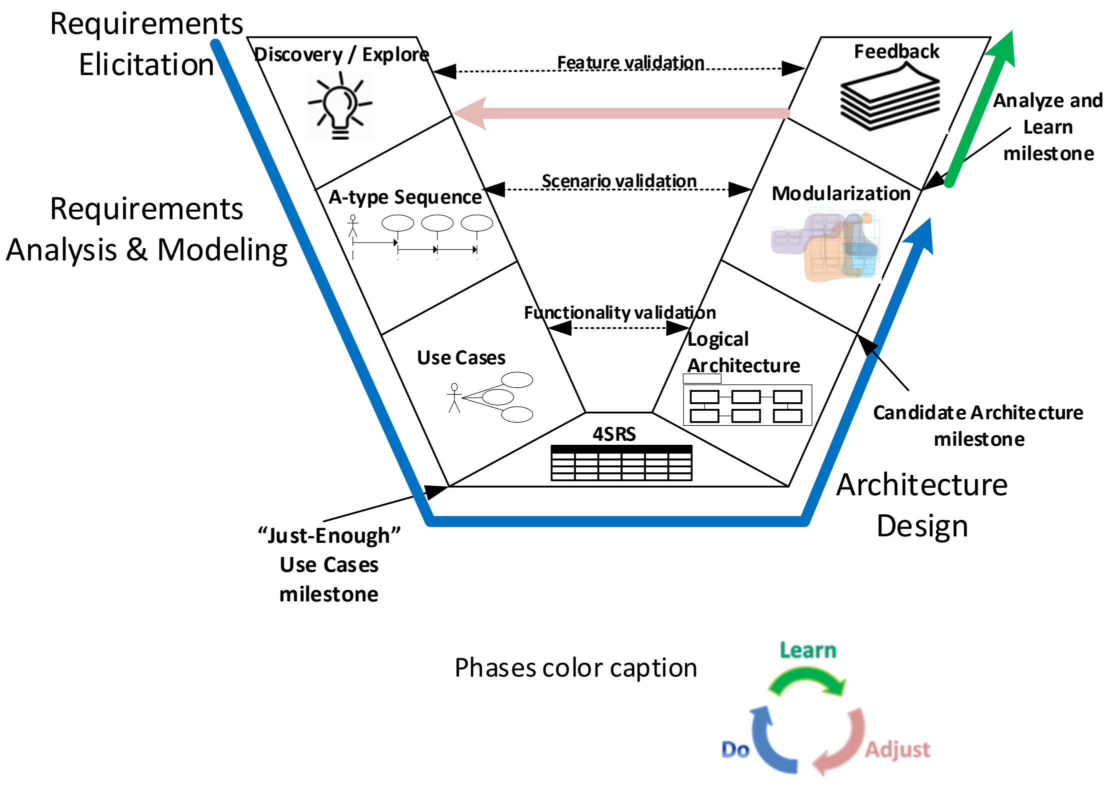


Figure 57. Overview of DUARTE approach

The elicitation and discovery phase of DUARTE relates to eliciting customer needs, exploring alternatives, discover new requirements, all aligned with current agile practices from ASD frameworks, techniques and philosophies. In AMPLA, this phase outputs a set of scenarios, i.e., processes and activities performed using the solution under development. These scenarios are documented in *A-type sequence diagrams*, a stereotyped version of UML sequence diagrams that only include actors and use cases.

At this stage, the use cases included in these diagrams are not yet composing part of the Use Case model from the next stage. Rather, they are classified as candidate use cases, because they relate to specific activities and tasks that a given actor performs (in software or not) within a given scenario. The flows between actors and candidate use cases relate to the actions performed. The use of these diagrams, instead of UML Activity diagrams, BPMN, or any other process-oriented language, relates to the use of candidate use cases, which help to construct the model in the next stage, because the candidate use cases are input for the use case model.

Applying agile practices in this phase influences:

- In **Lean Startup**, stakeholders define scenarios with the experiment mindset in mind. At this point, stakeholders have decided which features to include/experiment in the MVP. The scenarios for such features are elicited with the knowledge to date, where is not the purpose to have detailed technical description of how the solution will support such scenarios, but rather to define the referring processes. The remaining features may be refined afterwards. It is not the purpose of AMPLA to define how to reach minimum features (typically using 'bespoke RE' or 'market-driven RE' techniques), but rather to use the resulting business need as input for scenario modeling.

- In **Design Thinking** and **Kent Beck's 3X**, the customer's desires and expectations are included in the scenarios, but the idea is also to discover and explore scenarios with different solutions and processes rather than only address what customers dictate. *A-type sequence diagrams* represent as many tasks as the scenarios are discovered and explored (Figure 58).

- In **BDD** (or TDD, ATDD or SBE), the requirements discipline is addressed in the discovery and definition of scenarios (in gherkin language format). This format is mapped in *A-type sequence diagrams*, where "Given" contextualize each sequence diagram, "When" relates to a main sequence, or alternatively optional or exception sequence (if existing), and "Then" relate to the flows within the diagram.

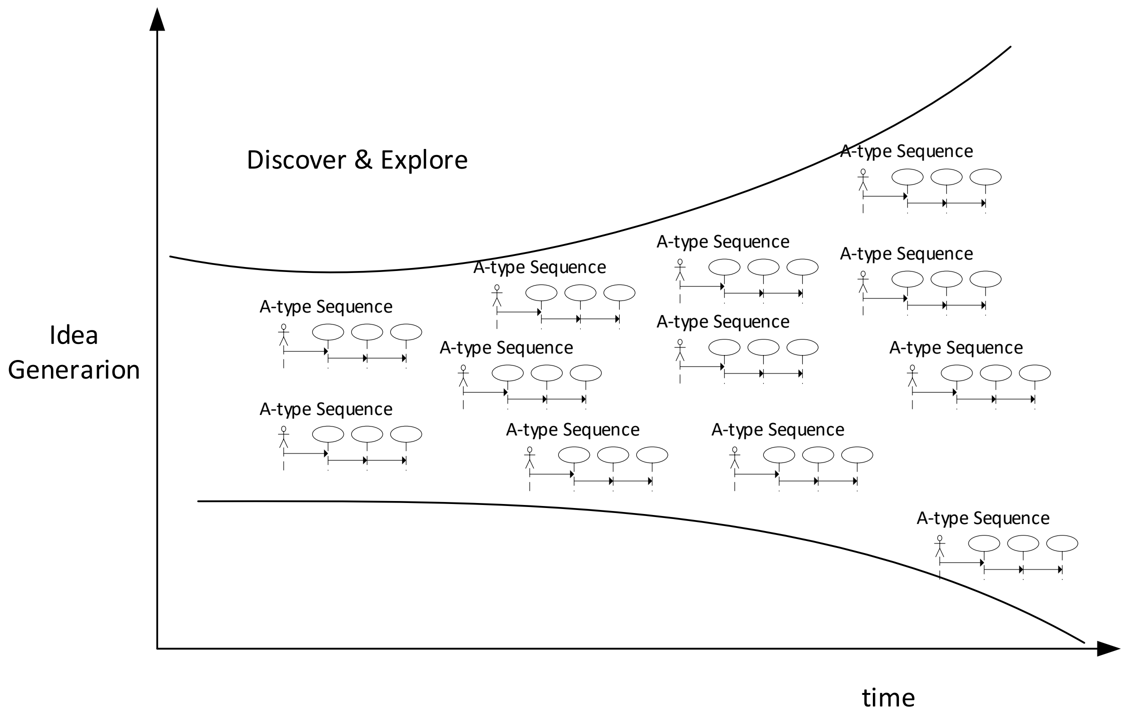


Figure 58. Discovery and exploration of the scenarios

These scenarios, right or wrong (have in mind this is an exploration phase) are modeled in *A-type sequence diagrams*. These diagrams are the first visual prototype where customers are able to provide feedback.

The Requirements Analysis and Modeling phase of DUARTE aims modeling a UML Use Cases diagram. Using as input the elicited scenarios, namely the model artifacts relating to *A-type sequence diagrams*, the Use Cases diagram is built and each Use Case refined. The gathering from the sequence diagrams are based in a set of decisions, which are aligned with agile practices as Design Thinking and DDD.

In this phase, candidate use cases from *A-type sequence diagrams* will give origin to "typical" use cases, i.e., formal software functional requirements. The idea is to use the gathered information and use it to model Use Cases and their refinements. The gathered information allows identifying detailed information about a requirement, which correspond to a use case functionally decomposed in refined use cases. Cruz *et al.* (Cruz, Machado, & Santos, 2014) and Azevedo *et al.* (Azevedo, Machado, Braganca, & Ribeiro, 2010) present such refinement by sub-domains that compose a domain or by splitting a process. They model use case refinement in decomposition trees, and so does this approach. The candidate use cases from *A-type sequence* *diagrams* are grouped in a logical way, typically grouping them to the scenario from *A-type sequence diagrams* that originated them.

Applying agile practices in this phase influences:

- In **Lean Startup**, as customers define which scenarios to include in MVP/MMR/MMP, they are expressed in more detail rather than the scenarios that are left out at this phase. Thus, there is more context to define the models that will compose MMF, which result in more decomposition of those use cases. The remaining requirements that are not refined for the MMF are identified however not afterwards decomposed.
- In **Design Thinking**, Use Cases are used as designed prototypes aiming firsts customer feedbacks
- In **BizDev**, use case models trace back to the scenarios, which support the continuous linking Business Strategy & Planning and Development;
- In **BDD**, the candidate use cases from A-type sequence diagrams are grouped in domains and sub-domains. The refinement "branches" of the decomposition tree hence relate to a single domain or sub-domain, which define bounded contexts for a (sub-)domain. This aspect assures a given team to work on a sub-domain and the independence is assured by the bounded context.

"Just-Enough" modeling

In this section, the objective is to describe the elicitation of the core requirements, and additionally to include techniques for deciding when the "just-enough" requirements model is complete. The elicitation of "just-enough" requirements, rather than promoting their elicitation all upfront, typically faces insufficient knowledge about using recent technologies. Not only there is an inclusion of a new technology, and their unpredictable adoption, but of new business models, processes, and the role of stakeholders within the supply chain.

The business needs, project goals, vision document, and other information that act as inputs for requirements is gathered in the inception phase (or earlier) of the project. In this phase, and namely in agile context, where the requirements are not known upfront, it is very difficult to know in advance how much RE is "just-enough". The vision document, for instance, reflects stakeholder's intentions (*i.e.*, features) towards the entire product, however the main purpose at this time is to assure that such features are included. At this point, stakeholders have decided which features to include in the MVP. The requirements that are object of "just-enough" refinement and modeling relate to such features, while the remaining features from the product roadmap may be refined afterwards.

The DUARTE approach starts by eliciting high-level requirements for the MVP. Key stakeholders are interviewed in order to list a set of their expectations towards the solution (as in "*I expect that the solution is able to do this, and that*...") and their perceived importance, namely to depict the scenarios with highest priority for this release. The interviewees may also identify features of the product roadmap to be included in further releases. The "just-enough" subset should include all identified features, however only the features related to the MVP are object of decomposition. For instance, if a roadmap contains fifty features and only ten are to be implemented in the MVP, the high-level architecture should clearly support those ten, but also include initial support for the upcoming forty features.

This section illustrates the elicitation process based on stakeholders' expectations. However, every business requirements-related information or document that provide inputs for the software requirements elicitation are useful for validating if high-level requirements were considered. Techniques like interviews, questionnaires, workshops, etc., are additional and complemental approaches of the aforementioned document analysis for gathering inputs on requirements. The use of these techniques is a decision of the requirements engineers as they best fit in a given context.

The elicitation process' goal is to model functional requirements in UML Use Case diagrams, promoting the modeling of the product roadmap by functional decomposition, in compliance with a work-breakdown structure (WBS). Use cases are decomposed once or twice, instead of several times like in upfront contexts. Since the main idea is to model "just-enough" requirements, one decomposition may be sufficient, namely the ones that stakeholders are aware at this point. The use of UML Use Case diagrams is mandatory in this approach, since the 4SRS method for deriving the logical architecture requires Use Cases as input.

To validate if the modeled use cases cover the requirements defined in the product scope, **Table 14** exemplifies the crosschecking between the stakeholders' expectations and the project goals. Since the expectations and goals list emphasizes the MVP features, there is a context for MVP features to be more decomposed than the remaining. The premises is that if all these concerns are included in the Use Case model, with more emphasis in decomposition detail to MVP requirements, one may consider that we have “just-enough” requirements information for this stage.

Table 14. Traceability matrix of requirements within the initial expectations

Req.	Expec. 1	Expec. 2	Expec. 3	Expec. n
UC.1	x			
UC.2		x		
UC.3			x	
UC.n	x			x

Like in any requirements process, one of the first critical tasks is to identify all projects stakeholders, as well as the solution’s interacting actors. By mapping stakeholders to the use cases (**Table 15**), one must assure that every stakeholder/actor has at least a requirement mapped to it, or is a symptom that critical requirements are missing.

Table 15. Traceability matrix of requirements within the identified project stakeholders and solution actors

Req.	Stkh A	Stkh B	Stkh C	Stkh D
UC.1	x			
UC.2		x	x	
UC.3				x
UC.n	x			

The mappings from **Table 14** and **Table 15** validates that the UML Use Cases diagram includes the features for MVP but also the product roadmap, and that all stakeholders have related functionalities. Both tables provide the required traceability to the expectations and stakeholder/actor needs, which hence are the mechanism used to respond to changes. When applied to mid-and long term projects, the approach supports updates of expectations and stakeholder/actor needs over time, as well as the inclusion of new Use Cases, by tracing requirements in **Table 14**. Additionally, the approach is able to lead with organizational changes, by tracing the Use Cases to stakeholder/actor needs in **Table 15**. The approach differs from the segmentation of requirements into different priorities by setting an initial set of

Use Cases to the entire solution, refining subsets of the model (section 3.2) and prioritizing them before actually begin the implementation.

Having all the "just-enough" requirements elicited, gathered, modeled and validated, these Use Cases are now able to be used as input for the candidate logical architecture derivation, composed with the "just-enough" architectural components.

4.4. Demonstration cases

Upfront RE for use case-driven product backlogs: the iFloW case

iFloW is an R&D project sponsored by the consortium between University of Minho (UMinho) and Bosch Car Multimedia Portugal (Bosch), that aims at developing an integrated logistics software system for inbound supply chain traceability (cf. Chapter 1). iFloW is a real-time tracking software system of freights in transit from the suppliers to the Bosch plant, located in Braga. The main goal of the project is to develop a tracking platform that by integrating information from freight forwarders and on-vehicle GPS devices allows to control the raw material flow from remote (Asian) and local (European) suppliers to the Bosch's warehouse, alerts users in case of any deviation to the Estimated Time of Arrival (ETA) and anticipates deviations of the delivery time window.

The organization's logistics-related processes and current gaps were documented in a report designated as '*As-Is report*'. Then, the requirements were elicited, formally specified in the form of UML use cases, a list of quality (non-functional) requirements and in a first version of the logical architecture (UML component diagram). This set of requirements was documented in a report designated as '*To-Be report*' (which was constantly updated as the implementation went along). Both use case models (especially the '*To-Be*') were used as basis to define a '*Product Backlog*'. This differs from other agile frameworks where, for instance, in Scrum (Ken Schwaber, 1997), backlogs are composed of user stories. A user story is a customer-centric characterization of a requirement. It contains only the information needed for the project developers to see clearly what is required to implement (Scott Ambler & Lines, 2012). However, use cases are also used in agile frameworks (Jacobson et al., 2011; Kroll & MacIsaac, 2006).

The use case diagram illustrated in Figure 59 shows the overall use case model of the iFloW project. Each of the use cases were functionally decomposed, which resulted in 90 lower level use cases. The use case model is presented in Annex A.

The Initialization phase ends with a Sprint 0 ceremony. Most of the technological research was performed during this ceremony, prior to the implementation in the following Sprints. Like in a typical Sprint 0, each item (use case) was prioritized by its perceived value from stakeholders, in this case by using MoSCoW ("Must", "Should", "Could", "Won't") prioritization technique (Waters, 2009). In addition, each use case was estimated related to a quantitative effort for its implementation. A commonly used technique is use case points (Anda, Dreiem, Sjøberg, & Jørgensen, 2001; Karner, 1993; Nageswaran, 2001), however in this project this technique was not used. Rather, the team of the iFloW project defined that for each Sprint corresponds a total effort of 20 points (which resulted in approximately five points per week) as basis for distribution of these points per use case and following a comparative technique similar to a planning poker (Grenning, 2002). Additionally, each use case was prioritized, and the work was estimated so '*Sprint Backlogs*' (which use cases from the '*Product Backlog*' to implement during the Sprint) could be defined.

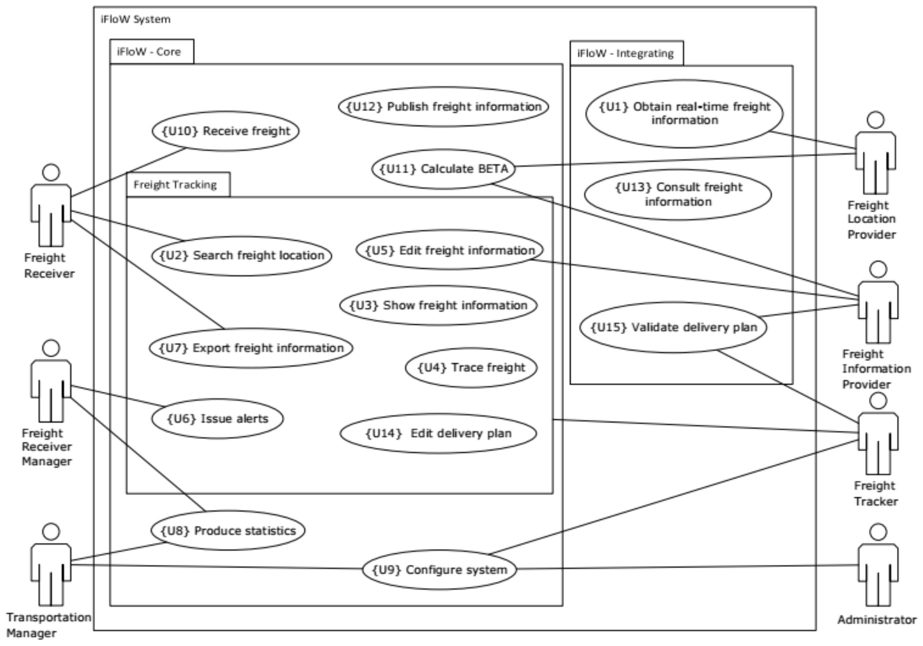
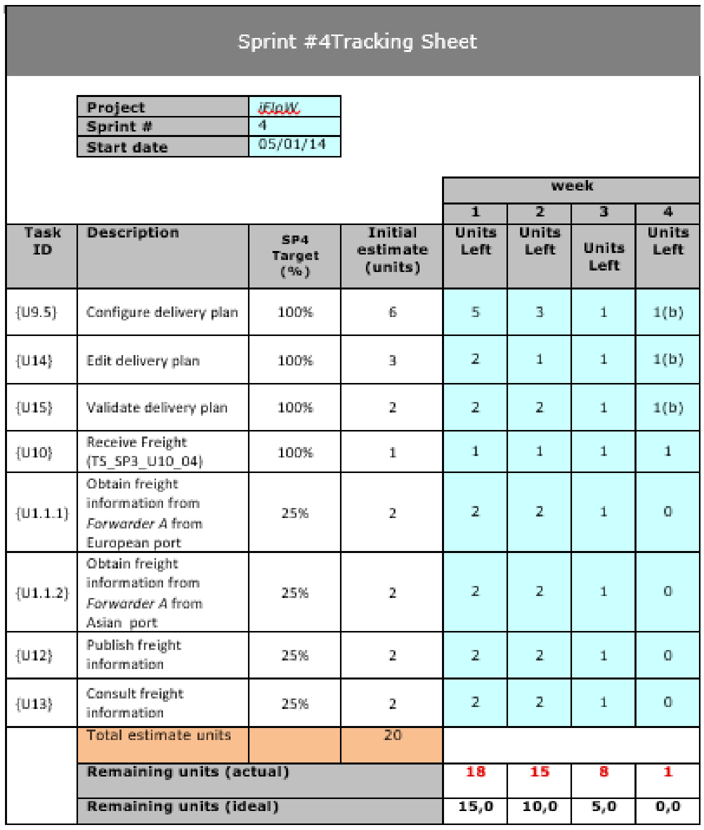

iFloW System
iFloW - Core
iFloW - Integrating
Freight Tracking
{U12} Publish freight information
{U10} Receive freight
{U11} Calculate BETA
{U1} Obtain real-time freight information
{U13} Consult freight information
{U2} Search freight location
{U5} Edit freight information
{U3} Show freight information
{U15} Validate delivery plan
{U7} Export freight information
{U4} Trace freight
{U6} Issue alerts
{U14} Edit delivery plan
{U8} Produce statistics
{U9} Configure system
Freight Receiver
Freight Receiver Manager
Transportation Manager
Freight Location Provider
Freight Information Provider
Freight Tracker
Administrator

Figure 59. Use Case diagram of iFloW project

Within the implementation phase, the use cases from the '*Product Backlog*' were implemented iteratively and incrementally during eight four-week Scrum Sprints. In this phase, typical Scrum iterations were performed, where each '*Sprint Backlog'* is a selected subset from the '*Product Backlog*'. In Figure 60 is depicted an example of a '*Sprint Backlog'* tracking sheet, composed by the iFloW use cases and whose progress was monitored.



Sprint #4Tracking Sheet

Project	iFloW
Sprint #	4
Start date	05/01/14

				week			
				1	2	3	4
Task ID	Description	SP4 Target (%)	Initial estimate (units)	Units Left	Units Left	Units Left	Units Left
{U9.5}	Configure delivery plan	100%	6	5	3	1	1(b)
{U14}	Edit delivery plan	100%	3	2	1	1	1(b)
{U15}	Validate delivery plan	100%	2	2	2	1	1(b)
{U10}	Receive Freight (TS_SP3_U10_04)	100%	1	1	1	1	1
{U1.1.1}	Obtain freight information from *Forwarder A* from European port	25%	2	2	2	1	0
{U1.1.2}	Obtain freight information from *Forwarder A* from Asian port	25%	2	2	2	1	0
{U12}	Publish freight information	25%	2	2	2	1	0
{U13}	Consult freight information	25%	2	2	2	1	0
	Total estimate units		20				
	Remaining units (actual)			18	15	8	1
	Remaining units (ideal)			15,0	10,0	5,0	0,0

Figure 60. Example of a Sprint Backlog based in Use Cases

Each Sprint has a standard planning and structure consisting of several milestones, previously negotiated by the project members:

- **Sprint development**: lasts four weeks, and is allocated to the development of the items from the '*Sprint Backlog*';
- **Sprint Monitoring meeting** takes place in second week to show Sprint progress and monitor Sprint tasks. The attendees are the Product Owner, R&D coordination and development team;
- **Sprint Verification and Validation (V+V) meeting** takes place in the fourth (*i.e.*, last) week and the goal is to test and validate the requirements implemented by the development

team. The attendees are the Product Owner, the development team, a member of the Bosch IT department, and an assigned Product User from Bosch. In each Sprint V+V meeting, the Product User was assigned a different user from Logistics department so the performed tests could encompass different insights from the organization. During the Sprint, if any requirement (use case) is moved to a next Sprint due to a given constraint and will not be presented in this meeting, the team is notified;

- **Sprint Closure and Planning meeting** takes place at most two days after the Sprint V+V meeting, and the attendees are the Product Owner, the R&D coordination, a member of the Bosch IT department and the development team. It is similar to a Sprint Retrospective and a Sprint Planning meeting from typical Scrum, performed within the same meeting. The main goal is to analyze the progress of the implementation phase, by assessing the percentage and completion of the use case implementation and thus updating the burndown chart. If applicable, short rework actions (depicted from the Sprint V+V) are approved to perform until the end of the Sprint. Additionally, the next Sprint is planned, resulting in the construction of the 'Sprint Backlog' artifact;
- **Sprint Rework meeting** takes place the day after the Sprint Closure meeting. After Sprint V+V, some rework actions can arise due to a suggestion by the verification and validation team. If applicable, the development team has to implement these rework actions until the end of the Sprint. The Sprint Rework meetings are used to validate the rework actions performed. The attendees are the assigned Product Users, Product Owner, a member of the Bosch IT department and the development team.

Upfront RE for user stories-driven product backlogs: the ISOFIN Cloud case

The ISOFIN (Interoperability in Financial Software) Cloud was a project where the architecture aimed enacting the coordination of independent services allowing the semantic and application interoperability between enrolled financial institutions (Banks, Insurance Companies and others) (cf. Chapter 1). The global ISOFIN architecture relies on two main service types: Interconnected Business Service (IBS) and Supplier Business Service (SBS). IBSs concern a set of functionalities that are exposed from the ISOFIN core platform to ISOFIN Customers. An IBS interconnects one or more SBS's and/or IBS's exposing functionalities that relate directly to business needs. SBS's are a set of functionalities that are exposed from the ISOFIN Suppliers production infrastructure. Finally, users may also use ISOFIN Applications, which are software applications that result of joining an interface to a single IBS.

Just like in the previous demonstration case, the requirements were identified and modeled upfront. Figure 61 depicts the resulting use case model. All these use cases were refined using functional decomposition, until 80 use cases were modeled. These refined use cases were then used within the 4SRS method towards the logical architecture design.

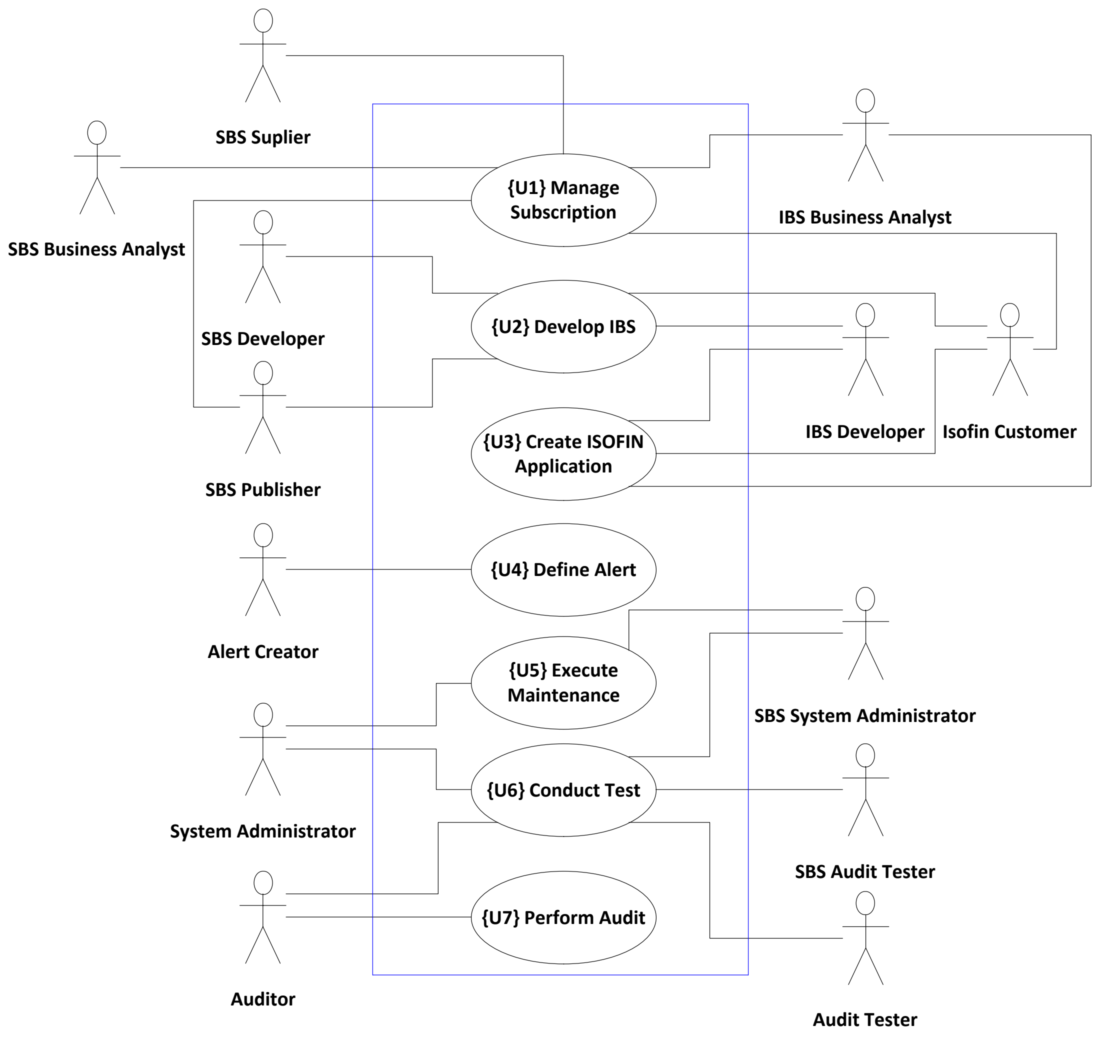


Figure 61. ISOFIN Use Case Model

The ISOFIN logical architecture diagram from performing the 4SRS method is depicted in Figure 62. In the "middle" of the logical architecture diagram is the "Repository" package, containing the several information repositories used in the ISOFIN Platform execution. The rest of the packages reflect the ISOFIN Platform usage, i.e., ISOFIN Applications, IBSs, Subscriptions, Alerts, Logs, Policies and Security Management. All these packages are associated to components regarding the “Repository” package. Finally, also depict the «*generates*» association, that assures a relation between the developed IBSs and ISOFIN Applications and the components that relate to their interface with users (ISOFIN Customer).

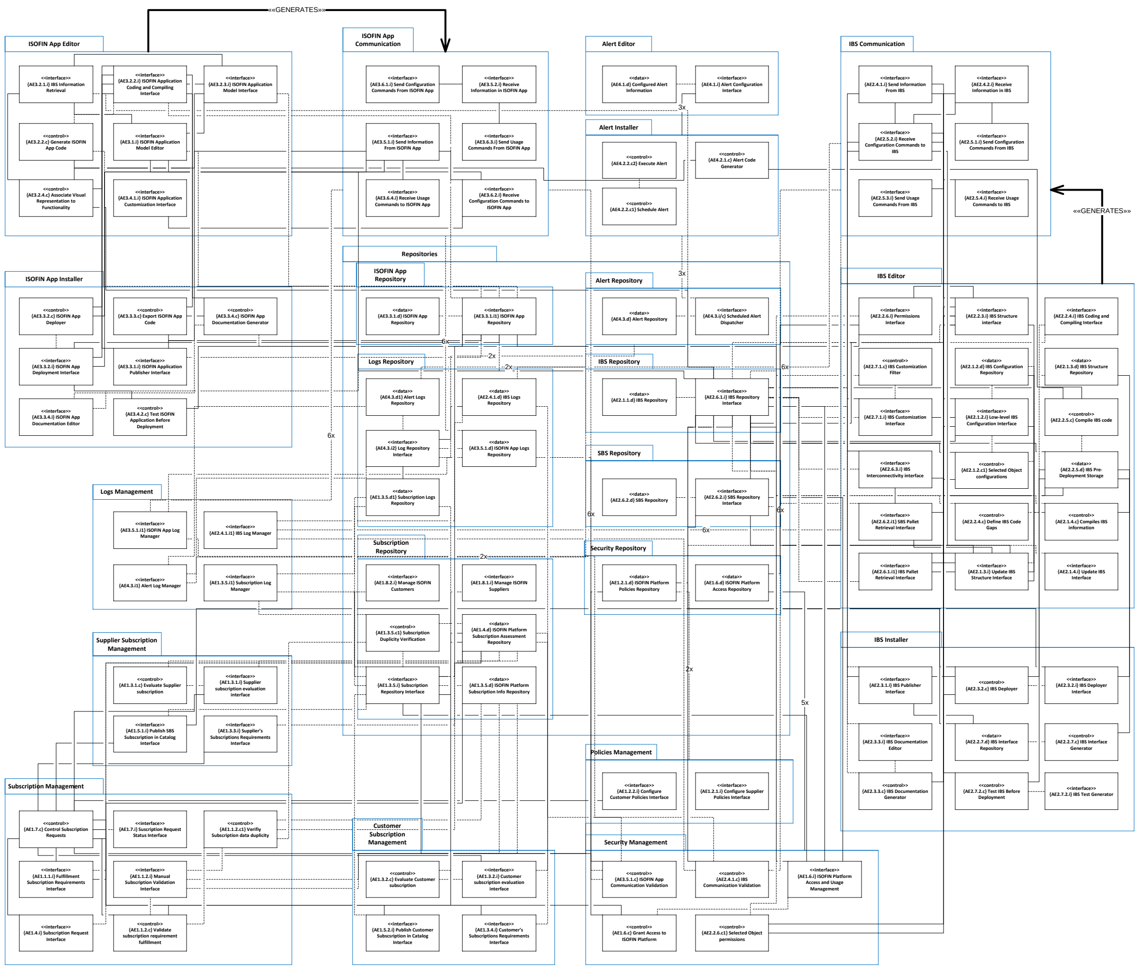


Figure 62. ISOFIN Logical Architecture

Then, the logical diagram was modularized and partitioned in a set of “spots” that traverse the logical architecture, covering the components. These spots represent applications to be developed and that depict independent applications that may be implemented by different teams, due to the complexity presented during the project. Seven applications were identified, which relate to the spots within the modularized architecture in Figure 63, which are: IBS Management Module; ISOFIN App Management Module; Alert Management Module; Subscription Management Module; Security Management Module; Policies Management Module; and Logs Management Module.

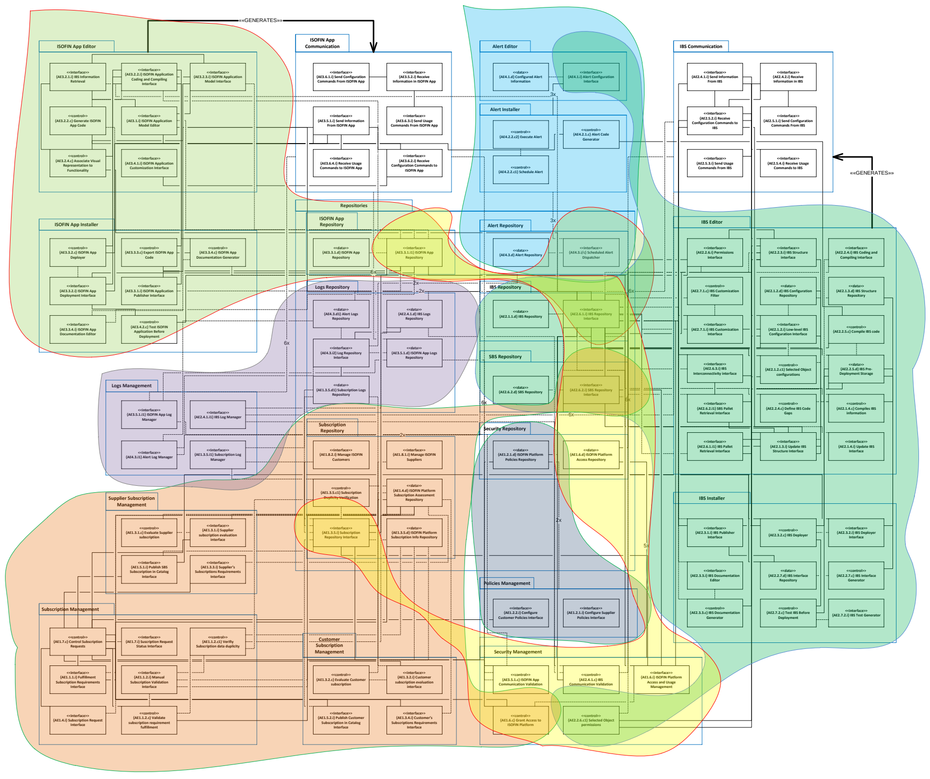


Figure 63. ISOFIN architecture modularization

Using these rules, remaining User Stories were derived that are listed in Table 16. In Figure 64 is represented a User Story sentence based in the derivation from Table 16.

Emerging RE using DUARTE: the UH4SP case

The UH4SP project aims developing a platform for integrating data from distributed industrial unit plants, allowing the use of the production data between plants, suppliers, forwarders and clients (cf. chapter 1). The consortium was composed with five different entities for software development where each had specific expected contributes, from cloud architectures to industrial software services and mobile applications. The entities are geographically distributed, but each entity had a single located team. An analysis team composed with elements from each entities, aiming to define the initial requirements, conducted the requirements phase. Since they belong to different entities, they had to schedule on-site meetings to perform requirements workshops. Only when beginning the software delivery cycles, after boundaries were clear, each team was responsible for refining their requirements.

The requirements elicitation started by listing a set of stakeholder expectations towards the product roadmap, encompassing the entire product but only MVP features were detailed. The expectations list of the project included 25 expectations (Figure 65),

Table 16. User Stories derived from *c-type* components

#	Component	As a(n) <actor>	I want/need <description>	In order to <outcome>
2.1.2.c1	Selected Object Configurations	ISOFIN Customer / IBS Developer	select object configurations	change (IBS Structure) configurations
2.1.4.c	Compiles IBS information	IBS Developer	compile IBS (changes and) information	create a new IBS
2.2.4.c	Define IBS Code Gaps	IBS Developer	(automatically generated code) and define IBS code gaps	create IBS code
2.2.5.c	Compile IBS code	IBS Developer	compile IBS code (and create new IBS catalog)	(keep IBS catalog and store) compile(d) IBS Code
2.2.6.c1	Selected Object Permissions	IBS Developer	select object permissions	set(/manage) permissions (and create IBS)
2.2.7.c	IBS Interface Generator	IBS Developer	(automatically) Generate IBS Interface	(store the) generate(d) IBS interface
2.3.2.c	IBS Deployer	IBS Developer	deploy IBS	execute IBS deployment
2.7.1.c	IBS Customization Filter	Business User	filter IBS (configuration and) customization	customize IBS
2.7.2.c	Test IBS Before deployment	Business User / IBS Developer	test IBS before deployment	render IBS Pre-Runtime

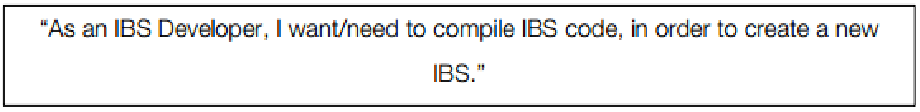
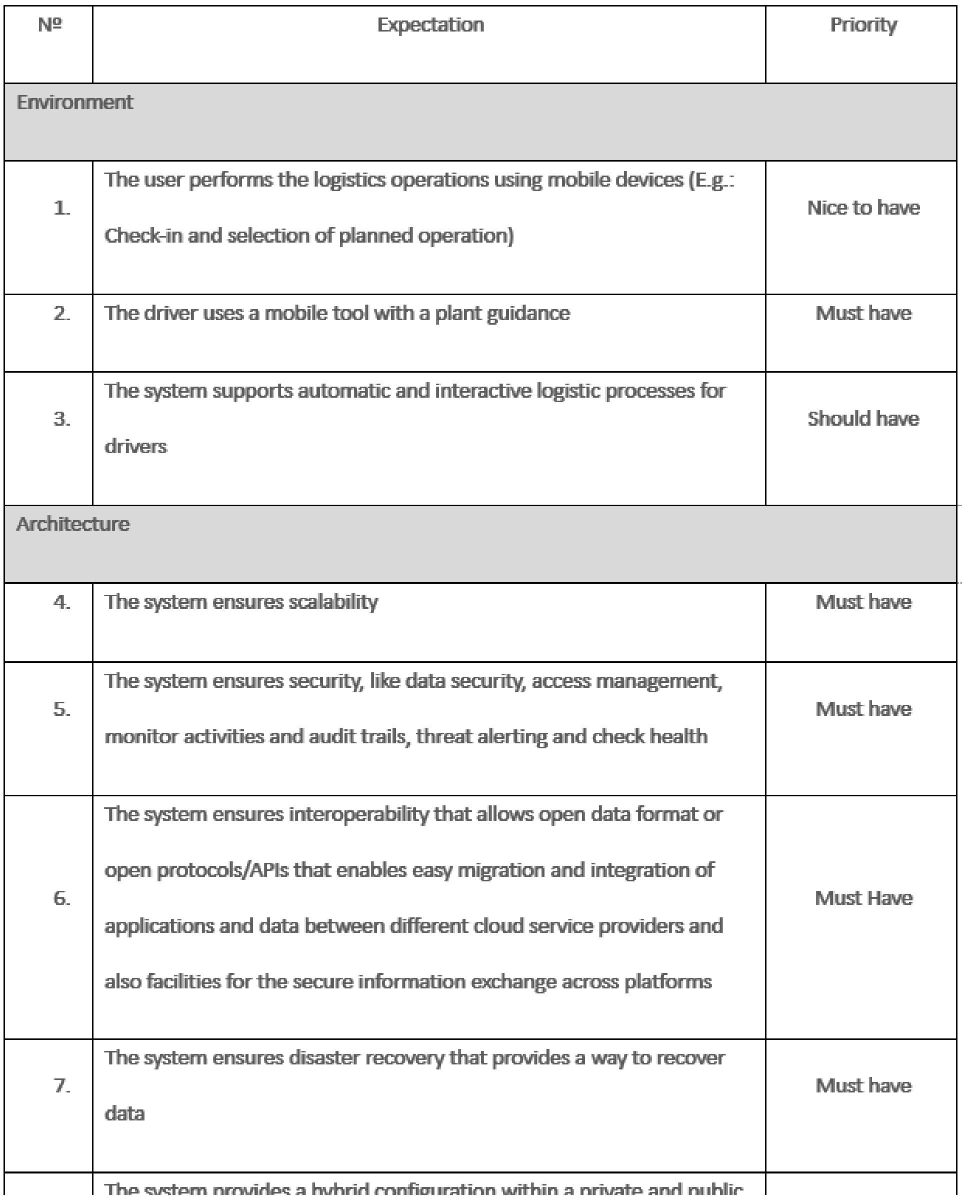

"As an IBS Developer, I want/need to compile IBS code, in order to create a new IBS."

Figure 64. User Story from 2.1.4.c

categorized by environment, architecture, functional and integration issues, which relate to business needs that afterwards promoted the discussion of scenarios (Figure 66). These scenarios were elicited with the customer, but this work additionally aimed exploring and discovering alternatives. The project's objectives that were stated referred to: (1) to define an approach for a unified view at the corporate (group of units) level; (2) to develop tools for third-party entities; (3) in-plant optimization; and (4) system reliability. This task output 15 *A-type sequence diagrams*, divided in four groups of scenarios. These groups relate directly to the project's four objectives.



Nº	Expectation	Priority
Environment		
1.	The user performs the logistics operations using mobile devices (E.g.: Check-in and selection of planned operation)	Nice to have
2.	The driver uses a mobile tool with a plant guidance	Must have
3.	The system supports automatic and interactive logistic processes for drivers	Should have
Architecture		
4.	The system ensures scalability	Must have
5.	The system ensures security, like data security, access management, monitor activities and audit trails, threat alerting and check health	Must have
6.	The system ensures interoperability that allows open data format or open protocols/APIs that enables easy migration and integration of applications and data between different cloud service providers and also facilities for the secure information exchange across platforms	Must Have
7.	The system ensures disaster recovery that provides a way to recover data	Must have
	The system provides a hybrid configuration within a private and public	

Figure 65 . Subset of project initial expectations

Afterwards, the requirements analysis included gathering the candidate use cases and defining the decomposition tree. The Use Case model was composed by 37 use cases after the refinement. They relate to business needs that afterwards allowed depicting functional requirements, modeled in use cases (Figure 67).

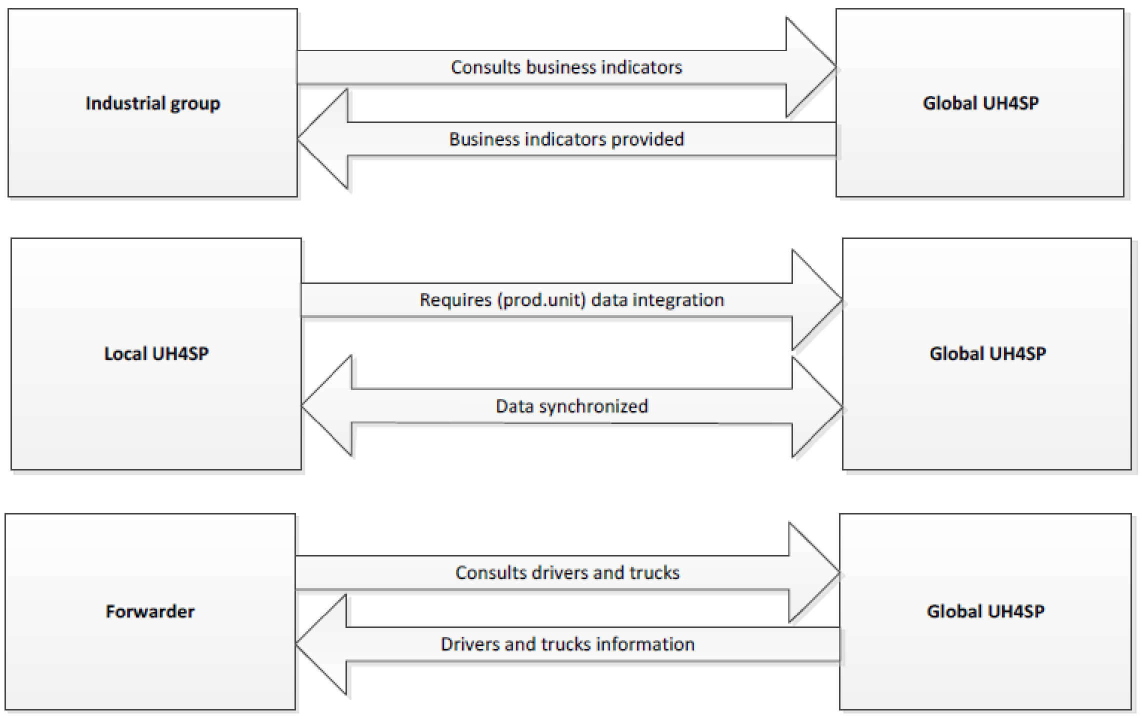

Industrial group
Consults business indicators
Business indicators provided
Global UH4SP
Local UH4SP
Requires (prod.unit) data integration
Data synchronized
Global UH4SP
Forwarder
Consults drivers and trucks
Drivers and trucks information
Global UH4SP

Figure 66. Scenarios elicited

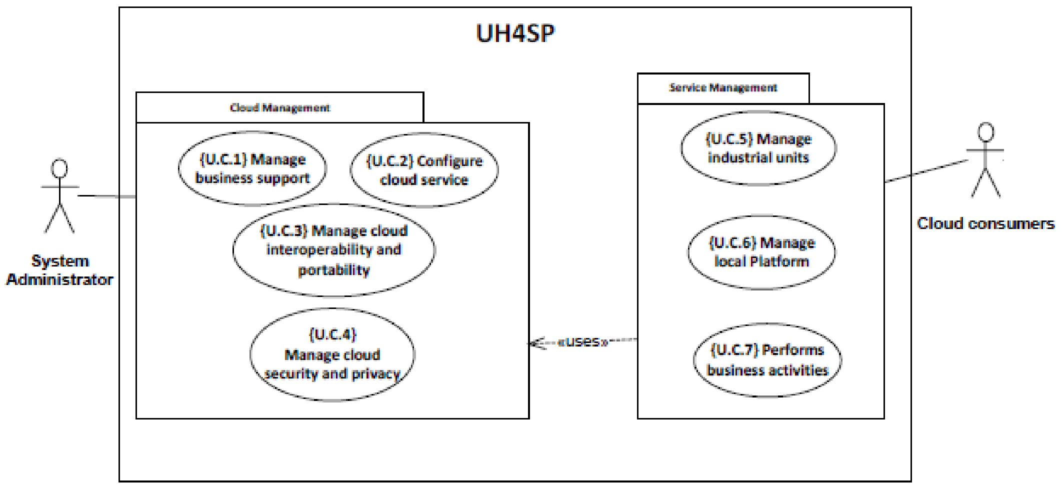

UH4SP
Cloud Management
{U.C.1} Manage business support
{U.C.2} Configure cloud service
{U.C.3} Manage cloud interoperability and portability
{U.C.4} Manage cloud security and privacy
Service Management
{U.C.5} Manage industrial units
{U.C.6} Manage local Platform
{U.C.7} Performs business activities
«uses»
System Administrator
Cloud consumers

Figure 67. UH4SP first-level Use Cases

The Use Case model was globally composed by 37 use cases after the decomposition (Annex C): Use case *{UC.1} Manage business support* was decomposed in five use cases, use case *{UC.2} Configure cloud service* was decomposed in eight use cases, use case *{UC.3} Manage cloud interoperability and portability* was decomposed in five use cases, use case *{UC.4} Manage cloud security and privacy* was decomposed in three use cases, use case *{UC.5} Manage industrial units* was decomposed in two use cases, use case *{UC.6} Manage local Platform* was decomposed in five use cases, and use case *{UC.7} Performs business activities* was decomposed in ten use cases.

Almost the entire model was detailed in one lower-level (*e.g.*, {UC5.1}, {UC5.2}, etc.). Only the cases of *{UC.1} Manage business support*, *{UC.2} Configure cloud service* and *{UC.7} Performs business activities* included an additional decomposition, composed with three use cases each, and are examples of bigger sized features of the MVP (based in the quantity of low-level use cases). Use cases *{UC.3} Manage cloud interoperability and portability* and *{UC.4} Manage cloud security and privacy* relate to features not addressed in the MVP, hence were not object of further decomposition.

The total of 37 use cases perceive the low effort in decomposing at this phase, taking into account the large-scale nature of the project, namely the number of expectations (25) and that it is to be implemented by five separate teams.

The impacts of these agile techniques in modeling the use cases:

- **By applying DDD**, Use Cases are grouped by the domains and sub-domains. This means that each of the tree's "branches" relate only to a given domain, which also assures that the contexts are properly bounded. The identified domains relate directly to the four scenario groups. Two of them, "tools for third-party entities" and "system reliability", was afterwards divided in two and three domains, respectively, hence making a total of seven. Two of the "system reliability" bounded context are not even depicted due to MVP decisions.
- **By applying Lean Startup**, the features defined to be included in the MVP are identified in the model by having refined use cases, while the remaining were just identified in the first-level. Use cases *{UC.3}* and *{UC.4}* relate to features not addressed in the MVP, hence were not object of further decomposition. The remaining use cases were included in the MVP, where, namely, *{UC.1}* was decomposed in five use cases, *{UC.2}* was decomposed in eight use cases, *{UC.5}* was decomposed in two use cases, *{UC.6}* was decomposed in five use cases, and *{UC.7}* was decomposed in ten use cases.

The UH4SP logical architecture had as input 37 use cases and, after executing 4SRS method (Annex C), was derived with 77 architectural components (Annex C) that compose it. The logical architecture is discussed in Chapter 5.

Discussion

Use case-driven Product Backlog from upfront requirements modeling

Defining a hybrid approach (waterfall-based during initialization and Scrum-based during implementation), with the inclusion of artifacts modeling and documentation, strengthened the adoption of a Scrum process in a context as the one presented within the iFloW project. However, the entire adoption was a learning process, with advantages and disadvantages, which are detailed in this section.

This demonstration case showed the following advantages:

Requirements documentation waterfall-based – the fact that the Product Backlog was composed of 90 use cases led to a shared perception of the system complexity that originated the need to perform proper efforts in documenting the requirements. Thus, consuming efforts in almost exclusively for requirements engineering typically performed in waterfall approaches, in the initialization phase, allowed the project team to gain the required knowledge to implement a system of such complexity.

Implementation Scrum-based – within a customer perspective, Bosch was always aware of the system's current state of development. The iterative development, in form of Scrum Sprints, was crucial to manage Bosch's expectations, due to the periodical meetings and the incremental delivery of working software.

Use of a logical architecture – to enforce a proper organization on the set of components. The relationships among components suggest dependencies that may affect the implementation (see Section 6.5).

On the other hand, it also showed the following disadvantages:

Effort estimation for use cases - the fact that it was a completely new development team, estimating the required effort for implementing use cases by comparing with other was itself a learning process. In many Sprints there were use cases not implemented due to error in estimating and required almost constant updates on effort estimating (see Section 6.5).

User story-driven Product Backlog from upfront requirements modeling

The upfront requirements modeling and logical architecture design, by using an architectural method like the 4SRS, provided information regarding the 'who?', 'what?', and 'why?' on modules' software requirements, which is the required information to be delivered to Scrum teams, for instance. The purpose is to provide information regarding software functionalities, regardless of the implementation techniques and technologies (*e.g.*, programming languages) the team uses. This means that functional requirements were elicited, but technical decisions have to be made by implementation teams.

Modularizing the architecture provides information about software functionalities of the module, interfaces with other modules, and the context of the module's usage. Decisions regarding messaging, protocols, amongst others, are made by the implementation team (or by the person responsible, like a project manager or a product owner). Nevertheless, the models provide functional and behavioral information of the module in design that will later support the technical specifications of the modules.

These models allow deriving software requirements compliant with Scrum teams, in the form of User Stories. The information from the 4SRS method execution regarding the components specification and the original use case model is gathered in order to derive the User Stories. The approach also allow identifying some contact points where there is a need for synchronizing efforts within distributed Scrums and effort dependencies. User Stories also properly cover these points. Since User Stories are not exclusive for the Scrum framework, this approach can be used in other ASD contexts besides Scrum.

Overall, in the ISOFIN project, there were clear advantages in using this approach:

(1) since the project consortium was composed by Scrum teams, they easily understood the artifacts (*i.e.*, User Stories);

(2) User Stories were derived having an already designed logical architecture as input, allowing them to be properly aligned within reduced time.

Connection points between modules were identified and properly covered by User Stories but there was not enough time during this research work to assess that the team's efforts were in fact synched. Besides the identification of connection points, the authors believe that there is a vast area of progress in the topic of distributed Scrum teams.

DUARTE approach within AMPLA

Applying DUARTE affected use case modeling with the following advantages:

- Overall, the use of agile practices per se did not make the process agile, but allowed specifying agilely the right product for the customer's needs. Delivering the product right is promoted by using Scrum, Kanban, XP, SAFe or LeSS, for instance, which is also present in AMPLA.

- Promoting scenarios discovery and exploration allowed defining 15 scenarios from four groups (from the project's objectives). Without the exploring, each groups would probably include one or two scenarios.
- Defining bounded contexts using DDD allowed to clearly understanding boundaries between what requirements different teams could address. The use of the Lean Startup strategy allowed to refine only the use cases from the MVP hypothesis that the project aim validating, rather than refining all use cases, even those that would not been included in the MVP.
- These practices ease customer feedback, which is fundamental in any ASD process.

However, AMPLA proposes additional activities and artifacts in agile RE methods and may require having dedicated teams for modeling, which may be perceived as a disadvantage. The main threat to validity is that AMPLA was only applied by the method's designers.

AMPLA provides a method for deriving a candidate logical architecture based in UML Use Cases, the 4SRS method. The approach also validated the coverage of the elicited "just-enough" model, gathered together with identified key stakeholders. The "just-enough" UML Use Case model allows to early identify main features, which provides an overview of the project and its scope. We acknowledge the impact of features' characteristics (size, complexity, interconnections, dependencies between subsystems etc.) to management issues, as tasks planning, budget proposals, and resource and skills allocation, which will be addressed further.

The design of "just-enough" architecture used an architectural method, the 4SRS, to derive the candidate architecture based in the small set of ("just-enough") UML Use Cases. There is not any difference within the steps of the method, in comparison with the original method, to derive a candidate architecture. Rather, as the input are high-level requirements in opposition to more refined ones, one may experience difficulties in identifying a proper classification of the use case in order to decide the components to be maintained within the second step, since a more refined information helps in better define the component's nature. However, as in AMPLA the requirements will be later refined and will emerge, the 4SRS method is used as in a "living table" that is opened alongside the development Sprints, rather than a waterfall-based and one-time-execution approach, providing traceability between the requirements and the components in order to agilely respond to changes.

4.5. Conclusions

This chapter discusses requirements modeling approaches in ASD settings, where research addressed initial artifact proposal regarding processes of upfront requirements elicitation and modeling, and a later artifact proposal for emerging requirements (DUARTE). Upfront requirements was used towards use case-driven and user story-driven Product Backlogs. DUARTE was used towards user story-driven Product Backlogs based in AMPLA.

While use case-driven Product Backlogs used directly the UML Use Cases from the requirements modeling as backlog items, the inputs for being able to define user story-driven Product Backlogs required a set of models, namely UML Use Cases and logical architecture (UML Components), supported by architectural method 4SRS and V-Model for traceability purposes.

AMPLA is a process for model derivation applicable in an Agile RE and AM context. Firstly, more specifically regarding Agile RE, this section presented the DUARTE approach, which is inspired by a V-Model approach (Nuno Ferreira et al., 2014) based in successive model derivation, namely referring to sequence, use case and components diagrams, this process aims modeling the same artifacts however using agile practices such as Lean Startup, Design Thinking, DDD, BDD, and others. The model derivation follows typical agile feedback loops, encompassing discovery and exploration, learning from feedbacks and adjusting posterior loops. It also addresses AM so requirements emerge from these loops, by including only core and high-level requirements in early phase of projects, use them for deriving a UML components diagram using the 4SRS method, and further incremental refinements within development cycles (e.g., Scrum Sprints).

This chapter essentially described the:

- A **hybrid process** with upfront modeling and use case-driven Product Backlog, when requirements are known upfront and are stable, followed by ASD cycles (Sprints);
- The DUARTE approach for requirements modeling, which demonstrated how requirements models like Sequence and Use Cases diagrams are affected when using agile practices such as Lean Startup, Design Thinking, Domain-driven Design (DDD), Behavior-driven Development (BDD), and others;

- A framework for assessing "just-enough" modeled requirements in order to design a candidate architecture.

The resulting UML use case model is now ready for use in the logical architecture design. In Chapter 5, the architecture evolves by performing "just-enough" design, which is derived by using the "just-enough" requirements within an execution of the 4SRS method.

Further Reading

About RE in agile, or "Agile RE", "The Agile Extension to the BABoK Guide" (IIBA, 2017), "RE@Agile" (IREB, 2018) and "Beyond Requirements" (McDonald, 2015) focus on performing RE techniques in ASD settings. These techniques are particularly helpful when balancing between product discovery and delivery, as in performing "Dual-track agile"[7] or a "Mobius Loop"[8].

Dean Leffingwell presents the associations between the types of requirements information in product backlogs in the book "Agile Software Requirements" (Leffingwell, 2010).

The process of developing models in a iterative, incremental and evolutionary fashion is presented in "Agile Modeling" (S Ambler, 2002).

The traceability of requirements to software delivery is presented by "User Story Mapping" technique (Patton & Economy, 2014)

References

Agile Alliance. (2001). Manifesto for agile software development.

Ambler, S. (2002). *Agile modeling: effective practices for extreme programming and the unified process*. John Wiley & Sons, Inc.

Ambler, S. (2003). Agile model driven development is good enough. *IEEE Software*, *20*(5), 71–73. https://doi.org/10.1109/MS.2003.1231156

Ambler, S. (2005). The agile unified process (aup). *Ambysoft*.

Ambler, S., & Lines, M. (2012). *Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise*. IBM Press.

[7] https://www.productplan.com/glossary/dual-track-agile
[8] https://mobiusloop.com/

Anda, B., Dreiem, H., Sjøberg, D. I. K., & Jørgensen, M. (2001). Estimating software development effort based on use cases—experiences from industry. In *≪ UML≫ 2001—The Unified Modeling Language. Modeling Languages, Concepts, and Tools* (pp. 487–502). Springer.

Azevedo, S., Machado, R. J., Braganca, A., & Ribeiro, H. (2010). The UML «include» relationship and the functional refinement of use cases. In *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on* (pp. 156–163). IEEE.

Brown, T. (2009). *Change by Design: How Design Thinking Transforms Organizations and Inspires Innovation*. New York: Harper Collins.

Cao, L., & Ramesh, B. (2008). Agile requirements engineering practices: An empirical study. *IEEE Software*, *25*(1), 60–67. https://doi.org/10.1109/MS.2008.1

Cho, J. (2009). A hybrid software development method for large-scale projects: rational unified process with scrum. *Issues in Information Systems*, *10*(2).

Cohn, M. (2004a). Advantages of user stories for requirements. *InformIT Network*.

Cohn, M. (2004b). *User stories applied: For agile software development*. Addison-Wesley Professional.

Cruz, E. F., Machado, R. J., & Santos, M. Y. (2014). On the Decomposition of Use Cases for the Refinement of Software Requirements. In *2014 14th International Conference on Computational Science and Its Applications* (pp. 237–240). IEEE. https://doi.org/10.1109/ICCSA.2014.54

Durdik, Z. (2011). Towards a process for architectural modelling in agile software development. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS - QoSA-ISARCS '11* (p. 183). New York, New York, USA: ACM Press. https://doi.org/10.1145/2000259.2000291

Evans, E. (2004). *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley.

Ferreira, N., Santos, N., Machado, R., Fernandes, J. E., & Gasević, D. (2014). A V-Model

Approach for Business Process Requirements Elicitation in Cloud Design. In A. Bouguettaya, Q. Z. Sheng, & F. Daniel (Eds.), *Advanced Web Services* (pp. 551–578). Springer New York. https://doi.org/10.1007/978-1-4614-7535-4_23

Ferreira, N., Santos, N., Machado, R. J., & Gasevic, D. (2012). Derivation of Process-Oriented Logical Architectures: An Elicitation Approach for Cloud Design. (A. J. O. Dieste and N. Juristo, Ed.), *13th International Conference on Product-Focused Software Development and Process Improvement - PROFES 2012*. Madrid, Spain: Springer-Verlag, Berlin Heidelberg, Germany .

Ferreira, N., Santos, N., Machado, R. J., & Gašević, D. (2013). *Aligning domain-related models for creating context for software product design*. *Lecture Notes in Business Information Processing* (Vol. 133 LNBIP). https://doi.org/10.1007/978-3-642-35702-2_11

Ferreira, N., Santos, N., Soares, P., Machado, R., & Gašević, D. (2013). A Demonstration Case on Steps and Rules for the Transition from Process-Level to Software Logical Architectures in Enterprise Models. In J. Grabis, M. Kirikova, J. Zdravkovic, & J. Stirna (Eds.), *The Practice of Enterprise Modeling* (Vol. 165, pp. 277–291). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-41641-5_20

Ferreira, N., Santos, N., Soares, P., Machado, R. J., & Gasevic, D. (2012). Transition from Process- to Product-level Perspective for Business Software. *6th International Conference on Research and Practical Issues of Enterprise Information Systems (CONFENIS'12)*. Ghent, Belgium.

Fitzgerald, B., & Stol, K.-J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, *123*, 176–189. https://doi.org/10.1016/J.JSS.2015.06.063

Grau, B. R., & Lauenroth, K. (2014). Requirements engineering and agile development - collaborative , just enough , just in time , sustainable. *International Requirements Engineering Board (IREB)*.

Grenning, J. (2002). Planning poker or how to avoid analysis paralysis while release planning. *Hawthorn Woods: Renaissance Software Consulting*, *3*.

IIBA. (2017). *Agile Extension to the BABOK Guide v2*. International Institute of Business Analysis.

IREB. (2018). *IREB Certified Professional for Requirements Engineering - Advanced Level RE@Agile*.

Jacobson, I., Spence, I., & Bittner, K. (2011). *Use case 2.0: The Definite Guide*. Ivar Jacobson International.

Karner, G. (1993). Resource estimation for objectory projects. *Objective Systems SF AB*, *17*.

Kroll, P., & MacIsaac, B. (2006). *Agility and discipline made easy: Practices from OpenUP and RUP*. Pearson Education.

Leffingwell, D. (2010). *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. Addison Wesley Longman.

McDonald, K. J. (2015). *Beyond Requirements: Analysis with an Agile Mindset (1st Edition)* (Agile Soft). Addison-Wesley Professional.

Nageswaran, S. (2001). Test effort estimation using use case points. In *Quality Week* (pp. 1–6).

Paetsch, F., Eberlein, A., & Maurer, F. (2003). Requirements engineering and agile software development. In *Proceedings of Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2003)*. IEEE.

Patton, J., & Economy, P. (2014). *User Story Mapping: Discover the Whole Story, Build the Right Product*. O'Reilly.

Ries, E. (2011). *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*. Crown Books.

Santos, N., Ferreira, N., & Machado, R. J. (2017). Transition from Information Systems to Service-Oriented Logical Architectures: Formalizing Steps and Rules with QVT. In M. Ramachandran & Z. Mahmood (Eds.), *Requirements Engineering for Service and Cloud Computing* (pp. 247–270). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-51310-2_11

Schwaber, K. (1997). Scrum development process. In *Business Object Design and Implementation* (pp. 117–134). Springer. https://doi.org/10.1007/978-1-4471-0947-1_11

Schwaber, K., & Beedle, M. (2001). *Agile Software Development with Scrum*. Upper Saddle

River: Prentice Hall.

Smart, J. F. (2015). *BDD in Action: Behavior-driven development for the whole software lifecycle*. Manning.

Waters, K. (2009). Prioritization using moscow. *Agile Planning*.

Chapter 5 – Agile Logical Architecting using AMPLA

This chapter relates to agile architecting approaches within AMPLA, using the requirements models from the previous chapter. It starts by proposing an agile architecting lifecycle, from business to deployment, supported by a pathway composed by multiple architecture viewpoints. As part of the pathway, the chapter firstly derives a candidate logical architecture using the 4SRS method. Afterwards, the models are incrementally refined throughout agile iterations. This chapter also includes a continuous architecture management by embracing changes using a change-impact analysis approach. The models are refined until they are at a service-level, in form of a microservices logical architecture, whose derivation was supported by the 4SRS-MSLA method. Architectural design during Sprints is demonstrated in the iFloW case. The AMPLA entire lifecycle is demonstrated in the UH4SP project. Finally, the IMP_4.0 and the ISMPM cases relate to the usage of the 4SRS-MSLA in brownfield projects. This chapter ends with the conclusions.

Chapter 5 – Agile Logical Architecting using AMPLA

"You should use iterative development only
on projects that you want to succeed."
***Martin Fowler**, Author of the book "Refactoring"*

5.1. Introduction

In previous section, a logical architecture derived from requirements was used in ASD methods so an agile architecting approach (AMPLA) was followed. This approach proposed modeling firstly a candidate version of the logical architecture, followed by incremental refinement as software emerged during delivery cycles. This section discusses architectural methods within delivery cycles phase of AMPLA.

ASD approaches are characterized from their frequent customer involvement, and well-known frameworks, such as Scrum and XP have specific events where incremental prototypes are shown and validated by the customer. It is during these events that eventual changes are proposed. However, changes in features and/or architecture may lead to unexpected consequences if not properly and previously analyzed. Thus, architecting includes analyzing dependencies, constraints, risks, etc. which are impacted by the changes (Pérez, Díaz, Garbajosa, & Yagüe, 2014). Changes in the software, often called refactoring (Fowler, 2018), are adopted in ASD as a way to continuous improve the structure and understandability of the source code during development (Moser, Abrahamsson, Pedrycz, Sillitti, & Succi, 2008).

In order to manage such changes, changes – or refactoring – are able to be traced to architectural models as well as requirements and business needs. Models are then a way to provide stakeholders with affected changes before the code is refactored. AMPLA supports the required traceability, where the 4SRS method support mapping between the requirements models (using UML use cases) and architectural model composed by UML components.

This chapter discusses the *agile logical architecting* topic addressed within the AMPLA process (Figure 49). In terms of the integrated modeling roadmap, this chapter encompasses stages 3, 4, 6 and 7.

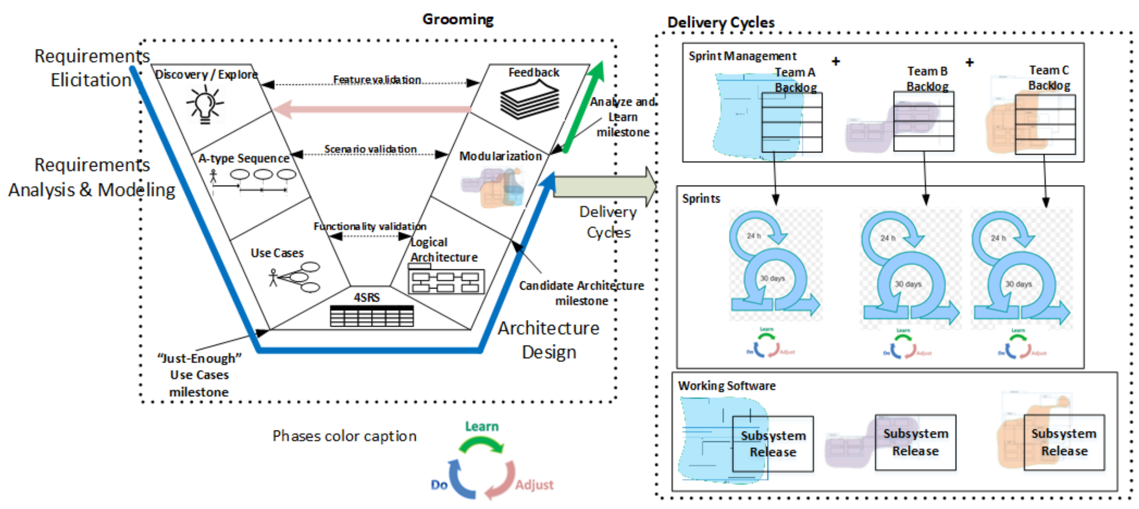


Figure 68. Overview of AMPLA

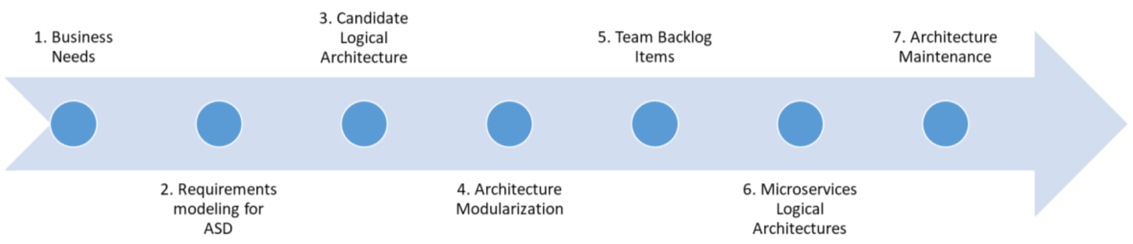

1. Business Needs
2. Requirements modeling for ASD
3. Candidate Logical Architecture
4. Architecture Modularization
5. Team Backlog Items
6. Microservices Logical Architectures
7. Architecture Maintenance

Figure 69. Integrated modeling roadmap

Previous works about *agile architecting* show that architecture emerges throughout the project. This chapter proposes the design of a candidate version of the architecture. The candidate version is a result of performing the 4SRS method with the requirements models resulting from applying DUARTE approach within AMPLA. Alongside with DUARTE, some design decisions are made focusing in concerns more related to domain and information systems context.

These delivery cycles then start with the scope refinement of each area, performed continuously within the iterations as needed. The architecture is thus refined in an incremental way, allowing it to emerge as iterations occur. During these iterations, when stakeholders review current software development status, *e.g.*, at a Sprint Review event, they analyze the delivered increment, accept it as it stands, or propose changes and/or new requirements. Before moving on towards these changes, there is a need to analyze its impact on the solution architecture. For this reason, there was a need to propose within AMPLA a need for using its traceability capability, promoted by the 4SRS method, between requirements and architecture, towards managing this impact as well as manage any technical debts.

However, agile architecting, as any architecting, includes multiple viewpoints. This research focuses in the *logical* one. Logical architecting is about addressing how components are organized so the software meets the business needs. The 4SRS method, historically, uses UML use case models to derive logical architectures. For agile settings, there must be a concern of supporting the logical architecture to emerge throughout Sprints but, in order to support the continuous flow, the relationship with other viewpoints and how they emerge is also a research point.

The software product delivery is a continuous process. The need advocated by ASD approaches on getting feedback, learning and adapting - towards “delivering the right product”, supported by approaches like “Lean Startup” (Ries, 2011) and “Lean Six Sigma” (George & George, 2003) – has put software companies adopting a “continuous” agenda (Fitzgerald & Stol, 2017). Within this agenda, practices as continuous integration, continuous delivery and continuous deployment play an important role. Also, the DevOps (Loukides, 2012) culture has brought the “Ops” concern early within the software process, rather than only in pre-production stage.

As a result, the architecting discipline must also encompass “dev” and “ops” concerns, which are overlapped rather than in separated stages. The multiple viewpoints are thus an aspect to consider continuously. Erder and Pureur propose some principles for continuous architecting, including leveraging the “power of small” which eases an incremental architecture management (Erder & Pureur, 2015).

In this sense, microservices architectures (MSA) (Newman, 2015) is one of the most common situations when companies adopt continuous architecting processes (Taibi, Lenarduzzi, & Pahl, 2017), based in patterns such as Domain-driven Design (DDD) (Evans, 2004), single responsibility principle (SRP) or Conway’s Law (Conway, 1968), which assure they are bounded so that they can scale independently.

MSA are an architectural style oriented towards modularization, where the idea is to split the application into smaller, interconnected services, running as a separate process that can be independently deployed, scaled and tested (Thönes, 2015). MSA are currently getting widespread attention as they extend the ‘design-stage architecture’ into deployment and operations as a continuous development style (Pahl & Jamshidi, 2016).

However, projects often struggle to properly bound them, resulting in insufficient knowledge for decisions related to database partition, the proper size of the microservice, inter-service communication and messaging, which are not addressed systematically by those patterns. By applying a modeling method in the process of designing a MSA, one may foresee issues on bounded contexts for microservices, namely intra-service behavior, interfaces and data models separation, and inter-service communication and messaging requirements (Newman, 2015).

This chapter describes an approach for designing a microservices-oriented logical architecture (MSLA), *i.e.*, a logical view (Kruchten, 1995) on the behavior of microservices and relationships between microservices. This approach uses UML use cases diagrams for domain modeling, which are further used as an input for designing a MSLA using a set of rule-based decisions, by using an adaptation of the 4SRS method. Each of these functionally decomposed UML use cases give origin to one or more components, which will then compose the microservices.

The research addressed in this chapter is the result of a set of agile architecting practices applied throughout different stages of software development, with emphasis to logical architecting. The evolution of agile architecting from grooming to software delivery stages – including initial inputs, candidate architecture design, incremental refinement, continuous architecting and change-impact analysis, and microservice logical architecture design (using 4SRS-MSLA method) and deployment - is presented within the UH4SP project. The specific practice of architecture spikes during Scrum Sprints is discussed within the iFloW project. Finally, regarding microservices logical architectures, aside from the UH4SP project, two research projects are used as complimentary demonstration cases. The IMP_4.0 project used the 4SRS-MSLA method for proposing a MSLA from a requirements modeling of a monolithic solution. The IMSPM used the 4SRS-MSLA but also discussed usage of other microservices patterns. The contributions of the projects are summarized in Table 17.

Table 17. Contribution of projects in continuous architecting

Research contribution \ demonstration case	UH4SP	IMP_4.0	iFloW	IMSPM
Agile logical architecting	X			
Candidate architecture design	X			
Recursive Subsystem design	X			
Architectural Spikes			X	
Change impact analysis	X			
Microservices design	X	X		X
Microservices deployment process				X

This chapter is structured as follows:

- Section 5.2 presents an approach for agile logical architecting;
- Section 5.3 presents logical architecting, through candidate design, incremental refinement and Change-impact analysis techniques within iterations, and the microservices logical architecture design;
- Section 5.4 describes the demonstration cases, applying the aforementioned approaches, and main discussions around agile architecting;
- Section 5.5 presents the chapter's conclusions;
- The chapter ends with complimentary reading

5.2. Agile architecting lifecycle (AAL)

ASD frameworks typically are structured in three phases within the lifecycle: Stories (or Requirements), Planning (of the cycles) and Delivery (of a working software increment). Note that (continuous) integration sometimes fall inside Delivery phase (when continuous integration, continuous delivery and DevOps practices are adopted), otherwise the process includes a Maintenance or Operations phase. For instance, well-known ASD framework as Scrum includes the Stories definition within the Pregame phase (Schwaber, 1997), whereas XP lifecycle includes User Stories definition within the Exploration phase (Beck & Andres, 2004). Planning – namely, the definition of the Product Backlog and its items – is also performed in the Pregame phase of Scrum (Schwaber, 1997) but in XP it is performed in the Planning phase (Beck & Andres, 2004). Delivery of software is performed in the Development phase of Scrum (Schwaber, 1997) and Iterations to Release phase of XP (Beck & Andres, 2004). The Postgame phase of Scrum and

Productionizing phase of Scrum may be included in the Delivery phase of the development lifecycle, or in an afterwards Operations phase.

With that in mind, an agile architecting lifecycle (AAL) should be oriented to these three phases. We propose architecting tasks, artefacts, possible inputs and outputs, for each of the three phases: Grooming, Backlog and Delivery.

AAL should first propose a high-level architecture, composed by main functional requirements and that allowed defining a separation of concerns. This separation is input for planning of each concern implementation, which each relate to a subsystem of the architecture. During delivery cycles (e.g., Scrum Sprints), each subsystem is refined into a more detailed architecture, composed with logical components that at this phase have more detail for being passed on to implementation teams.

Approaches differ from using a predefined artifact to using simplified versions initially, but the approaches in (Abrahamsson, Babar, & Kruchten, 2010; Cockburn, 2006; Coplien & Bjørnvig, 2011; Erdogmus, 2009; Farhan, Tauseef, & Fahiem, 2009; Harvick, 2012; Mancl, Fraser, Opdyke, Hadar, & Hadar, 2009; Rick Kazman, 2013; Waterman, Noble, & Allan, 2012; Zhang, Hu, Lu, & Gu, 2011) all advocate an initial model that afterwards is refined. Table 18 compares the research works presented in section 2.2, within the proposed AAL phases.

In any SDLC, whether waterfall, ASD, or other, the performed software engineering disciplines typically fall under the scope of business modeling, requirements, design, implementation, testing and deployment. The difference between these SDLC relies in the time where they are performed, but inputs from all disciplines are required.

Table 18. Comparison of agile architecting approaches and their contextualization within the architecting lifecycle

Agile architecting approaches	Grooming	Backlog	Delivery
(Nord & Tomayko, 2006)	Planning and stories	Designing	Analysis and Testing
(Jeon, Han, Lee, & Lee, 2011)	Planning and stories	Designing	-
(Farhan et al., 2009)	-	-	Analysis and Testing
(Sharifloo, Saffarian, & Shams, 2008)	Planning and stories	-	-
(Kanwal, Junaid, & Fahiem, 2010)	Develop an Overall Model, Build a Features List, Plan by Feature	Design by Feature	Build by Feature
(Madison, 2010)	up-front planning, storyboarding	Sprints	working software
(Díaz, Pérez, & Garbajosa, 2014)	Software Product Line (SPL) Backlog, Agile Product Line Architecting (APLA)	Sprints	Working Product-Line Architecture (PLA), Working Products (SPL)

Based in this premise, AAL pathway includes all disciplines, with the goal of incrementally evolving an architecture and reducing BDUF. Figure 70 proposes including in an AAL pathway description of Context, Functionalities, a Candidate Architecture and, then, a Refined Architecture.

In terms of Context, it relates to the knowledge acquisition of domain and enterprise settings where software solutions will execute. The understanding of such knowledge is the starting point. Functionalities relate to the definition of software needs (in opposition to a more process and business orientation of the previous stage), e.g., the definition of a "minimum viable product" (MVP). Then, a first candidate version is proposed and refined afterwards in order to emerge during delivery cycles. Proposing a candidate version relates to defining a high-level architecture, composed by main functional requirements and that allow defining a separation of concerns. This separation is input for planning of each concern implementation, which each relate to a subsystem of the architecture. During delivery cycles (e.g., Scrum Sprints), each subsystem is refined into a more detailed architecture, composed with logical components that at this phase have more detail for being passed on to implementation teams.

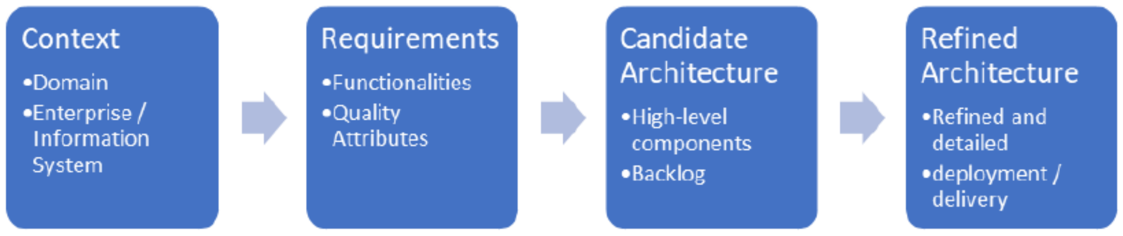


Figure 70. Steps proposal for agile architecting

As referred, Context relates to knowledge acquisition of the domain, enterprise, business and information system where the project is scoped. It is composed in its majority with Business Modeling tasks. Typical examples of this exercise may be the modeling of the enterprise's business processes (*e.g.*, using Business Process Modeling Notation – BPMN), identification of technical and/or product glossary, relationships between main domain concepts, specification of the structure of the involved systems (like the name, the exchanged data, the data location – *e.g.*, which table from the database, etc.) and how to access that data. This is typically performed under an "as-is" analysis, however, even when the aim is to perform the characterization of the "*to-be*" situation, it is advisable that the SDLC firstly includes a proper domain characterization, by analyzing the business processes, the information (data), and the systems (hardware/software) that compose the ecosystem.

Additionally, in Functionalities, process reference models have an interesting role in requirements elicitation. For instance, it is common that manufacturing sector follows Supply Chain Operations Reference (SCOR). These reference models are composed with processes, sub-processes, roles, tasks, operations, that easily may be mapped in a business process notation language (Santos, Duarte, Machado, & Fernandes, 2013). It is not expected that an enterprise follows only one reference model. For instance, the GS1 global standard for traceability is widely adopted for carrying out tasks for product traceability (Neiva, Santos, Martins, & Machado, 2015), and may be adopted complementary.

Now that the solution needs are identified – and properly specified – the next step typically relates to designing the system. System design is typically performed using a model, *e.g.*, an architecture. However, architecture design should be addressed as an iterative process, as design should start in a conceptual level and refined until it is detailed enough, which is to say the abstraction level goes from high to low during this process. For that reason, the pathway proposes designing a Candidate Architecture and afterwards a Refined Architecture.

Architecture design includes from conceptual level to more refined one (Fernandes & Machado, 2016). Such argument is in line with the design process proposed by Douglass: architectural, mechanistic, and detailed (Douglass, 1999). The evolution of architecture was already discusses in Section 3.4.

Figure 71 presents the results from classifying Candidate and Refined architectures using the proposed framework. The presented classification is as follows:

Candidate Logical – **Phase**: Analysis/Design; **4+1**: Logical; **Abstraction**: CIM/PIM.

Refined Logical – **Phase**: Design; **4+1**: Logical; **Abstraction**: PIM/PSM.

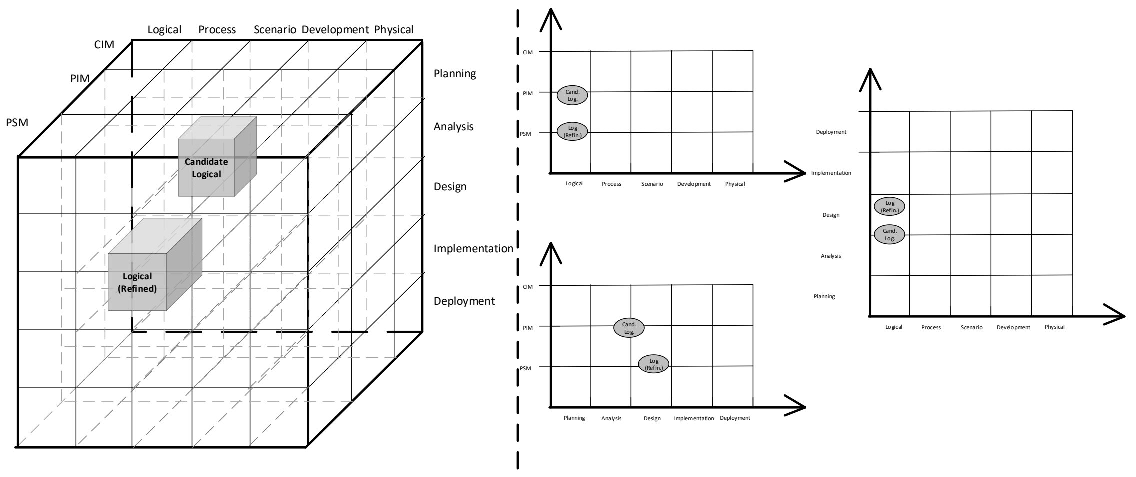


Figure 71. Classification of Candidate and Refined logical architectures

The AAL is thus composed by three phases: Stories, Planning and Delivery. Figure 72 depicts the pathway between phases under the eleven viewpoints. Additionally, **Table 19** inputs and outputs between viewpoints throughout the AAL.

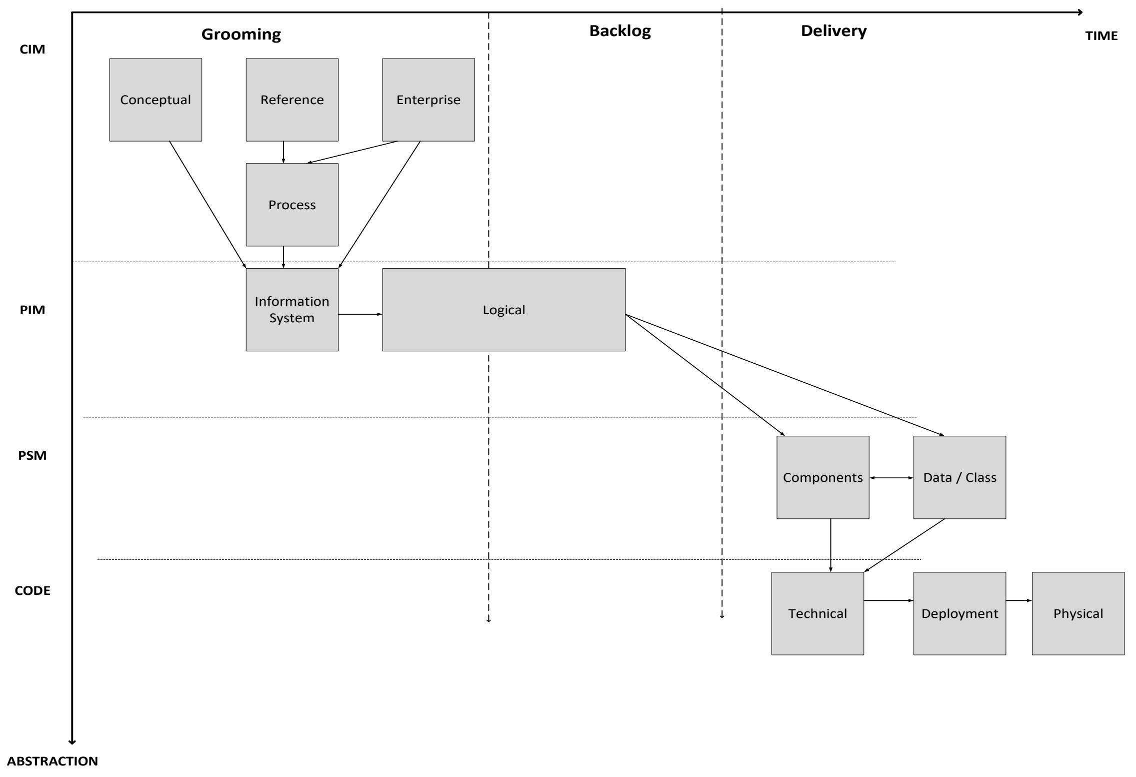


Figure 72. Architectural views and abstraction within AAL phases

Table 19. The inputs and outputs of AAL artefacts

View	Input	Output	Relation SA
Conceptual	e.g., Product Vision, Product Roadmap, Domain model	e.g., Concepts, vocabulary, ontology	Information System
Reference	Process Reference Models	Domain's best practices	Process
Enterprise	Structure of Enterprise	Enterprise processes	Process
Process	Business processes, Process reference models	Process requirements	Information Systems
Information System			Logical
Logical			Components, Data/Class
Components			Technical
Data/Class			Technical
Technical			Deployment
Deployment			Physical
Physical			

5.3. Architecture evolution and management

Candidate architecture design using the 4SRS method

This research uses the 4SRS method, by taking advantage from the method's ability to use a functionally decomposed UML Use Case model – so, requirements have been elicited and split in smaller pieces as possible - as input to derive in a stepwise manner a component-based logical architecture (i.e., the software product version of the 4SRS). For this reason, the method evolution is presented and its steps are detailed in this section.

The 4SRS method takes as input a set of use cases in the problem space, describing the requirements for the processes that tackle the initial problem. They are then refined trough successive 4SRS iterations (by recurring to tabular transformations), producing progressively more detailed requirements and a design specification, in the form of a logical architecture representation of the system, representing the intended concerns of the involved business and technological stakeholders. These tabular transformations are supported by a spreadsheet where each column has its own meaning and rules. Some of the steps have micro-steps, of which some can be completely automated. A correct application of the tabular transformations assures alignment and traceability, between the derived logical architecture diagram and the initial use cases representations, and at the same time allows adjusting the results of the transformation to any changing requirements.

The 4SRS method has proven successful in differentiated contexts, for instance:

- Process architectures (Nuno Ferreira, Santos, Machado, Fernandes, & Gasević, 2014; Nuno Ferreira et al., 2012; Machado, 2002),
- Software product architectures (Fernandes, Machado, Monteiro, & Rodrigues, 2006; Machado, Fernandes, Monteiro, & Rodrigues, 2005; Machado et al., 2006),
- Software product lines (A Bragança & Machado, 2007, 2005; Alexandre Bragança & Machado, 2009),
- Software product lines with variability (Azevedo, Machado, Muthig, & Ribeiro, 2009; Azevedo, Machado, & Maciel, 2012),
- Class diagrams (Cruz, Machado, & Santos, 2014; Santos & Machado, 2010) and
- Service-oriented (Salgado, Teixeira, Santos, Machado, & Maciel, 2015) - namely, SoaML (OMG, 2012) - logical architectures.

The usage context of the proposed 4SRS, with “just-enough requirements and deriving the logical architecture with “just-enough” components is depicted in Figure 73. Previous experiences with the 4SRS method relate do BDUF contexts, where a larger number of requirements were known upfront, rather than executing the method with smaller number of requirements. However, since in AMPLA the requirements will be later refined and will emerge, the 4SRS method is regularly revisited alongside the development Sprints.

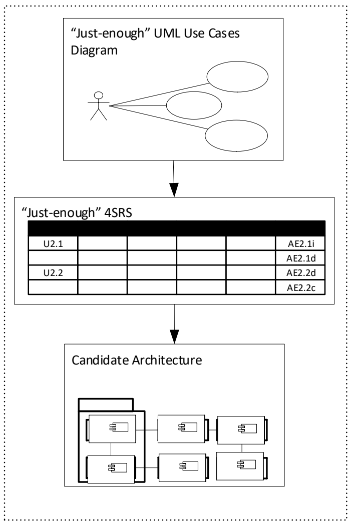


Figure 73. Method for designing the candidate architecture with 4SRS

The 4SRS method is composed by four steps: Component Creation; Component Elimination; Packaging and Aggregation; and Component Associations.

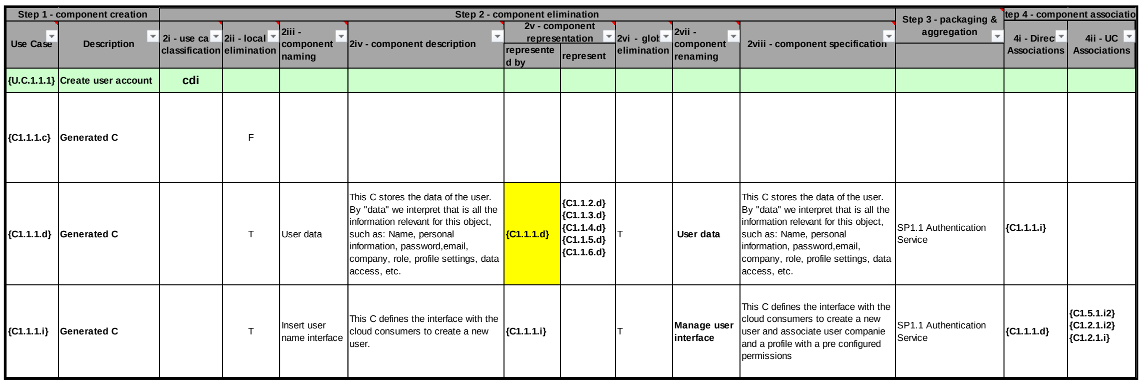
The first step regards the creation of software architectural components. The 4SRS method associates, to each component found in analysis, a given category: interface, data, and control. Interface components refer to interfaces with users, software or other entities (*e.g.*, devices, etc.); data components refer to generic repositories, typically containing the type of information to be stored in a database; and control components refer to the business logic and programmatic processing. This categorization makes the architectures derived by the 4SRS to be compliant with architectures from object-oriented programming, or by Model-View-Controller (MVC) patterns.

In the second step, components are submitted to elimination according to pre-defined rules. At this moment, the system architect decides which of the original three components (i, c, d) are maintained or eliminated, firstly taking into account the context of a use case from Step 1, and later compared for redundancy within the entire system. Additionally, each component is named and textually described relating to its behavior.

In the third step, the remaining components (those that were maintained after executing step 2), where there is an advantage in treating them in a unified process, should give the origin to aggregations or packages of semantically consistent components.

The final step refers to the associations between components. The method provides steps for identifying such associations based in descriptions from use cases, as well as from the own components during the components creation.

The execution of the 4SRS transformation steps can be supported in tabular representations. These representations enables partial automation of the transformations steps and constitute the main mechanism to automate a set of decision assisted model transformation steps. A small part of the 4SRS method execution table is represented in Figure 74. The table is not zoomed due to size limitations. The cells are filled with the set of decisions that were taken and made possible the derivation of a logical architecture. Each column of the table concerns a step/micro-step of the method execution.



Step 1 - component creation		Step 2 - component elimination									Step 3 - packaging & aggregation	tep 4 - component associatio	
Use Case	Description	2i - use ca classification	2ii - local elimination	2iii - component naming	2iv - component description	2v - component representation		2vi - glob elimination	2vii - component renaming	2viii - component specification		4i - Direc Associations	4ii - UC Associations
						represented by	represent						
{U.C.1.1.1}	Create user account	cdi											
{C1.1.1.c}	Generated C		F										
{C1.1.1.d}	Generated C		T	User data	This C stores the data of the user. By "data" we interpret that is all the information relevant for this object, such as: Name, personal information, password,email, company, role, profile settings, data access, etc.	{C1.1.1.d}	{C1.1.2.d} {C1.1.3.d} {C1.1.4.d} {C1.1.5.d} {C1.1.6.d}	T	User data	This C stores the data of the user. By "data" we interpret that is all the information relevant for this object, such as: Name, personal information, password,email, company, role, profile settings, data access, etc.	SP1.1 Authentication Service	{C1.1.1.i}	
{C1.1.1.i}	Generated C		T	Insert user name interface	This C defines the interface with the cloud consumers to create a new user.	{C1.1.1.i}		T	Manage user interface	This C defines the interface with the cloud consumers to create a new user and associate user companie and a profile with a pre configured permissions	SP1.1 Authentication Service	{C1.1.1.d}	{C1.5.1.i2} {C1.2.1.i2} {C1.2.1.i}

Figure 74. 4SRS method execution using tabular transformations

The output is a logical architectural diagram, composed by a set of software components and relations/flows between them. The architectural diagram is modeled in UML Components, as depicted in a simple example in Figure 75.

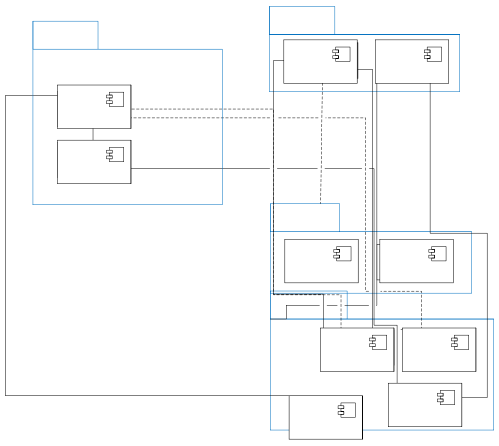


Figure 75. Simple example of a candidate UML components architecture

It must be pointed out that AMPLA and the 4SRS supports the logical view (Kruchten, 1995) of the architecture, namely the identification and design of software components referring to functional requirements. Architecture design should also address quality requirements. Addressing quality requirements is out of the scope of AMPLA, but is able to coexist with other architecture-centric methods such as QAW or ADD.

The use of 4SRS throughout the process, first in the scope of the candidate architecture, and afterwards in the scope of each refinement, provides the traceability between components and the functional requirements, allowing an agile response to changing requirements. The number of components or the number of associations between components are possible indicators to determine the architecture's riskiest components.

Incremental design for refining the logical architecture

As referred in the previous section, the output of performing the 4SRS method during requirements is a candidate logical architectural diagram, composed by a set of software components and relations/flows between them.

The idea is to refine the architecture incrementally, in parallel with the implementation efforts regarding the architectural components that already were refined. Like in most cases, the best approach to issue complex problems is to divide them into smaller ones and address them one by one, ultimately allowing to address the big solution.

With the purpose of modularizing the architecture, the candidate logical architecture is partitioned into sub-systems. AMPLA is able to be performed in parallel within a typical ASD context, however the subsystem bordering is helpful in complex ecosystems contexts. This bordering is likely to reflect the ecosystem, e.g., IoT, cloud, infrastructure, embedded systems, etc., and the dependencies between the subsystems. The sub-system border provides context for modeling more refined Use Cases, as well as context for making technical design decisions that in the initial phase were very difficult to make. The Use Cases may then be used for a new execution of the 4SRS (Figure 76), since they relate to "new" Use Cases that emerged and hence did not exist before the definition of the sub-systems. Within these "new" Use Cases, they may relate to refined use cases from the previous model, as well as relating to new use cases within the same level of refinement. *E.g.*, a sub-system related to a use case {UC1.5} may result in refinements like {UC1.5.1} and {UC1.5.2}, as well as new use cases in the same level of refinement like {UC1.6} and {UC1.7}.

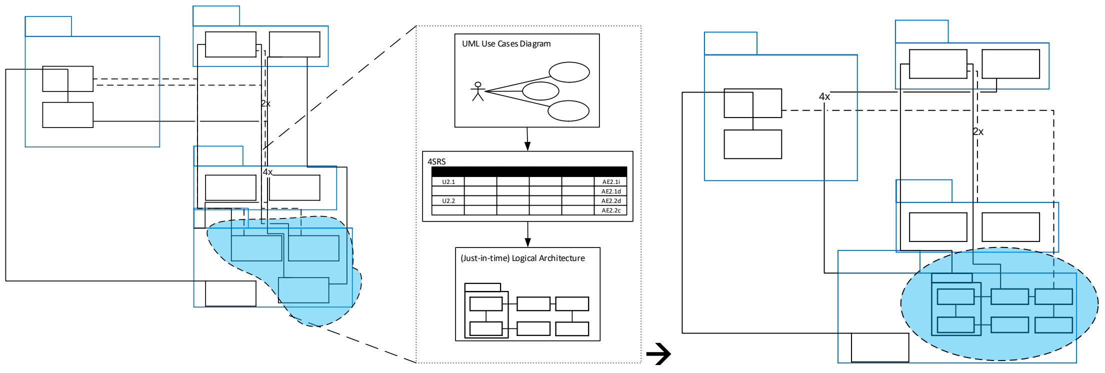


Figure 76. Recursive execution of 4SRS for refining a given example module

By performing the 4SRS method within this new use case model, the resulting software components relate to refined components for that sub-system that can be "replaced" in the initial candidate architecture, resulting in a more refined version. In order to assure that the defined sub-system, even though composed by more components, is still able to fit in the overall logical architecture, its interfaces must be maintained (system of systems theory). It must be pointed out that the execution of the 4SRS for the new and emerged use cases in Figure 76 are an increment to the previous execution, rather than performing a new 4SRS execution with only the new use cases.

The sub-system requirements are object of analysis within each iterative cycle. In this section is described the refinement process aligned with iterative cycles, e.g., Scrum Sprints. Figure 77 depicts the distribution of the module requirements for refinement and respective implementation tasks to be included as Product Backlog Items (PBI), distributed within the cycles (Sprints).

After each subsystem is defined, the process is then structured as an iterative approach (e.g., Scrum Sprints). Within every iteration, the team performs tasks involving several software engineering disciplines. The terminology from RUP's and AUP's disciplines is used (only for demonstration purposes) to depict the type of effort involved within the Sprints. These efforts are illustrated in Figure 77 by the colored bars within each cycle, where each bar is a software engineering disciplines, and with more or less effort during the cycle, similar to RUP and AUP. The main difference is that, in parallel with carrying out typical disciplines within the Sprints (Implementation, Testing and Deploy) that result in the delivery of a software increment, other team members are responsible for refining requirements regarding the features not yet included in the team's Sprint Backlog, and that are planned to be implemented in further Sprints. These requirements are modeled, hence the Requirements discipline, and then an incremental execution of the 4SRS method, hence the Analysis & Design (A&D) discipline.

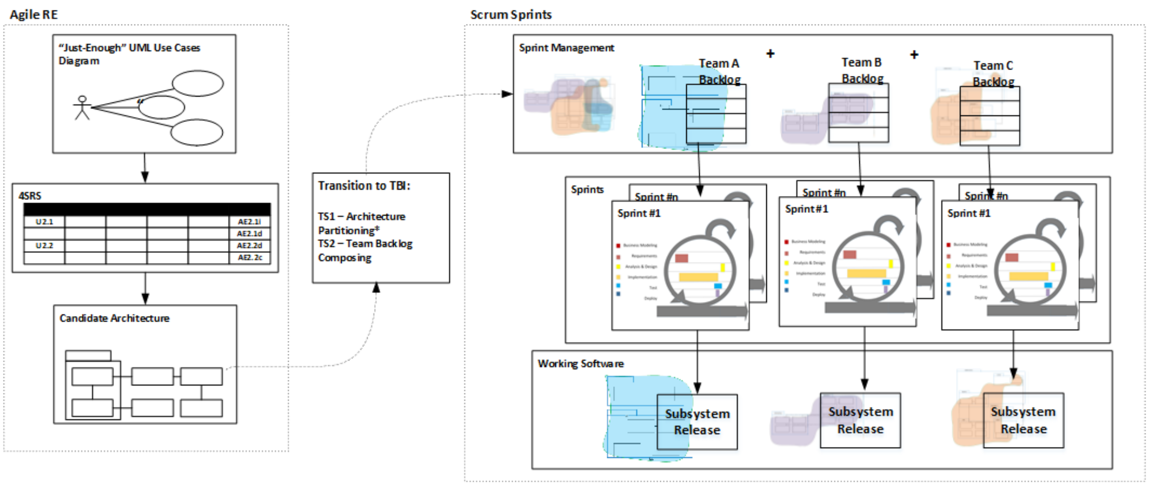


Figure 77. Distributed implementation of each architecture module

The Requirements and A&D tasks are performed in iterative cycles and incrementally, synchronized with the development and deployment during the Sprints, in a sense that what is modeled during a cycle is then ready to be implemented in next cycles (Figure 78). While the implementation is being performed, requirements from the original logical architecture, not yet refined in previous cycles, are modeled in use cases. They originate a new increment of the 4SRS method, deriving additional components. These models require validation from customer before being ready to be included in the Team Backlog. The first time this requirements cycle is being performed, the development Sprint is not yet performed (or a Sprint 0 cycle is performed where no software is delivered, but rather the development infrastructure is built). Then, requirements and the 4SRS address the functionalities to be implemented in the next Sprint or even more ahead Sprints. It is assumed that, since the components have been refined, they are in an improved situation to be now passed for implementation.

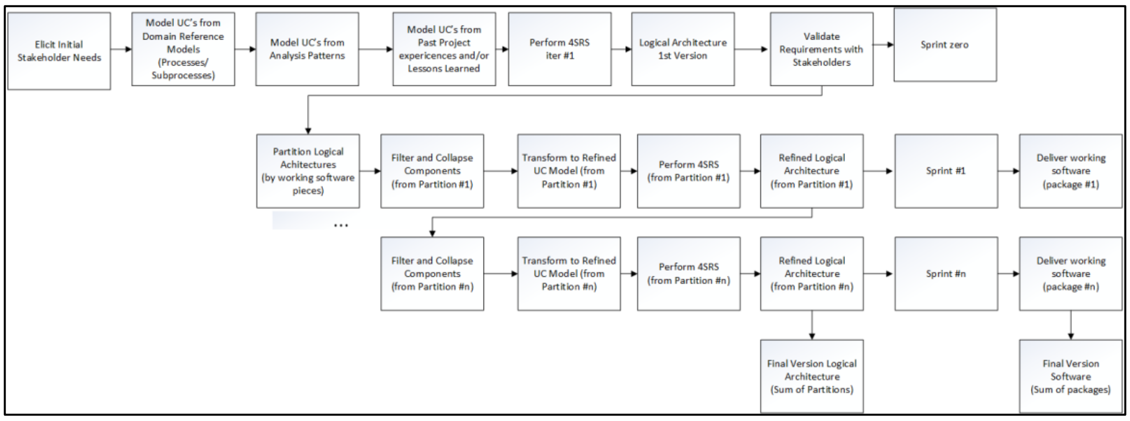

Elicit Initial Stakeholder Needs
Model UC's from Domain Reference Models (Processes/ Subprocesses)
Model UC's from Analysis Patterns
Model UC's from Past Project expericences and/or Lessons Learned
Perform 4SRS iter #1
Logical Architecture 1st Version
Validate Requirements with Stakeholders
Sprint zero
Partition Logical Achitectures (by working software pieces)
Filter and Collapse Components (from Partition #1)
Transform to Refined UC Model (from Partition #1)
Perform 4SRS (from Partition #1)
Refined Logical Architecture (from Partition #1)
Sprint #1
Deliver working software (package #1)
Filter and Collapse Components (from Partition #n)
Transform to Refined UC Model (from Partition #n)
Perform 4SRS (from Partition #n)
Refined Logical Architecture (from Partition #n)
Sprint #n
Deliver working software (package #n)
Final Version Logical Architecture (Sum of Partitions)
Final Version Software (Sum of packages)

Figure 78. Incremental requirements and 4SRS execution throughout the Sprints

While this refined information is implemented, coded, tested and deployed, ending in a delivery of a software increment to customer, in parallel other model is refined within the same approach (*i.e.*, use cases, 4SRS and architecture). This process is depicted in Figure 79 as a Software Process Engineering Metamodel (SPEM) diagram. This way, when a cycle is finished, it is expectable that new sub-systems were refined and able for implementation.

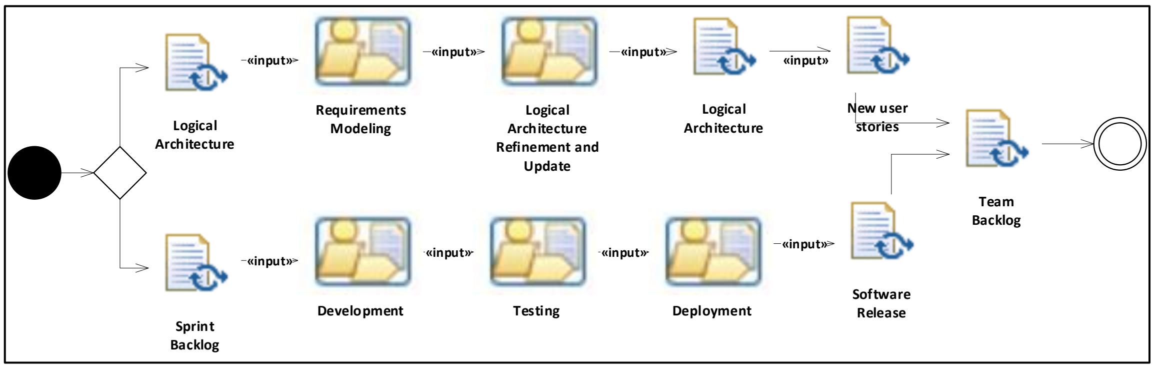


Figure 79. Parallel tasks within Sprints in SPEM diagram

Continuous Architecture and change-impact analysis

After the feedback and consequent learning and adjustments (if needed), the approach ends with the candidate logical architecture, which is then used as input for defining the required backlog items for delivering the software (please consult Chapter 6 for more details).

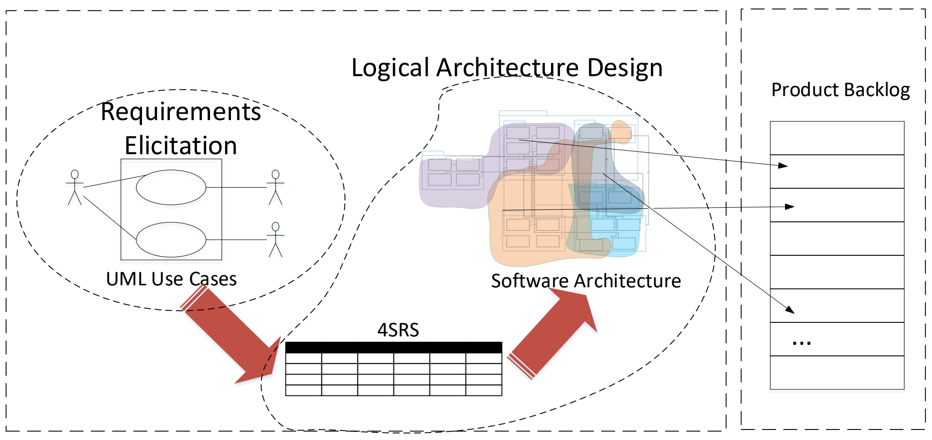


Figure 80. Traceability between models and product backlog

The derivation of models described so far is performed before the Sprints cycles. Only then, there is context for deriving backlog items. The transformation of the artifacts in backlog items may be performed before or during a Sprint 0 cycle, where no software was delivered. When software delivery begins, the process is performed in typical cycles, whether in Scrum, Kanban, or other frameworks. Figure 81 depicts a process structured in Scrum Sprints.

The main difference is that, in parallel with delivering a software increment, other team members are responsible for refining requirements not yet included in the backlog, and that are planned to be implemented in further Sprints.

In all Sprints, the requirements refining for further Sprints justifies the inclusion of the Requirements discipline in the each Sprint, as shown in Figure 82. Other example of Requirements discipline are spikes (originally defined within XP), a technique used for activities such as research, innovation, design, investigation and prototyping. With spikes, one can properly estimate the development effort associated with a requirement or even a better understanding a requirement.

Some design-oriented spikes (similar to architectural spikes from XP) like monitoring the current status of the system, controlling the technical debt, compliance with regulatory requirements, among others, result in a focus in Analysis & Design. These are some of the continuous architecting (CA) tasks (Martini, Pareto, & Bosch, 2014). Afterwards, the typical disciplines are carried out within the Sprints: Implementation, Testing and Deploy. We use the terminology from RUP's disciplines (only for demonstration purposes) to depict the type of effort involved.

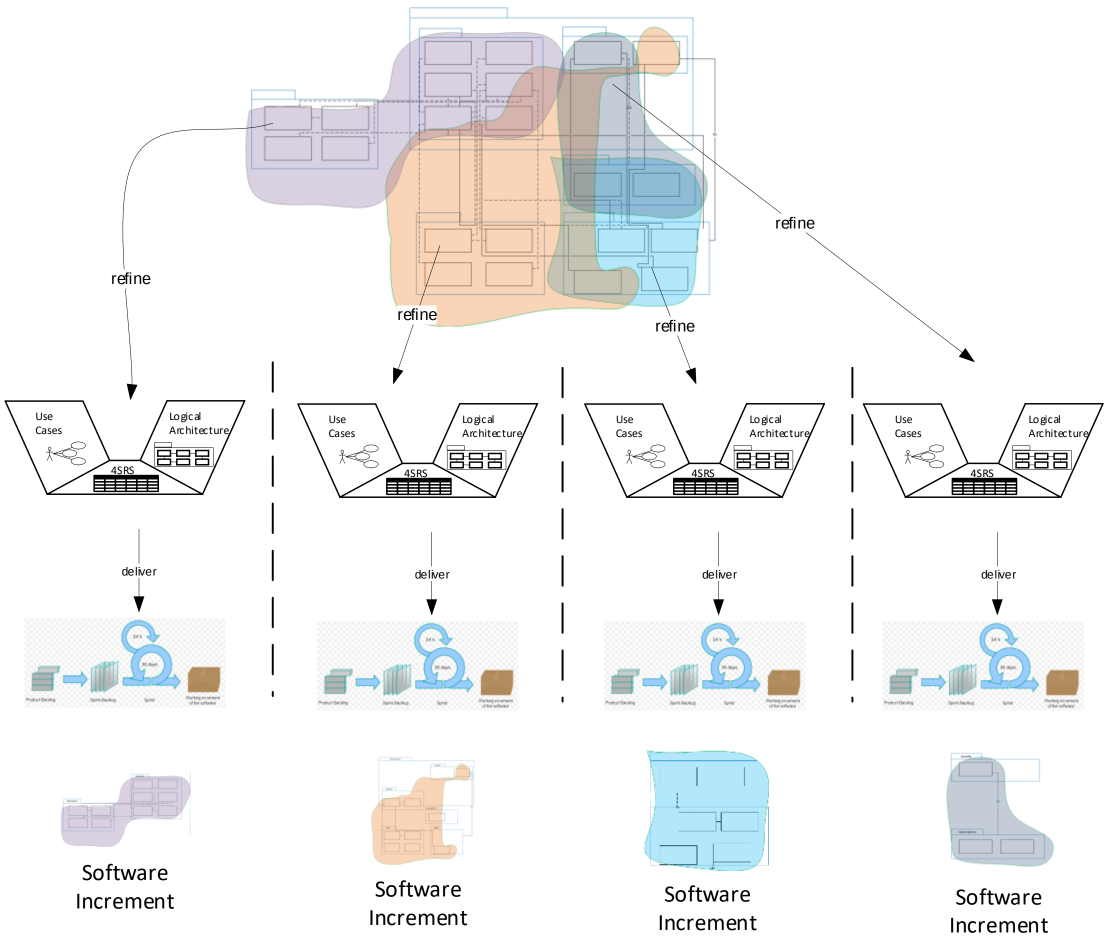


Figure 81. AMPLA during Sprints

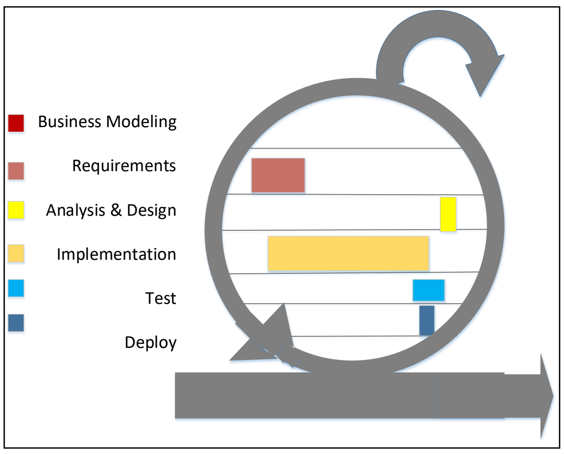


Figure 82. The performed disciplines within the Sprints

It is during these delivery cycles (in Figure 83 the cycles are represented as Scrum Sprints) that the software supervision (of what is delivered) is performed. As ASD promote rapid feedbacks and adaptation, many events (or ceremonies) are performed whether for customer feedback on the delivered software increment, or for team review. In the later, many design and implementation decisions are made, whether related to the team's perception of the delivered quality, the development process, the adopted technologies or concerning the way the architecture is emerging. Moreover, it is based within the customer feedback or team review that changes are proposed during the cycles. For both cases, a proper in-depth analysis (especially for safety-critical systems) related to the proposed changes should be performed.

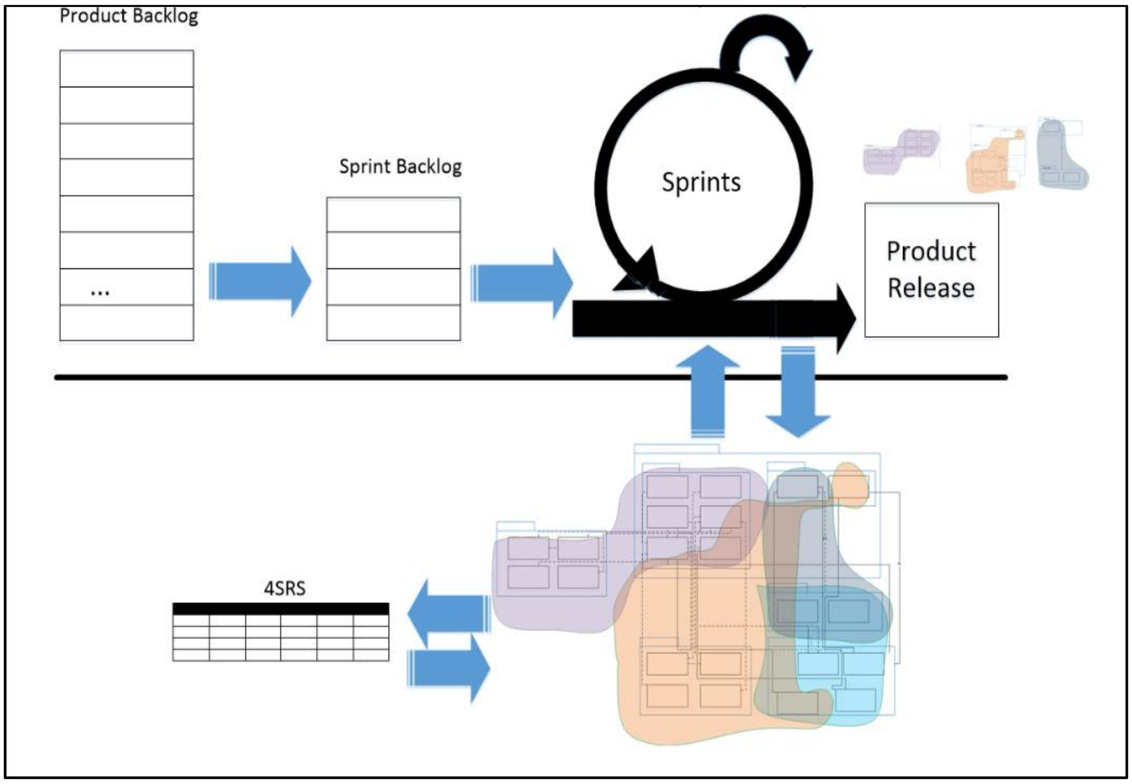


Figure 83. Parallel traceability within Sprints using the 4SRS

Just like during the refinement process, the remaining Sprints the teams perform tasks related to several disciplines, as previously depicted in Figure 82. In this section, the "Analysis & Design" discipline is focused.

The architecture should be present at all moments of the development. Whenever new requirements emerge, or a need for change/refactoring of a given component is identified, the architect should carefully analyze the impact before accepting such occurrences.

As described in the previous section, AMPLA provides traceability between requirements models (UML Use Cases), architectural components (UML Components) and the product backlog items. Thus, it is within these three artifacts that impact analysis is based in.

The architect may analyze the architectural artifact (arrow (a) in Figure 84) for a component candidate for refactoring, which remaining components are affected by any change to that component, or depicting which quality characteristic is affected. Alternatively, the architect may trace back to the requirements model for depicting the business value of the requirements that relates to the component affected by the refactoring (arrow (b) in Figure 84). Finally, when a new requirement emerges, it is added to the UML Use Cases model and afterwards "is followed" the AMPLA process flow towards the architecture and product backlog (arrow (a) in Figure 84). Arrow (c) may also relate to address technical and architectural debt, as a requirement typically is refined which ultimately gives origin to a product backlog item (User Story and backlog task) in order to tackle the debt.

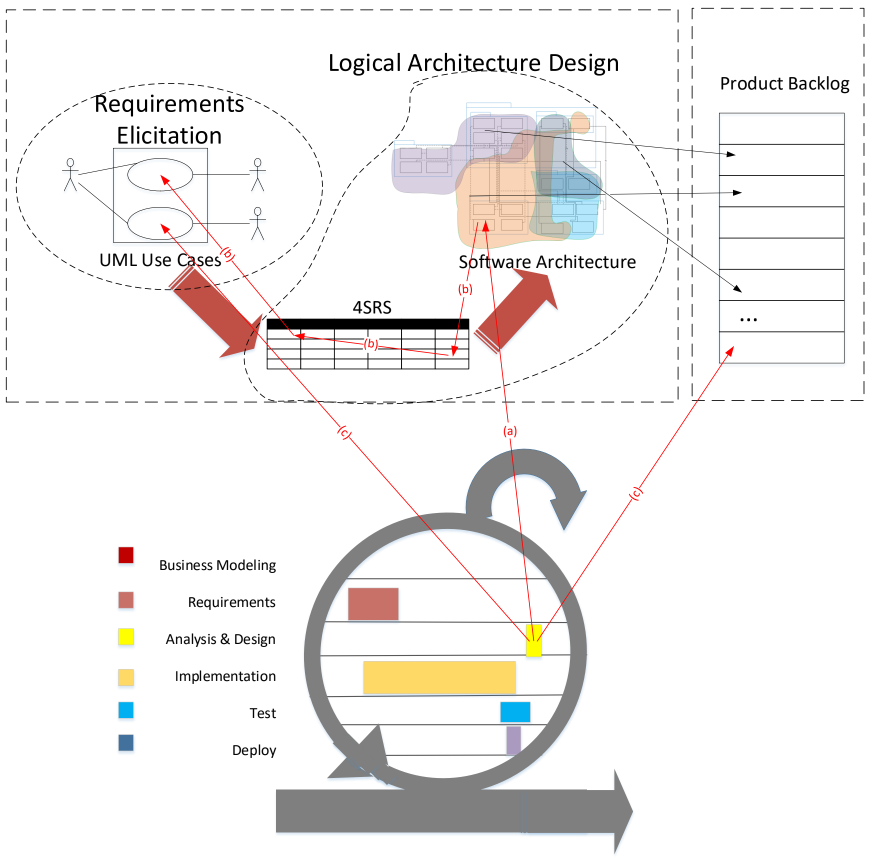


Figure 84. Possible targets of CIA within AMPLA

Martini, Pareto and Bosch describe context, scope and within which agile ceremony was more appropriate for a set of given CA practices (Martini et al., 2014). In AMPLA it is also agreed that CA, as the name implies, is a continuous process, thus the proposed CIA practices should not be limited to a moment in time within cycles (or Sprints), like, e.g., Retrospectives, but rather at any time of the delivery. Of course, CIA practice could not be performed right after a given change proposal, although evidently the quickest response to that proposal is adequate.

It is not the purpose of this research to provide inputs for architects to accept or deny any change proposal, but rather to provide CIA practices insights and where the architectural information (derived within AMPLA) is located in order for providing the required inputs for an appropriate decision, as listed in Table 20.

Table 20. List of CIA practices and their targeted models

Type of practice	Target	Analyzed model
Architecture significant requirement (ASR)	The goal which derived the ASR	UML Component
Quality characteristic	Capability of assuring SLAs	UML Component
Business and customer value of the requirement	Priority of the requirement	UML Use Cases
Which components are affected	Dependencies between components	UML Components, 4SRS
Compliance with standards	Legal and regulation assurance	UML Components
Requirements emerge	Add new requirements	UML Use Case
Managing architectural debt	Refine requirements and update architecture and product backlog	UML Use Case, UML Components and Product Backlog items

Thus, the proposed CIA practices allow analysing, monitoring and assessing the architectures in many ways. If analysed all, or just some of them, the architect is able to provide an appropriate acceptance or denial of a change proposal.

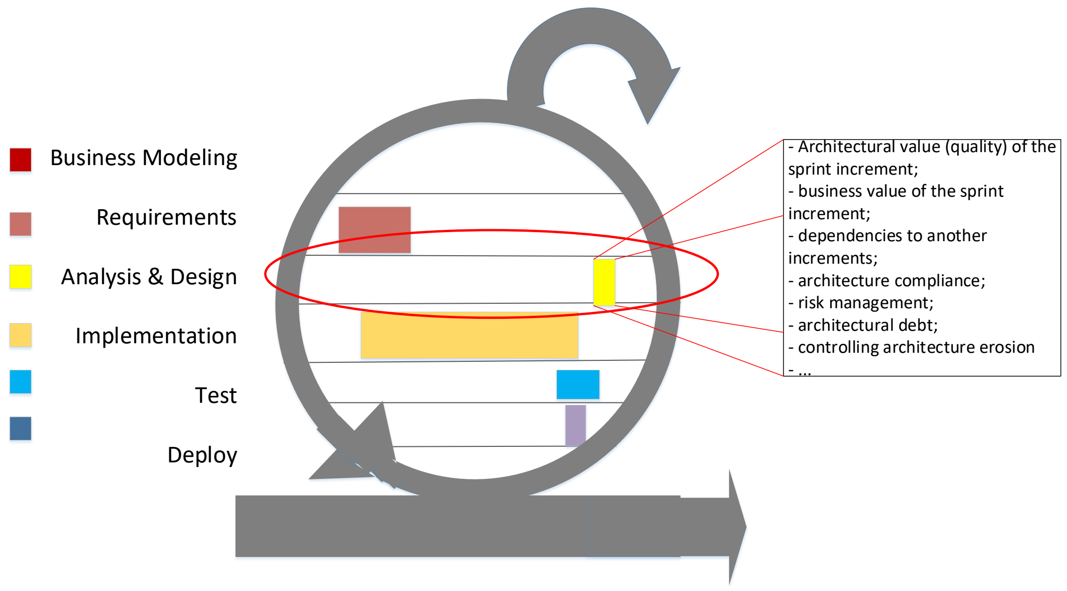

Business Modeling
Requirements
Analysis & Design
Implementation
Test
Deploy
- Architectural value (quality) of the sprint increment;
- business value of the sprint increment;
- dependencies to another increments;
- architecture compliance;
- risk management;
- architectural debt;
- controlling architecture erosion
- ...

Figure 85. CA-related practices within "Analysis & Design" discipline

Microservices design towards Continuous Architecting

Breaking the monolith system

Any software project is performed in one of these contexts: either in a greenfield project, *i.e.*, from scratch; or alternatively, in a brownfield project, *i.e.*, on top of an existing legacy system. Independent of whether an MSA-based system is designed, greenfield or brownfield (by decomposing a monolith), a set of microservices need to be identified (Newman, 2015). Accordingly, the 'Decomposition by business capability' pattern or 'Decomposition by subdomain' (Richardson, 2018) - also known as DDD - is a common approach for designing microservices architectures.

In this section, whether the context is greenfield or brownfield, use cases are always the final output form the initial domain modeling process, as they are the input for the 4SRS-MSLA method. If in a greenfield project, the requirements are elicited based in the processes that the MSA aims to support, where we propose the use of Use Case models, requirements that are structured by decomposing functionalities in a tree-like form (Figure 86). The tree's "branches" are a reflection of the boundaries of the domains and subdomains derived from applying DDD pattern. If in a brownfield project, the decomposition begins by analyzing the processes and their mapping within the monoliths' components. The elicitation refers to processes that legacy (or, in this case, monoliths) systems currently support. The processes are the basis for identifying the domains with the DDD pattern. By applying reverse engineering to requirements modeling, the output is again an UML Use Case model.

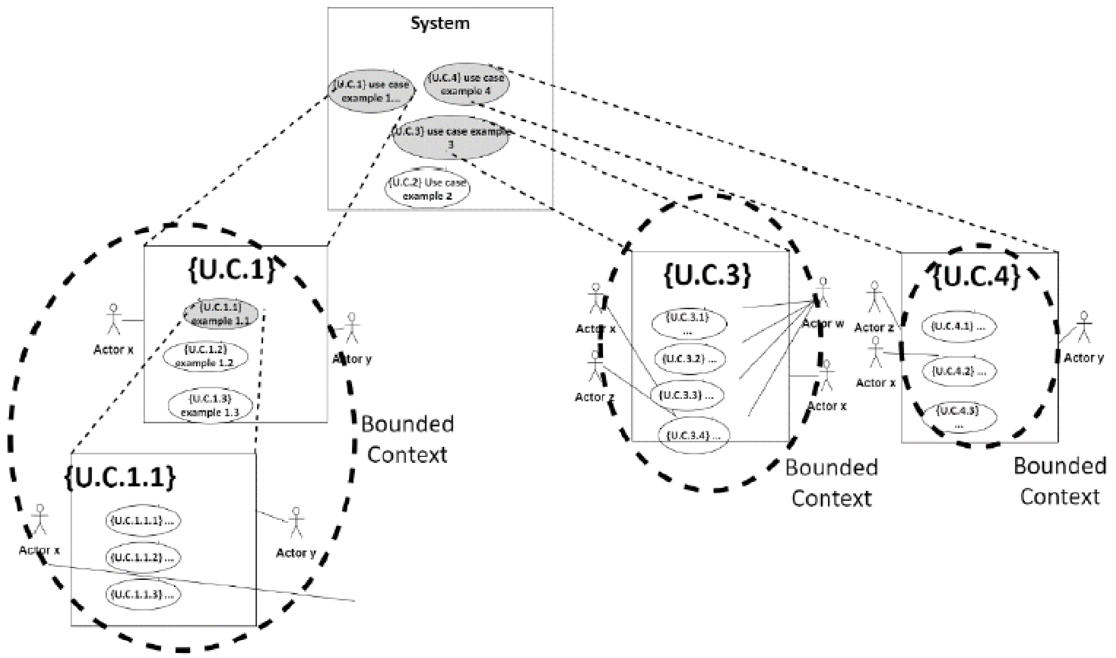


Figure 86. Inferring Domain's and sub-domain's bounded contexts from UML Use Cases

Each use case is functionally decomposed based on specific tasks that the current solution supports. Additionally, this elicitation requirements process may be used to analyze current limitations on the business domain and hence stakeholders may elicit new requirements, which are added to the use case model.

Functional decomposition relates to dividing functionalities in smaller pieces. Within UML use cases, this requirements technique is used for enabling a software functionality to be divided into refined use cases. Such technique allows for providing detail in describing a given requirement. This tree-like organization results in a structure for specifying lower-level details on each requirement, where low-level requirements are a specialization of the higher-level requirements. The requirements analyst then decides when decomposition may end. Often, the tendency is to perform scaffolding, where low-level requirements end up in Create, Read, Update and Delete (CRUD) operations. This is just an example, since decomposing aiming for CRUD operations is a decision that the requirements analyst has to make.

Greenfield settings

The requirements engineering process follows the DUARTE approach within the AMPLA process (cf. Section 4.3) that, based in successive model derivation, namely referring to sequence, use case and components diagrams, allows deriving just enough requirements/use cases into a candidate logical architecture. Requirements emerge in a continuous way, as the 4SRS method is regularly revisited alongside the development Sprints (cf. Section 5.3). Although not mandatory that the requirements engineering process (necessary for the proposed microservices design approach) follows either DUARTE or AMPLA, for the purpose of this research this section describes how this approach for microservices uses the resulting models from DUARTA/AMPLA. However, it is only necessary that the requirements engineering process outputs a UML use cases diagram.

Designing microservices for a given business capability or domain, typically uses patterns such as Domain-driven Design (DDD) (Evans, 2004), single responsibility principle (SRP) or Conway's Law (Conway, 1968). However, microservice design often faces challenges related to database partition, the proper size of the microservice, inter-service communication and messaging, which are not addressed systematically by those patterns. By applying a modeling method in the process of designing a MSA, one may foresee issues on bounded contexts for microservices, namely intra-service behavior, interfaces and data models separation, and inter-service communication and messaging requirements (Newman, 2015).

The requirements elicitation started by listing a set of stakeholder expectations towards the product roadmap, encompassing the entire product but only MVP features were detailed. They relate to business needs that afterwards allowed depicting functional requirements, modeled in use cases. After executing the 4SRS, the logical architecture is derived. This architecture was afterwards divided in a set of modules to be assigned to each of the project's teams (Figure 87). The modularization exercise followed the DDD rationale.

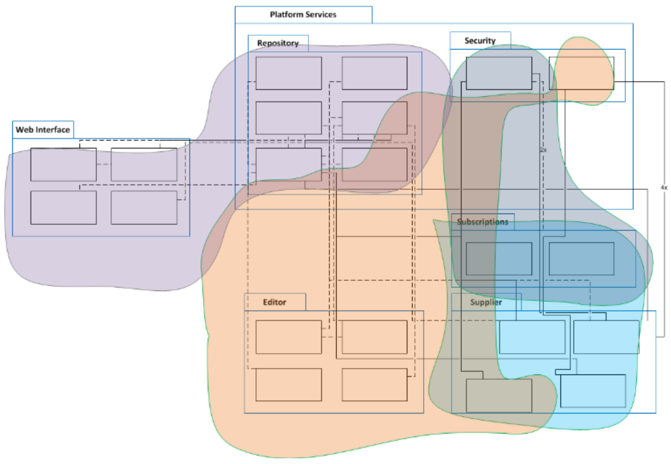


Figure 87. Architecture modularization example

By identifying the domains present in the architecture (DDD) we propose to refine sub-systems (regarding each domain) of the architecture iteratively, in order to identify, model and specify a set of software services in SoaML diagrams, such as Service Participants, Service Interface, Capabilities, Service Data, Service Architecture, Service Contracts, among others, until all logical components are supported by software services.

The 4SRS method takes as input a set of UML Use Cases describing the user requirements and derives a software logical architecture using UML Components. The logical architecture is then refined trough successive 4SRS iterations (by recurring to tabular transformations), producing progressively more detailed requirements and design specifications. An overview of the approach is depicted in Figure 88.

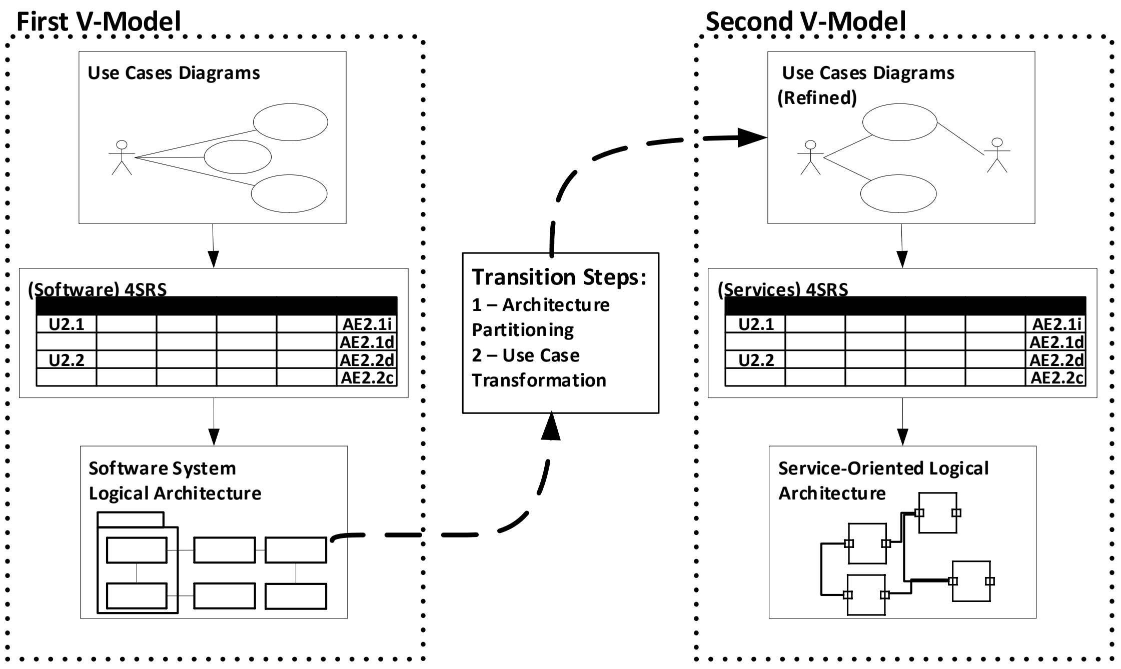

First V-Model
Use Cases Diagrams
(Software) 4SRS
U2.1
U2.2
AE2.1i
AE2.1d
AE2.2d
AE2.2c
Software System Logical Architecture
Transition Steps:
1 – Architecture Partitioning
2 – Use Case Transformation
Second V-Model
Use Cases Diagrams (Refined)
(Services) 4SRS
U2.1
U2.2
AE2.1i
AE2.1d
AE2.2d
AE2.2c
Service-Oriented Logical Architecture

Figure 88. Recursive architectural model transformations for service design

Each module could have sub-domains, which was responsibility of the assigned team to identify them. At this point, it was also important to identify dependencies and flows between domains for performing aimed business processes. The modeling support for this exercise can be, e.g., sequence diagrams as in Figure 89, where microservices regarding the sub-system were identified within the scope of a given business process. In fact, these diagrams are powerful tools for bordering the modules, as well as validating (not just the modules but as well the whole) architecture. Additionally, defining the sequence flows also supported eliciting communication specification between microservices.

Designing microservices logical architectures in SoaML using the 4SRS-MSLA

Resulting from the modularization, now each sub-system is refined independently. For that purpose, new UML Use Cases are identified, regarding only the sub-system, in order to refine the existing information. This section describes the steps that comprise the 4SRS-MSLA method (Figure 90), from where each UML component is initially specified. Next, these components are identified and their behavior derived in microservices (SoaML's Service Participants), as also the channels and contracts between them.

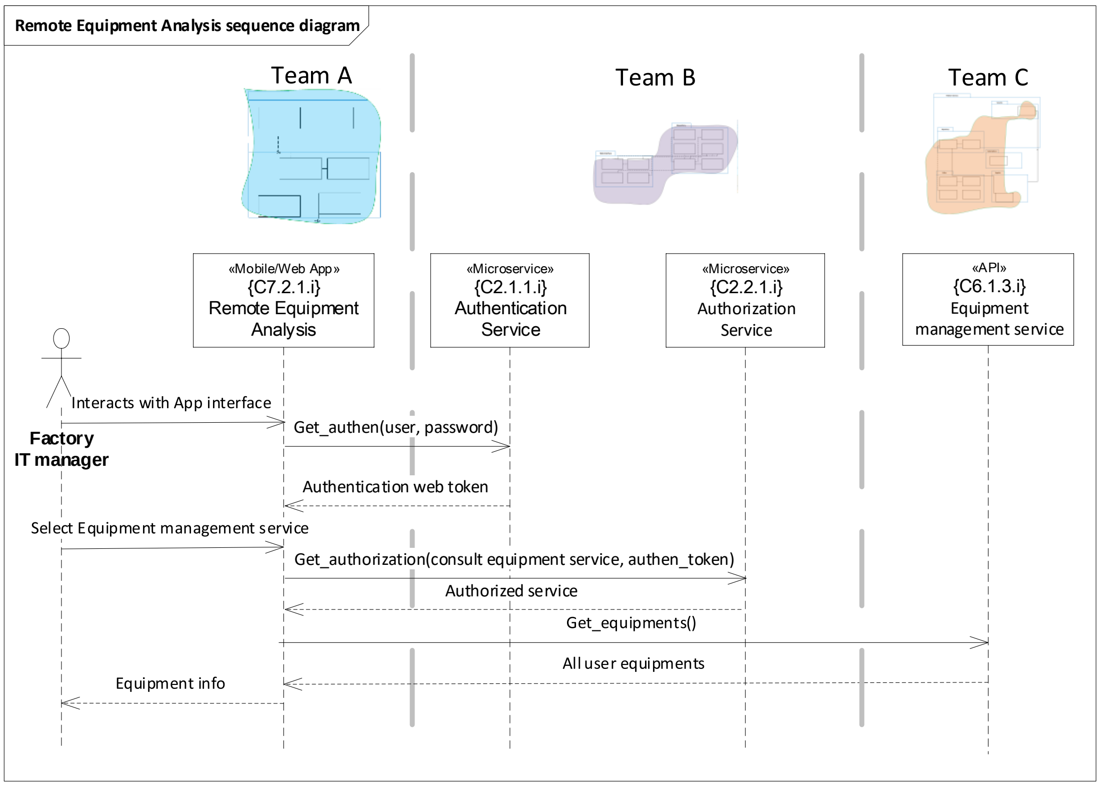

Remote Equipment Analysis sequence diagram
Team A
Team B
Team C
«Mobile/Web App» {C7.2.1.i} Remote Equipment Analysis
«Microservice» {C2.1.1.i} Authentication Service
«Microservice» {C2.2.1.i} Authorization Service
«API» {C6.1.3.i} Equipment management service
Factory IT manager
Interacts with App interface
Get_authen(user, password)
Authentication web token
Select Equipment management service
Get_authorization(consult equipment service, authen_token)
Authorized service
Get_equipments()
All user equipments
Equipment info

Figure 89. Dependency between different teams

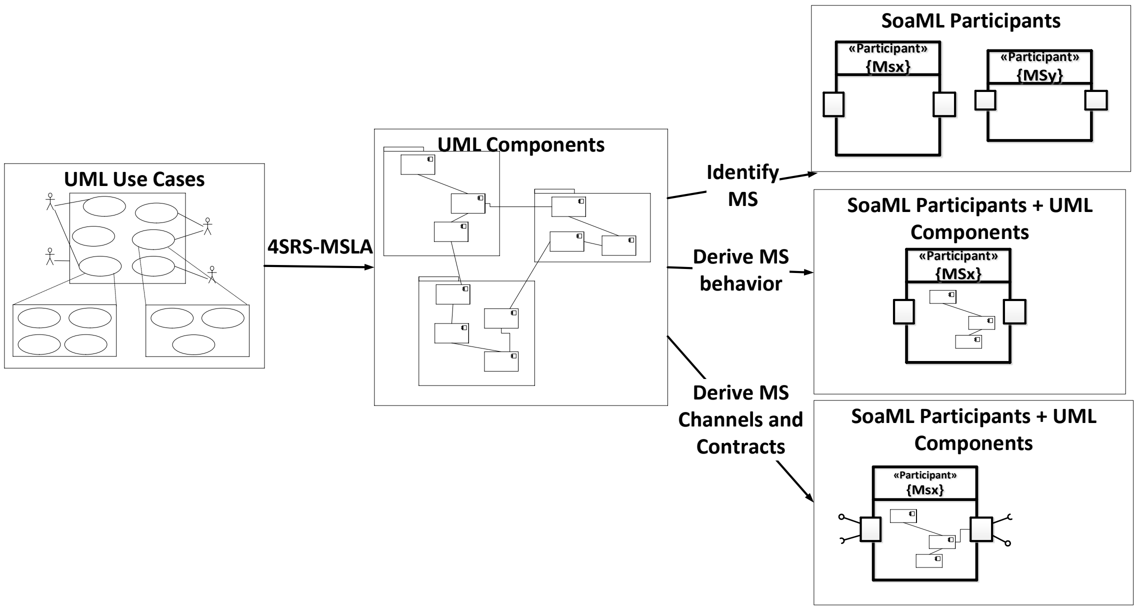


Figure 90. Specifying microservices using 4SRS-MSLA

The aim for using the 4SRS-MSLA is to have a logical view of the microservices' internal behavior and communications, so that all the elicited functional requirements are met in the derived solution. The four steps of the 4SRS-MSLA are the following:

Step 1. Components Creation

The first step regards the creation of three components, where the 4SRS-MSLA method associates, for each use case, a component for interface with users or systems (*i-type*), a component for the data model (*d-type*), and a component for logic/control of the microservice domain (*c-type*).

Step 2. Components Elimination

In the second step, components are submitted to elimination tasks. In previous versions of the method, the redundancy identification often includes components that are (functionally) similar but with different usage, which result in eliminating redundant components but defining a wider representation for the retained component. This often occurs within *c-* or *i-types* components. Nevertheless, the microservices principles suggest that the microservice has only one specific purpose, hence one may suggest that a component should be eliminated only if its purpose is exactly the same as of another one, and thus not eliminating any of them if their purpose is just similar.

Step 3. Component Packaging / Microservice Identification

The third step consists in grouping a set of components in packages, which further compose higher-level microservices. In 4SRS-MSLA, packaging is based in the use cases model obtained in the first-level refinement. Components, regardless of their category (*i-*, *d-*, or *c-type*), are assigned to one package (higher-level microservice) based in the process they relate to, or based in the non-leaf use case (that includes the leaf) originally derived from. Such packaging assures that the DDD pattern is followed.

Step 4. Microservices Associations

The associations between components are then generalized in order for depicting the associations between microservices. In a microservices context, these associations relate to service channels that exist in order to allow communication between microservices to support a given business process or information flow. This view is intended for identifying the need for such channels, regardless of the communication Pattern adopted, i.e., messaging between services or use of middleware such as API Gateways or lightweight message bus. Identifying such associations is based in descriptions from use cases (dependencies between functionalities at user requirements level), as well as from the components themselves, during the execution of step 2.

In this section, the inputs from the derived UML models by performing AMPLA and the 4SRS-MSLA are used to model the SoaML diagrams and their components. The modeling so far allows deriving the microservices' internal behavior, their data models, and the existing communications. These different concerns are included in different SoaML diagrams, in form of transition rules. These rules are grouped in 'Boundary', 'Data' and 'communication', as depicted in **Table 21**.

Table 21. Transition from UML (within AMPLA) to SoaML

Rule	Input from UML	Output in SoaML
1. Boundary	UML Packages	Service Participants
2. Boundary	UML Packages	Service Architecture diagram
3. Boundary	UML Components (within Packages)	Service Capabilities (methods)
4. Boundary	i-types	Separate web apps from Service Participants
1. Data	d-types	Service Capabilities
1. Communication	4SRS (associations)	Service Participants (Requests/Services and Ports)
		Service Interfaces
2. Communication	UML Sequences	Service Interfaces

Each microservice identified within the 4SRS-MSLA method execution is represented as a Service Participant. Thus, the set of Service Participants compose the microservices architecture. The required invocations for the Participant (Figure 91) were identified based in the use case description, where the interactions with other use cases were previously described. Additionally, the same interactions allowed identifying the need for methods that call those services and the properties (data) within the Capabilities.

It is during Step 2 of the 4SRS-MSLA that it is defined the expected behavior of the microservice. In order to align with typical composing layers of a microservice (UI, API, Logic and database), this approach proposes maintaining a general purpose description, but also the inclusion of HTTP verbs under which that component is called (used for defining «request» ports), the invocation of HTTP verbs required to consume services that are necessary in order to fulfill its purpose (used for defining «service» ports), and the properties that compose the dedicated database of the microservice (input for Service Data).

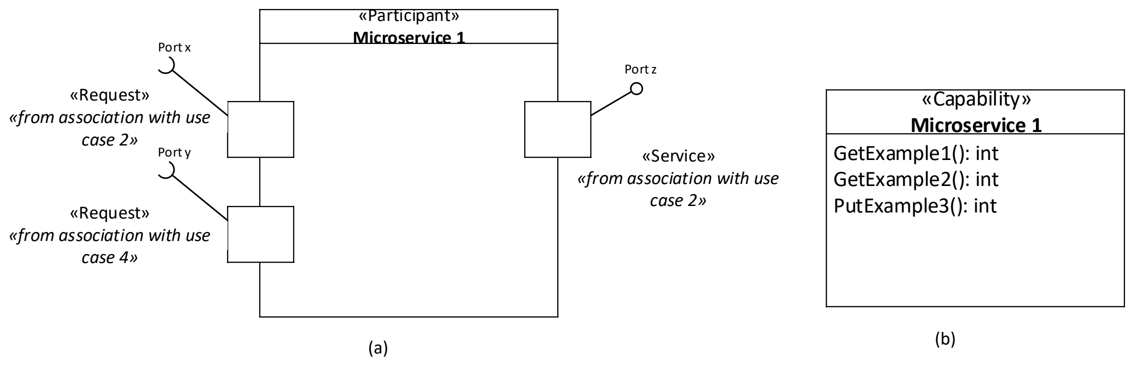


Figure 91. Participant with ports, interfaces and capabilities (methods/properties)

In Step 3 of the 4SRS-MSLA, it is common that *i-type* components that relate to user interface (UI) actions are grouped together into one or more *i-type* components. This occurs because these components are typically part of a web application rather than a given consumed service.

Data management

In Step 3 of the 4SRS-MSLA, *d-types* may also be grouped if the goal is to centralize the data, as in the "*shared database*" pattern. Alternatively, they may be included in the package from the higher-level microservice they relate. This decision results in including within the microservice *d-type* components that are responsible for the related data access, which reflects an application of a "*database per service*".

If the solution uses the shared database pattern, the MSLA is likely to have a dedicated package for *d-type* components, which must be assured when performing Step 3 of 4SRS, *i.e.*, assigning a package to *d-type* components. This package is not transformed into a microservice, but rather is remained as a dedicated package (just like the UI package for web apps). If the solution uses the database per service pattern, *d-types* are assigned in Step 3 to a given service, *i.e.*, any package except for the UI. Additional patterns are then followed, like "*API Composition*", "*Command Query Responsibility Segregation*" (CQRS) and *Saga*. These patterns are out of the scope, but will be discussed in future research.

Inter-service communication

Defining inter-service communication is very complex during specifications, as some information about communication needs (parameters, formats, protocols, etc.) are not always clear during specification tasks. This section proposes defining such communication needs, by using inputs that may come from the 4SRS-MSLA method execution as from the sequence diagrams exercise (Figure 89). In terms of modeling, SoaML diagrams able to be used are the ones such as Service Architecture, Service Interface and Service Channels.

From the 4SRS-MSLA, in Step 4 defining microservices associations should follow some constraints in order to prevent ineffective communication". Figure 92 represents the associations and rules that this step has to follow for proper component association. On the left side, are represented direct associations between components within the same sub-domain (*i.e.*, *i-*, *c-* and *d-type* components derived from the same use case), and, on the right side, the associations derived from use case dependencies. In terms of the required association rules, five rules are mainly applied. On the left side, if the three components are maintained, *i-type* should associate with *c-type*, and *c-type* associate with *d-type(s)* (Figure 92a). Next, associations with the ones exemplified on the right side should be assigned only to *c-types* (Figure 92a, b and c). Use case-related associations between *i-* or *d-types* should only occur if any *c-types* were not maintained (Figure 92d). Finally, by applying these rules, there may be a case where only *d-types* were maintained. In this case, an analysis by the architect is required, since *d-types* usually respond (in CRUD actions towards data) to another component's call.

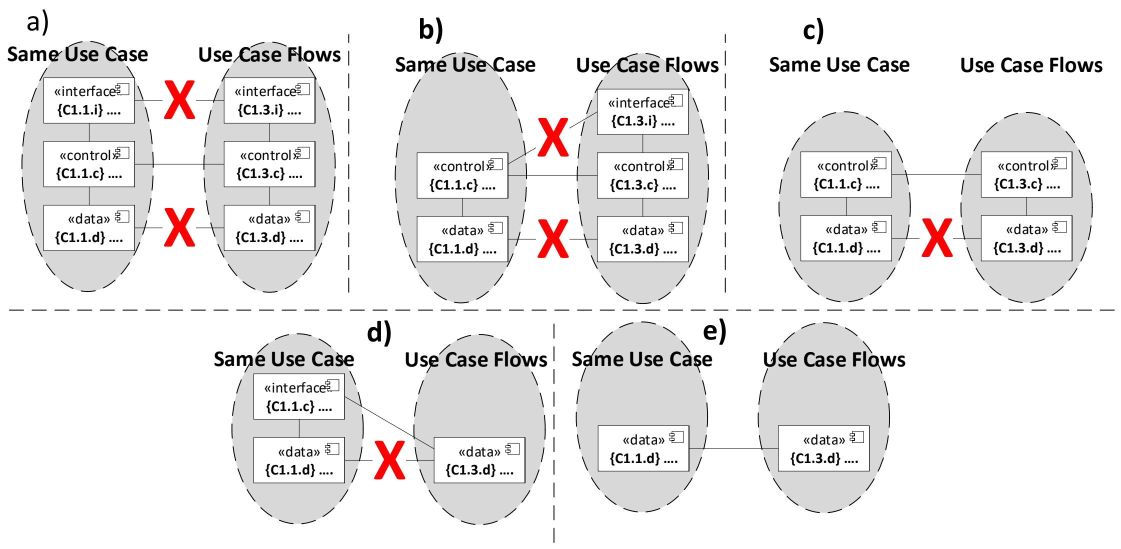


Figure 92. Defining associations between components

In order to implement API Composition, modeling refers to the microservice's response to a given process, which is derived from the associations from Step 4 and depicted in Service

Participant's service ports. Additionally, identifying needs for implementing CQRS refers to the dependencies between microservices, using the Service Architecture (Figure 93). It should be referred that both patterns are typically used only in "*database per service*" settings.

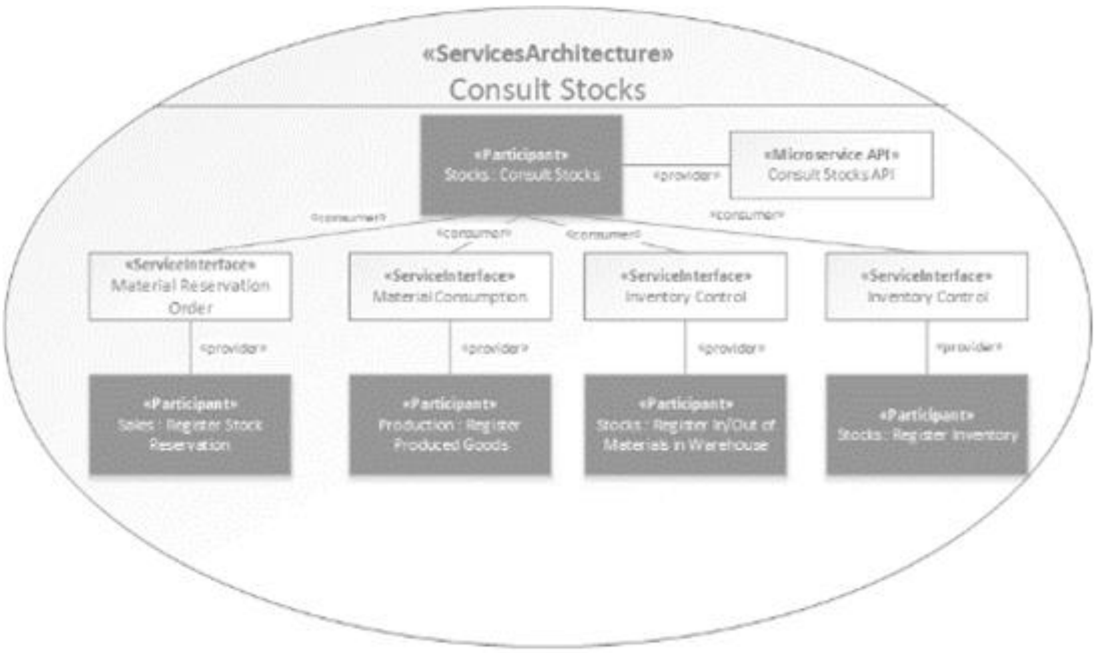

«ServicesArchitecture»
Consult Stocks
«Participant» Stocks : Consult Stocks
«provider»
«Microservice API» Consult Stocks API
«consumer»
«consumer»
«consumer»
«consumer»
«ServiceInterface» Material Reservation Order
«ServiceInterface» Material Consumption
«ServiceInterface» Inventory Control
«ServiceInterface» Inventory Control
«provider»
«provider»
«provider»
«provider»
«Participant» Sales : Register Stock Reservation
«Participant» Production : Register Produced Goods
«Participant» Stocks : Register In/Out of Materials in Warehouse
«Participant» Stocks : Register Inventory

Figure 93. Service Architecture

The approach for a given communication pattern (API gateways, remote procedure invocation, messaging or a domain-specific protocol, etc.) is not yet defined in MSLA, rather it only defines the necessity of existence of a flow between microservices. However, any design decisions on adopting a given pattern may be directly included in the components specification, the ServiceChannels, or in Service Interface diagram (Figure 94). For these interfaces, besides defining the parameters of the exchanged data,

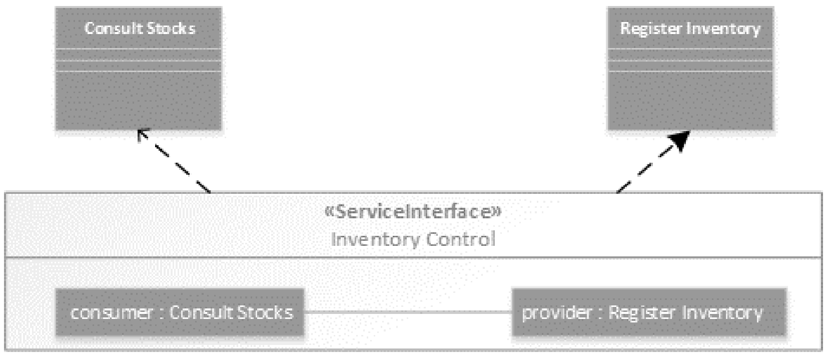


Figure 94. Service Interface

the design decisions rely in whether the communication is synchronous or asynchronous. This decision will then support the protocol for brokerage to be used (e.g., REST and gRPC for synchronous, or MQTT, AMQP, OPC-UA or Kafka for asynchronous).

Automatization

The agility provided by a microservices architecture mainly provides from having an infrastructure that supports a proper continuous integration/deployment (CI/CD) pipeline of the microservices to a production environment.

For this particular goal, obstacles refer to maximizing as possible CI/CD of the microservices, namely by performing a set of tests to the microservice. For this purpose, modeling may only provide some guidance on the expected behavior of the microservice.

Components and associations are the required input for performing several types of testing, from unit to acceptance testing. Additionally, component testing is enabled by validating the microservice behavior as described its composing components. Service integration contract testing is enabled by validating the scenarios where services invoke other services. These invokes are represented as ServiceChannels by the associations described in Step 4. Diagrams such as Service Contracts or even UML Sequence diagrams enable the contract validation.

5.4. Demonstration Cases

Architecture Spikes: the iFloW case

By revisiting what was discussed in the previous chapter about the iFloW project, it should be pointed out that iFloW is an R&D project between a University of Minho (UMinho) and Bosch

Car Multimedia Portugal (Bosch) consortium, that aims at developing an integrated logistics software system for inbound supply chain traceability (cf. Chapter 1). The main goal of the project is to develop a tracking platform that is backed by integrating information from freight forwarders and on-vehicle GPS devices. Main functionalities regard the control of raw material flow from Asian and European suppliers, and deviation of Estimated Time of Arrival (ETA) value.

The elements from UMinho had no previous knowledge of the domain (in this case, logistics) or of the involved technologies, such as third-party service providers, GPS, EPCIS or SAP-OER. Thud, besides performing upfront gathering and documentation of organization's (logistics-related) activities, to define terms, and to analyze flows, legacy software and data, Scrum Sprints often included performing spikes. Spikes, originally defined within XP, are a technique used for activities such as research, innovation, design, investigation and prototyping. Additionally, the team also performed architectural spikes, whenever required a deeper understanding of a design decision.

Within the project, there was a concern of analyzing the performed software engineering disciplines throughout the Sprints. In this research, we use the terminology from RUP's disciplines (only for demonstration purposes) to depicts the type of effort involved.

In all Sprints, the need for updates to the logical architecture was assessed (within the Analysis & Design). Afterwards, the typical disciplines were carried out within the Sprints: Implementation, Testing and Deploy.

The use of spikes in the iFloW project justifies the inclusion of the Requirements discipline in the each Sprint, as shown in Figure 95. These spikes were, in their majority, originated from middleware-based use cases (for instance, related to integration with third-party service providers, GPS, EPCIS or SAP-OER).

Within the remaining use cases, the Requirements discipline was not required. Thus, in comparison with the disciplines included in Figure 95, the Sprint performed the remaining disciplines like illustrated with exception of Requirements. In fact, it is what indeed occurs in typical Scrum process (where almost every requirements-related effort is performed before Sprint cycles, like Sprint 0 or similar).

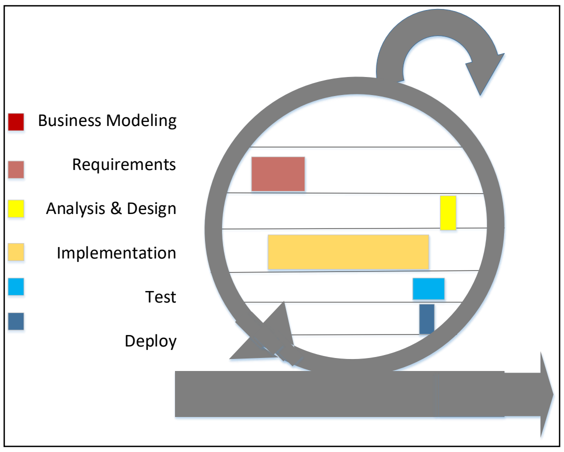


Figure 95. The performed disciplines within the Sprints

At a given point in time, both Bosch and UMinho identified the need for refactoring the code and the architecture of the system, namely to cope with security and standardization issues. Such refactoring led to a pause in the implementation tasks. The software logical architecture was revisited and the impacts were analyzed. Some design-oriented spikes (similar to architectural spikes from XP) were conducted, which then followed the re-design of the architecture. In this case, there was a focus in Analysis & Design instead of Implementation (see Figure 96, where it is detailed the sixth Sprint. Similarly to the other Sprints, this effort also lasted four weeks. This effort was required in this case but it may occur, or not, in any project.

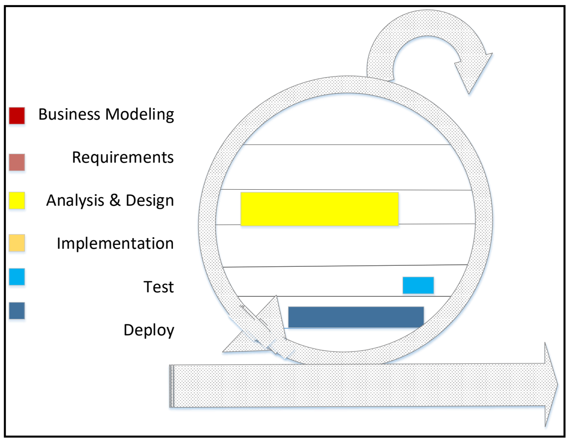


Figure 96. The performed disciplines within the architectural spike Sprint

Agile logical architecting: the UH4SP case

The UH4SP project is the demonstration case that encompasses the modeling result of the agile architecting lifecycle (AAL) presented in Section 5.2, which analyzes the evolution of an architecture for a software initiative throughout SDLC stages, from an enterprise level to the deployment of software components. The UH4SP artifacts presented in this section relate to the three main stages of AAL: Grooming, Backlog and Delivery. During these stages, the architecture evolves within different viewpoints, which are directly related between them (Concepts, Information Systems, Software Systems and Infrastructure).

The Conceptual architecture, due to the I4.0 nature of the project, relied in identifying concerns aligned with Industrial reference models like Industrial Internet Reference Architecture (IIRA) and *Industrie 4.0* Reference Architecture Model (RAMI 4.0). By analyzing its layers, for instance within IIRA, the management of corporate-level production, tools for collaborative processes within the supply chain and the microservices architecture refer to the Business layer, and the production data at the industrial unit level are acquired from a Manufacturing Execution System (MES), at the Operations layer.

This separation reflected the intended adoption of layers relating to business management, intermediate management at a cloud layer, and industrial local management at an edge layer. Such adoption was reflected in terms of the reference models for the Reference Architecture, by adopting NIST Cloud Computing Reference Architecture (NIST-CCRA) for the cloud layer and OpenFog Reference Architecture for the edge layer.

The information system architecture (Figure 97) is thus based in the same separation. At the industrial physical space level (D), operations take place and the interaction between the various actors and the system is verified through the various interface devices. It is at this level that operational information is generated to support the services to be made available by the system. At an intermediate level (C), typically located at the edges for each industrial unit, distributed capabilities, namely related to computation, networking, and storage and offered. At the cloud level (B), a service-oriented architecture is deployed to support horizontal functionality integration. Finally, at the top-level (A), business apps, either desktop web apps or mobile web apps, are the main interfaces with human actors. They use the cloud services to execute their processes.

For the logical architecture design, the functional requirements for supporting the business processes under the information system were elicited, since a logical architecture is an abstracted view of a system supporting functional requirements. The requirements analysis in the UH4SP project included gathering the requirements from a set of "to-be" scenarios and modeling a set of functional decomposed UML use cases. The Use Case model was composed by 37 use cases. This modeling work was based in the DUARTE approach, previously discussed in Chapter 4.

After executing the 4SRS method (Annex C), the logical architecture components (exemplified in Figure 98 and zoomed in Annex C) was derived with 77 architectural components. The components were grouped into five major packages, namely: *P1 Configurations*, *P2 Monitoring*, *P3 Business management*, *P4 UH4SP integration*, *P5 UH4SP fog data*. The logical architecture diagram was then used to specify microservices, responsible for retrieving production data from local industrial units. The

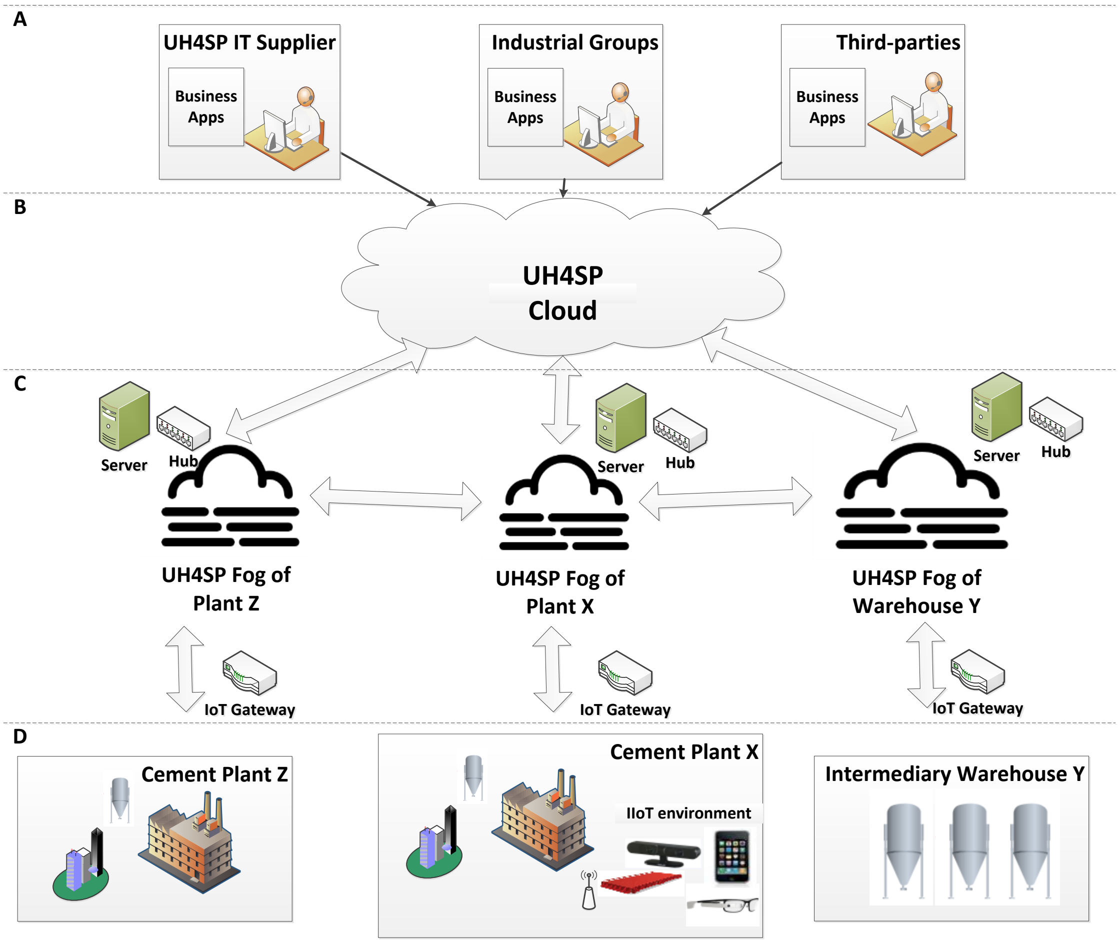


Figure 97. UH4SP information systems architecture

logical architecture is discussed in Chapter 5. This architecture was afterwards divided in a set of modules to be assigned to each of the project's teams (Figure 99).

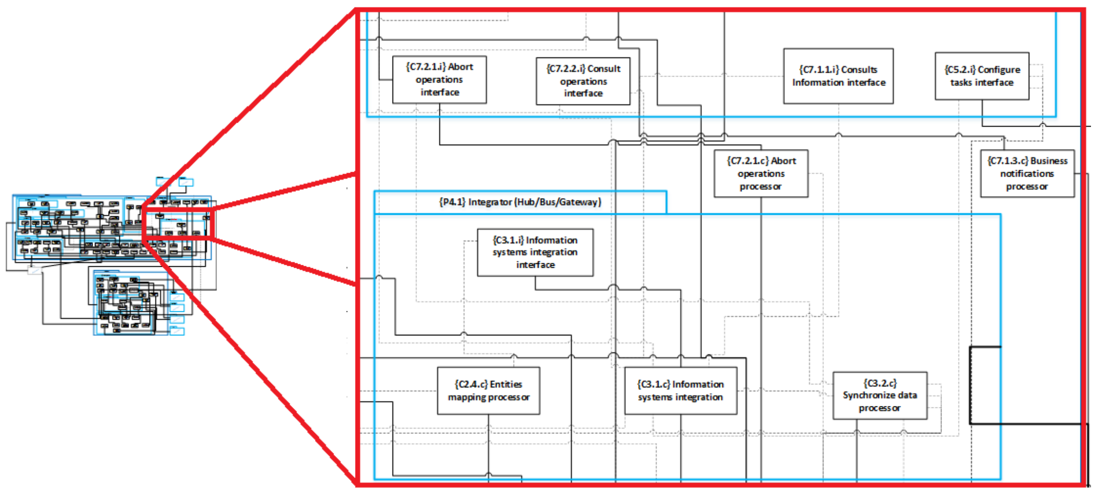

{C7.2.1.i} Abort operations interface
{C7.2.2.i} Consult operations interface
{C7.1.1.i} Consults Information interface
{C5.2.i} Configure tasks interface
{C7.2.1.c} Abort operations processor
{C7.1.3.c} Business notifications processor
{P4.1} Integrator (Hub/Bus/Gateway)
{C3.1.i} Information systems integration interface
{C2.4.c} Entities mapping processor
{C3.1.c} Information systems integration
{C3.2.c} Synchronize data processor

Figure 98. UH4SP logical architecture derived after 4SRS execution

The modularization depicted in Figure 99 originated five modules/subsystems, each assigned for 'Team A', 'Team B', 'Team C', 'Team D' and 'Team E'. The bordering was based in the contributions that each team brings to the consortium, namely IoT, cloud infrastructure, cloud applications and sensors/embedded systems. Each border is thus a part of a complex ecosystem and the dependencies are depicted in Figure 99.

'Team B' was the focus of this research. During the first "Just-enough" modeling, there were 37 use cases and 77 architectural components after performing the 4SRS method. After modularizing, 11 use cases from the 37, and the 15 components from the 77, compose the module under analysis. Finally, the requirements refinement output 29 use cases, *i.e.*, 18 refined functionalities, which then derived 37 architectural components from the 4SRS method. Having in mind the large-scale and complex ecosystem context of the project, these values may be perceived as acceptable. It is an increase of almost three times as the original models.

The refinement was performed incrementally and in parallel with team's Sprints (using Scrum). The requirements were refined, modeled and validated with the consortium, and only afterwards were input for the 4SRS method. The requirements validation was thus iterative as well. In UH4SP, after modeling in UML Use Cases, the requirements package was also composed with wireframes, to enrich the discussion. Additionally, in Scrum's Sprint Retrospective ceremonies, these models were object of feedback, and, if applicable, missing requirements were included. These validations were crucial in the project to enable a complete team buy-in.

After the 4SRS method, those five use cases derived 11 components. The MVP was implemented at the end of these Sprints, which was composed by 94 components. These components supported the project's pilot scenarios. However, next releases in order to follow the product roadmap are to be developed.

Each modularization may be refined. Flow between components (including components from different modules) are validated by modeling some processes. Dependencies can be depicted using, e.g., *A-type sequence diagrams*, namely some functionalities that must be implemented and executable in order for other functionalities to proper execute. In fact, this variant sequence diagrams pare powerful tools for bordering the modules, as well as validating (not just the modules but as well the whole) architecture.

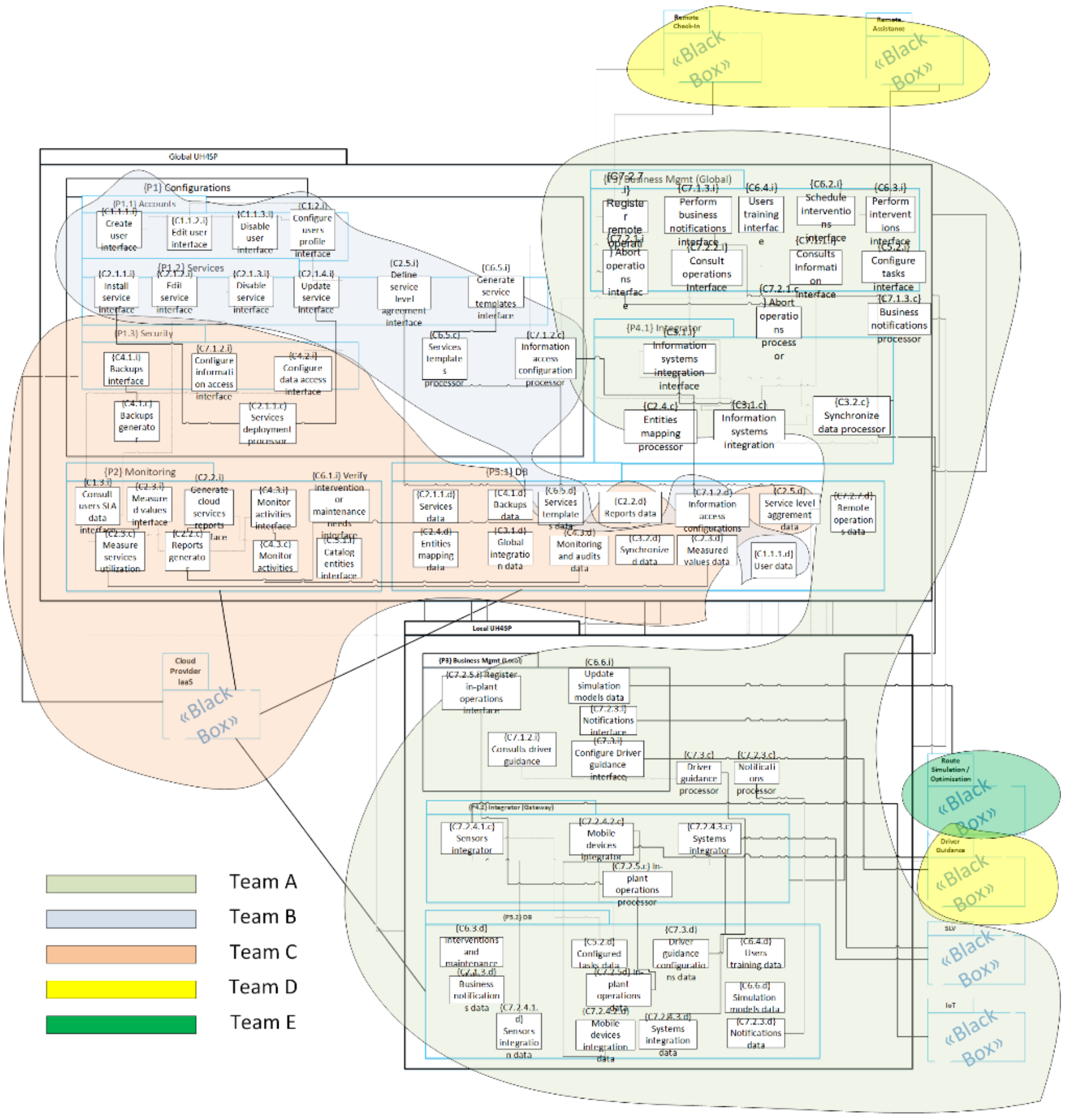


Figure 99. The modularization of the logical architecture

It must be assured that a module has composing software components that, together, deliver working software. Only if the set of components are able to deliver working software, it is also possible to perform acceptance testing and, afterwards, integration testing within the code deployment.

We applied filtering and collapsing techniques to the border that relates to the components within the module. These techniques redefine the system boundaries, which now regards only the given module as a subsystem to be designed. During the filtering process, all entities not directly connected to the module must be removed from the resulting filtered diagram.

Inside the system border defined for the given module through the respective coverage, the components were maintained as originally characterized. The components with direct connections to the module are maintained, and the ones without direct connection are removed. Figure 100 depicts one of the UH4SP sub-systems, namely the one composed with fog related functionalities.

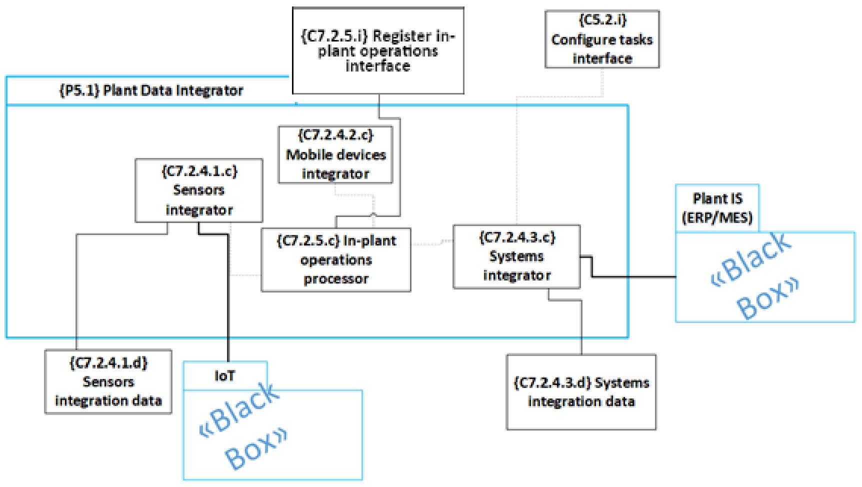

{C7.2.5.i} Register in-plant operations interface
{C5.2.i} Configure tasks interface
{P5.1} Plant Data Integrator
{C7.2.4.2.c} Mobile devices integrator
{C7.2.4.1.c} Sensors integrator
{C7.2.5.c} In-plant operations processor
{C7.2.4.3.c} Systems integrator
Plant IS (ERP/MES)
«Black Box»
{C7.2.4.1.d} Sensors integration data
IoT
«Black Box»
{C7.2.4.3.d} Systems integration data

Figure 100. UH4SP sub-system

The transition from the software system logical architectural diagram to the service use case diagram is performed by applying defined rules (Machado, Fernandes, Monteiro, & Rodrigues, 2006). An inbound (software) component is transformed into a (service) use case of the same type. By inbound we mean that the element belongs to the partition under analysis. On the other hand, an outbound (software) component is transformed into an actor, representing an external software component that interacts with the (service) use case, for instance, through messaging or APIs. The service use case diagram are depicted in Figure 101.

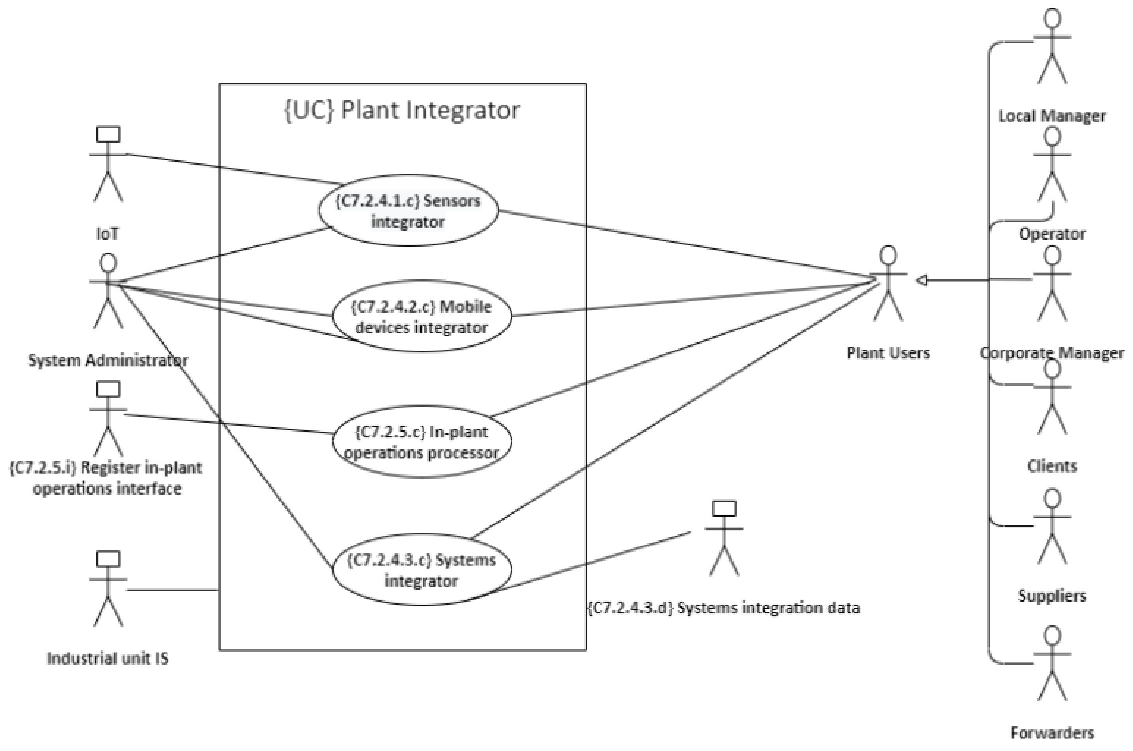

{UC} Plant Integrator
IoT
System Administrator
{C7.2.5.i} Register in-plant operations interface
Industrial unit IS
{C7.2.4.1.c} Sensors integrator
{C7.2.4.2.c} Mobile devices integrator
{C7.2.5.c} In-plant operations processor
{C7.2.4.3.c} Systems integrator
{C7.2.4.3.d} Systems integration data
Plant Users
Local Manager
Operator
Corporate Manager
Clients
Suppliers
Forwarders

Figure 101. Refined use cases resulting from the model transformation

The use cases are then used as input for a recursive execution of the 4SRS (Machado et al., 2006). The recursive execution of the 4SRS method is composed with the same steps as the previous execution, with the difference that at this point the addressed requirements relate to refined functionalities. This section describes how to use the 4SRS within the design of the microservices architecture. With that purpose, the 4SRS is used to derive services based in use cases. The front-end apps used within the monitor and configuration of the fog infrastructure, namely its computation and storage, is not described since it is out of the presented partition. In this case, the 4SRS must derive typical software components rather than software services.

Each microservice identified within the 4SRS method execution is represented as a Service Participant. Thus, the set of Service Participants compose the microservices architecture. The required invocations for the Participant (Figure 102) were identified based in the use case description, where the interactions with other use cases were previously described. Additionally, the same interactions allowed identifying the need for methods that call those services and the properties (data) within the Capabilities.

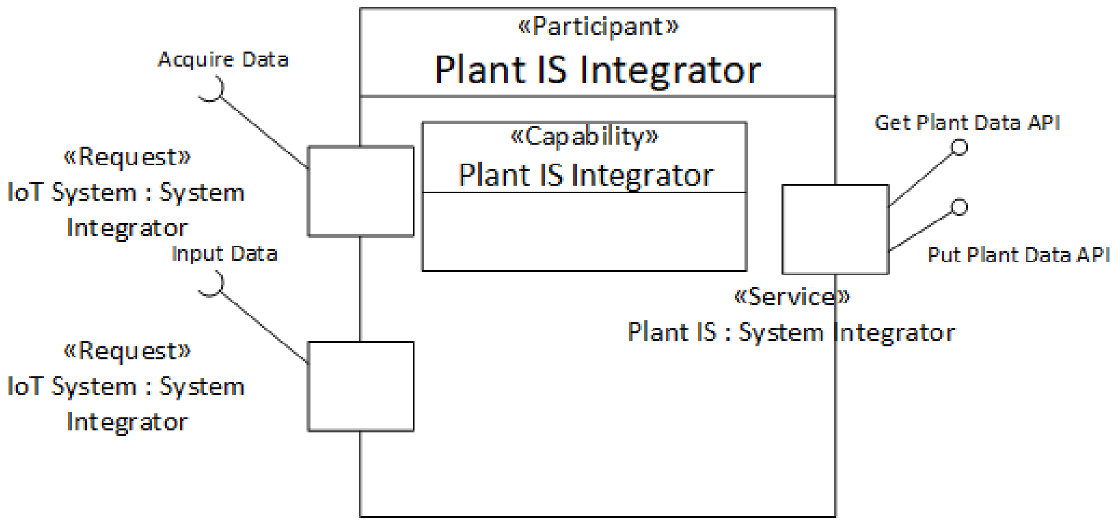


Figure 102. Participant with ports, interfaces and capabilities (methods/properties)

The Capabilities of a given service are applicable for a Database per service scenario. On the other hand, Capabilities of services that include representation (micro-step 2.v) as well as the services with association (further in Step 4) are applicable for a Shared database scenario.

A given ServiceChannel is related to a service associations from the 4SRS (step 4). They allow that services can consume or provide other services. Within SoaML diagrams, these associations provide input for the Service Interface or Contracts between services, and a Service Architecture with the definition of provided and consumed services (Figure 103).

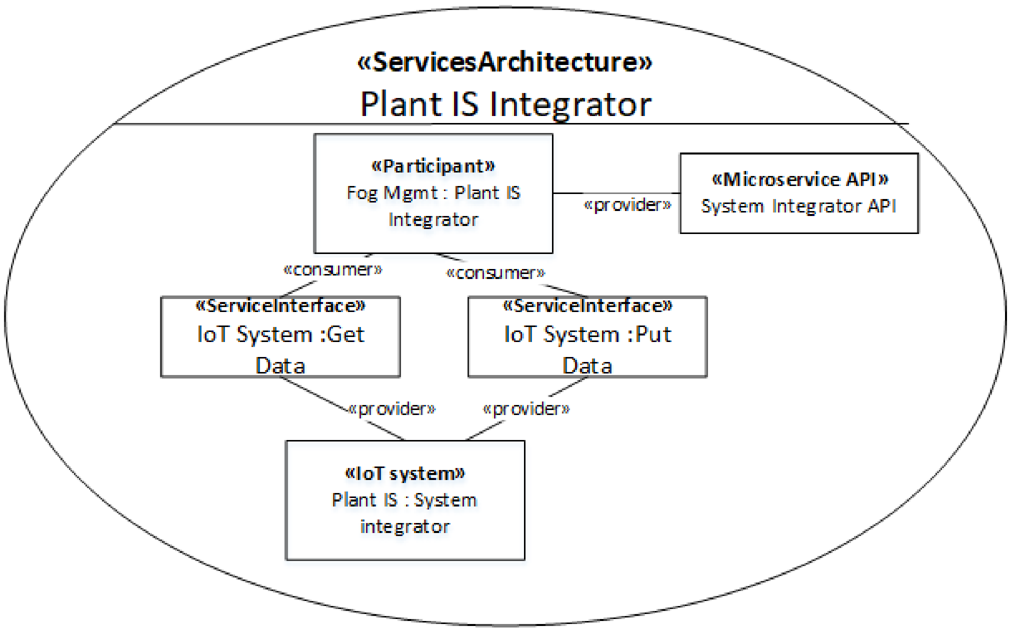


Figure 103. Service Architecture

The Service Architecture includes the other services that are consumed. Additionally, microservices are consumed by exposing an API. Hence, it is not specified in this diagram any particular service but rather the «*Microservice API*» stereotype of service interface (the specification of such extension to the metamodel is out of the scope of this thesis).

This API is also referred in the Participant diagram, which supports the service ports *Get Plant Data API* and *Put Plant Data API*. The remaining consumed services are represented in the request port.

The Service Interface diagram (Figure 104) refers to one of the interfaces included in the Service Architecture. It shall be noted that the use of tools such as API Gateway or Enterprise Service Bus and the representation of the associations between services in such scenarios are out of the scope of this thesis.

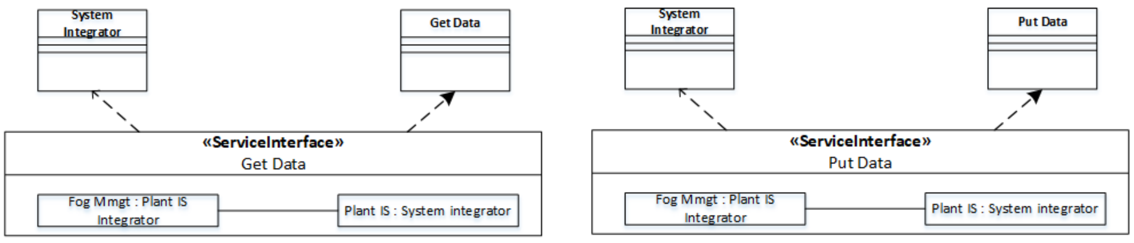

System Integrator
Get Data
«ServiceInterface»
Get Data
Fog Mmgt : Plant IS Integrator
Plant IS : System integrator
System Integrator
Put Data
«ServiceInterface»
Put Data
Fog Mmgt : Plant IS Integrator
Plant IS : System integrator

Figure 104. Service Interface

Finally, the deployment of the functionalities were also addressed. For that reason, the Deployment Architecture depicts the deployment location of the applications. The deployment architecture for the UH4SP project is depicted in Figure 105: (a) UH4SP business apps layer, either desktop web apps or mobile web apps, are the main interfaces with human actors; (b) UH4SP integration layer, located at the edges for each industrial unit, distributed capabilities, namely related to computation, networking, and storage; (c) UH4SP cloud services layer, a microservice-oriented architecture; and (d) local industrial unit system layer, where operational information is generated.

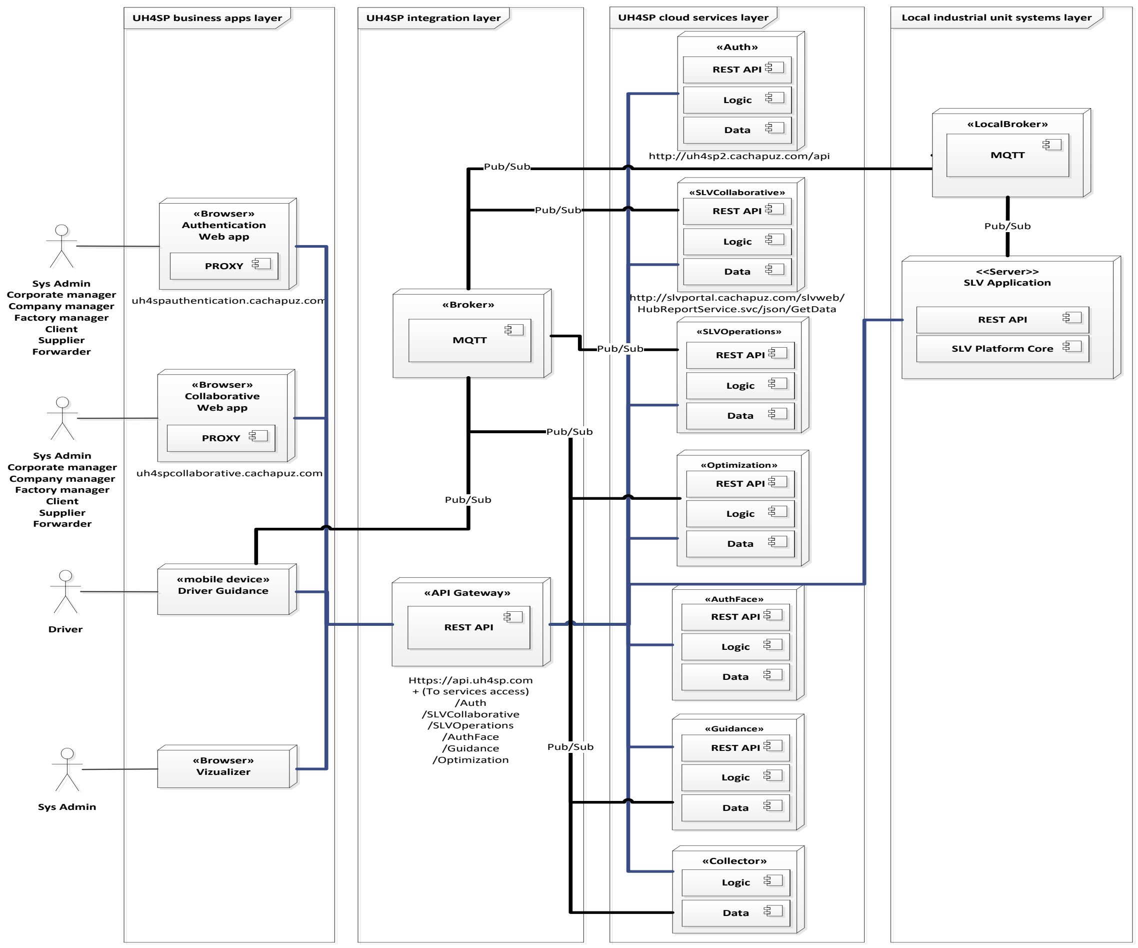


Figure 105. UH4SP deployment diagram

A cloud infrastructure is responsible for supporting the deployment of UH4SP services, the required computing, virtualization and storage needs. Developing the infrastructure required for meeting the deployment needs included infrastructure tools for assuring communication between industrial units and cloud services and between bussiness apps and the cloud services layer, but also for promoting the continuous integration and delivery of the microservices architecture. Such infrastructure is composed by a set of software tools, depicted in Table 22.

Table 22. Deployment setting for UH4SP

Infrastructure/deployment	Used software tool
Deployment	Virtual machines
Monitoring	Kibana
Factory gateway	Node-Red + Mosquitto
Cloud gateway	Mosquitto
Service discovery	Fluentd
Load balancer	Elasticsearch
Cloud resource management	Terraform
Cloud provider	Microsoft Azure
Configurations management	Ansible
API documentation	Swagger

The 4SRS-MSLA in brownfield projects: the IMP_4.0 case

The IMP_4.0 platform enables a software-house, *F3M – Information Systems, SA,* to optimize the development process for delivering solutions to their customers with tools to support all their decision-making processes. The solution is based on public and private clouds, which are interoperable with devices in an IoT and Cyber-Physical Systems (CPS) approach.

The IMP_4.0 project is about an Enterprise Resource Planning (ERP) system for the textile production domain, where the focus is to support milling, weaving and clothing processes, by providing a set of reusable and integrated modules. Additionally, the platform's development includes establishment of generic modules and variability management for enabling its extension to textile, footwear, cutlery, metal-mechanic, glassware and other sectors.

In terms of development, the IMP_4.0 project included developing:

- a set of management ERP-based features for the manufacturing sector to be delivered to customers cloud-based;
- microservices for process execution;
- shopfloor software services for manufacturing processes, *e.g.*, the control of the production lines, instructions for cutting machines.

The research is conducted within an F3M's software team. The team was composed by one Product Manager, which owned the business vision, a team of four software architects and four analysts, that modeled requirements and executed the 4SRS-MSLA method, and two development teams, responsible for implementing the resulting microservice architecture. The architects and analysts also performed the measurements within this research.

Within the IMP_4.0 project, all software product management, from the identification of market needs, asset identification and releases management, was based on an initial domain engineering, where it was intended to characterize the processes of the spinning, textile and garment domains, in terms of commonalities and domain variabilities.

The software requirements for the identified processes resulted in a specification of UML Use Cases. The Use Case model was composed by the following use cases, related to the ERP's modules (Figure 106), namely Stocks; Sales; Purchases; Production; Planning; Outsourcing; Quality Control; Packing list; Finances; and Stakeholder Management. Additional use cases related to integration with cloud infrastructures were also included.

Each use case was refined in functional decomposed use cases, resulting in 86 low-level (also called leaf) use cases, *i.e.*, the ones that could not be further divided. The decomposition process ended when leaf use cases represented basic CRUD operations. The functional decomposition shows that the 10 bounded contexts were structured within the tree's "branches" (Annex D). For representative purposes, Figure 107 depicts the functional decomposition related to Stocks.

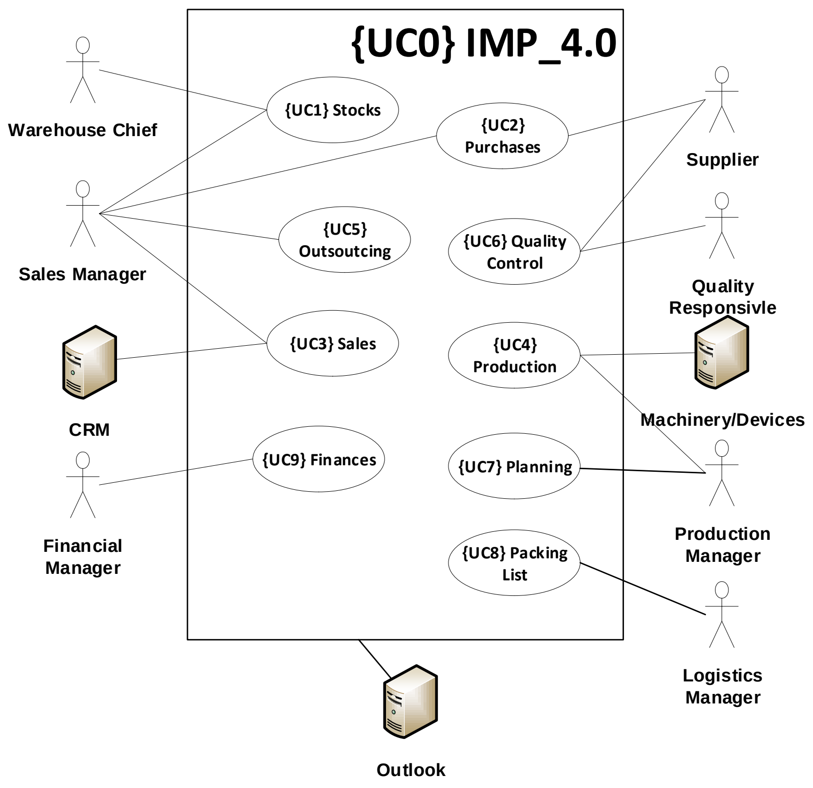


Figure 106. Use Case model of IMP_4.0 project

The next phase related to the execution of the 4SRS method. The 86 leaf-use cases were used as input, which allowed deriving 140 components after creating (Step 1) and eliminating/maintaining the components (Step 2). Next, component packaging (Step 3) formalized the identification of the microservices within the architecture, as well as their interface between web applications (ERP modules) and the MSA.

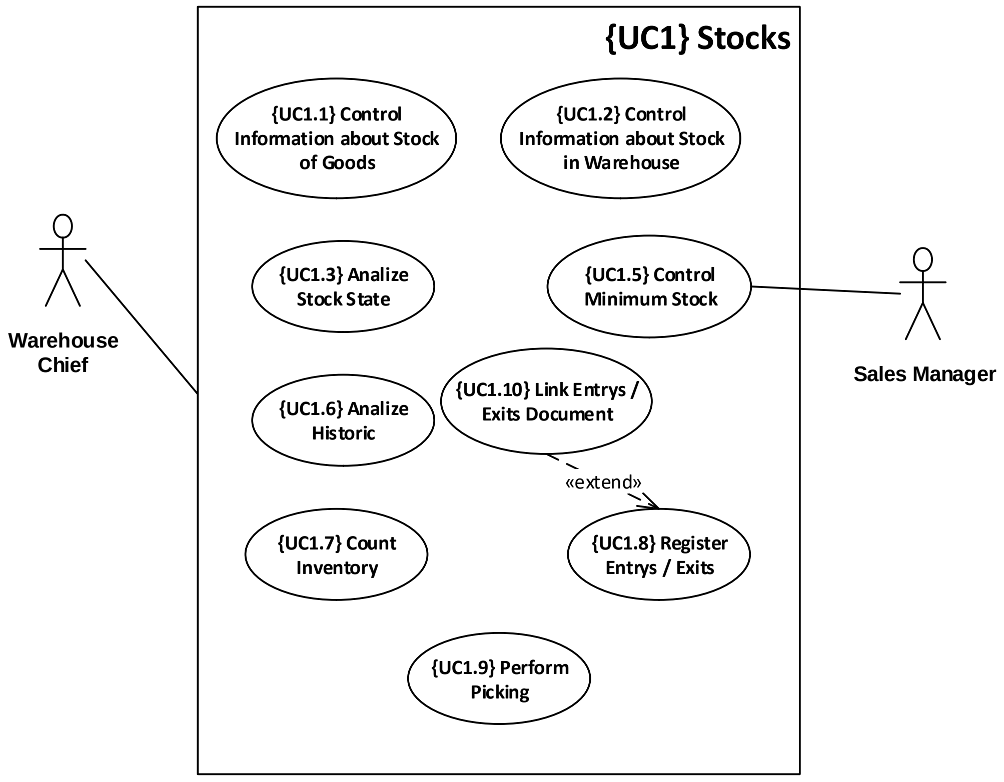

{UC1} Stocks
{UC1.1} Control Information about Stock of Goods
{UC1.2} Control Information about Stock in Warehouse
{UC1.3} Analize Stock State
{UC1.5} Control Minimum Stock
{UC1.6} Analize Historic
{UC1.10} Link Entrys / Exits Document
«extend»
{UC1.7} Count Inventory
{UC1.8} Register Entrys / Exits
{UC1.9} Perform Picking
Warehouse Chief
Sales Manager

Figure 107. Use case refinement of {UC1} Stocks

Based in the first-level of use cases, the MSLA is composed by the following microservices: *Stocks*; *Sales*; *Purchases*; *Production Orders*; *Bills of Materials*; *Planning*; *Outsourcing*; *Quality Control*; *Packing list*; *Checking Accounts*; *Banking*; and *Stakeholder Management*. Step 4 allowed identifying the required flows between the derived microservices. The general microservice identification and inter-service communication result, by collapsing the components, is depicted in Figure 108, while Figure 109 depicts the specific component behavior for the particular Stocks microservice.

From the initial 86 use cases, the application of the 4SRS-MSLA allowed to specify 140 components, from where the IMP_4.0 project MSLA derived to 12 microservices. It would be very difficult to design an architecture composed with such detail on components without the support of an architectural method such as the 4SRS.

In terms of MSLA implementation, at the time of this research, two parallel teams already had developed 6 microservices, of the 12 resulting from the 4SRS-MSLA, and their corresponding web applications. Namely, the deployed microservices are *Stocks* (*{P1}*), *Purchases* (*{P2}*), *Sales* (*{P3}*); *Production Orders* (*{P4.1}*), *Bills of Materials*

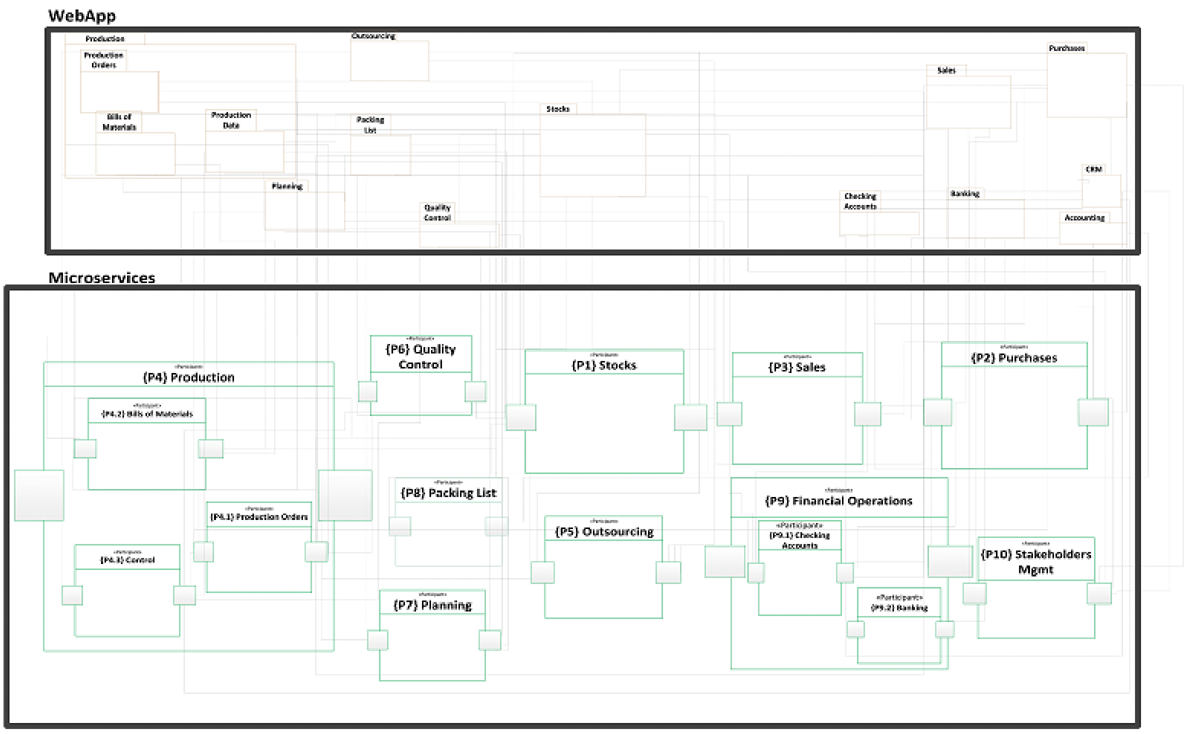


Figure 108. IMP_4.0 MSLA overview (with collapsed components)

(*{P4.2}*), and *Checking Accounts* (*{P9.1}*). As teams were developing each bounded context (microservice and web app), the domains were iteratively validated and, whenever a domain was implemented, their communication was also validated so the business processes were incrementally supported.

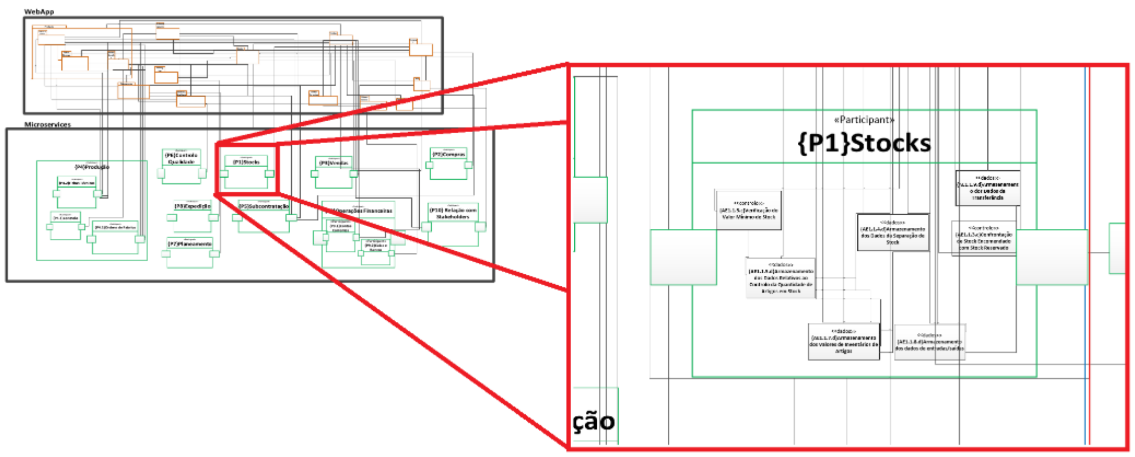


Figure 109. The IMP_4.0 Stocks microservice

Microservices deployment using the 4SRS-MSLA: the IMSPM case

The last demonstration case is different from the remaining, whereas it is an internal project at a software company called i2S Insurance Knowledge, S.A., located in Porto, Portugal. One of the applications used in i2S, called Internal Management System of Project Management (IMSPM), in recent years has had to increase new functionalities, which has led to the application to grow from two modules (customer management and proposal management) to six. This increase in functionality was accompanied by an increase in the load on the system, number of users and number of accesses.

In project management it has been found that there are several tasks during the execution of this process that are performed by different people in the organization. For this representation a use-case diagram was used, identifying the different actors that intervene in the process. The use case of Figure 110 represents an external view of the system and graphically the actors associated with the use of the application.

Additionally, as part of the artifacts included in the AMPLA process, an *A-type sequence diagram* was modeled, referring to the business processes and the dependencies among them that compose the ISMPM problem domain. In addition, the diagrams depicted in Figure 111 allowed an upfront understanding of how the microservices were going to communicate.

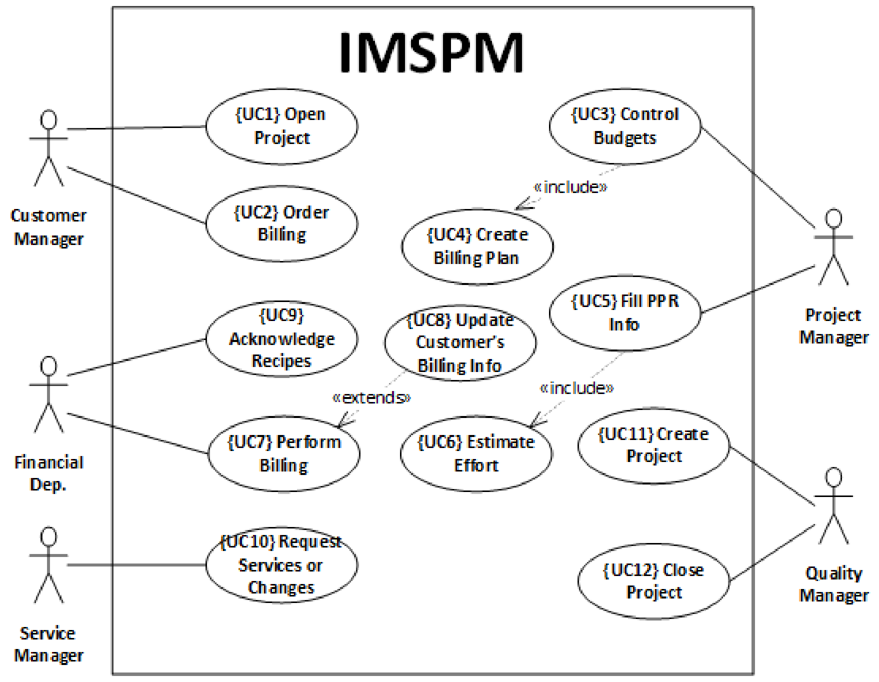


Figure 110. IMSPM Use cases diagram

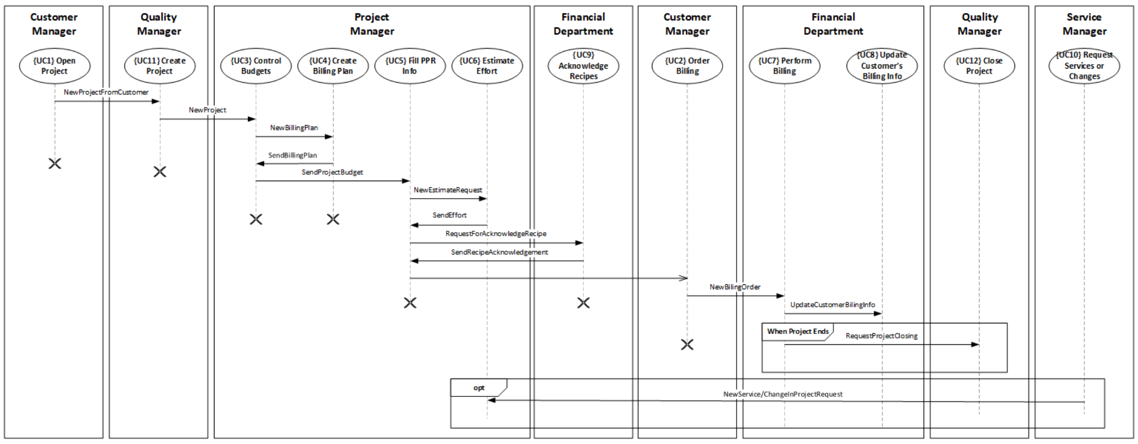
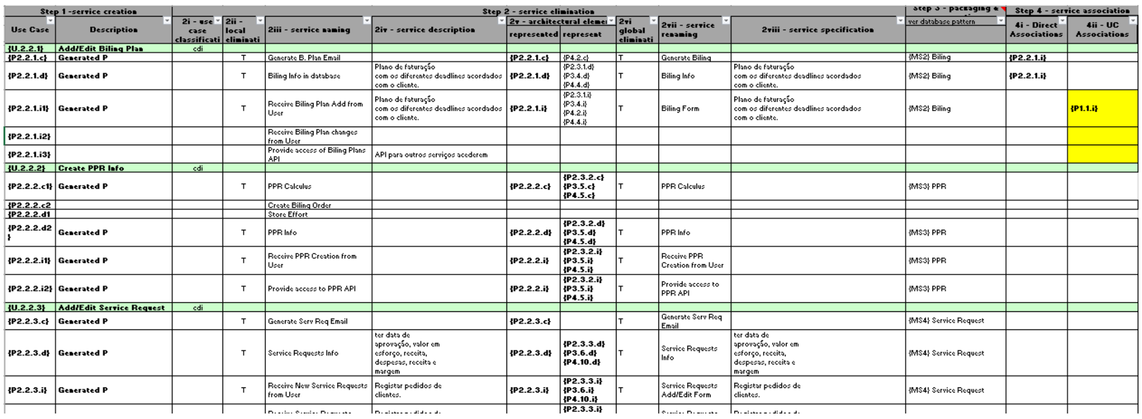


Figure 111. IMSPM Sequence diagram

After performing the 4SRS-MSLA (Figure 112) to the modeled use cases, the MSLA was derived. A zoomed version of the 4SRS table is depicted in Annex E. Each use case relates to a business domain, thus each one resulted in a microservice. Then, the 4SRS-MSLA was used to model the components included within the microservices. Thus, the microservices are Project Opening, Billing, PPR, Service Request, Change Request, Project Forecast, Sage Invoice Integration, Budgets, Resource Forecast, Customer Billing, and Project Closure. A simplified view of the microservices, without providing component information from each microservices, is depicted in Figure 113. Annex E also includes a zoomed version of the MSLA.



Step 1 -service creation		Step 2 - service elimination									Step 3 - packaging &	Step 4 - service association	
Use Case	Description	2i - use case classificati	2ii - local eliminati	2iii - service naming	2iv - service description	2v - architectural eleme represented	represent	2vi global eliminati	2vii - service renaming	2viii - service specification	ver database pattern	4i - Direct Associations	4ii - UC Associations
{U.2.2.1}	Add/Edit Biling Plan	cdi											
{P2.2.1.c}	Generated P		T	Generate B. Plan Email		{P2.2.1.c}	{P4.2.c}	T	Generate Biling		{MS2} Biling	{P2.2.1.i}	
{P2.2.1.d}	Generated P		T	Biling Info in database	Plano de faturação com os diferentes deadlines acordados com o cliente.	{P2.2.1.d}	{P2.3.1.d} {P3.4.d} {P4.4.d}	T	Biling Info	Plano de faturação com os diferentes deadlines acordados com o cliente.	{MS2} Biling	{P2.2.1.i}	
{P2.2.1.i1}	Generated P		T	Receive Biling Plan Add from User	Plano de faturação com os diferentes deadlines acordados com o cliente.	{P2.2.1.i}	{P2.3.1.i} {P3.4.i} {P4.2.i} {P4.4.i}	T	Biling Form	Plano de faturação com os diferentes deadlines acordados com o cliente.	{MS2} Biling		{P1.1.i}
{P2.2.1.i2}				Receive Biling Plan changes from User									
{P2.2.1.i3}				Provide access of Biling Plans API	API para outros serviços acederem								
{U.2.2.2}	Create PPR Info	cdi											
{P2.2.2.c1}	Generated P		T	PPR Calculus		{P2.2.2.c}	{P2.3.2.c} {P3.5.c} {P4.5.c}	T	PPR Calculus		{MS3} PPR		
{P2.2.2.c2				Create Biling Order									
{P2.2.2.d1				Store Effort									
{P2.2.2.d2 }	Generated P		T	PPR Info		{P2.2.2.d}	{P2.3.2.d} {P3.5.d} {P4.5.d}	T	PPR Info		{MS3} PPR		
{P2.2.2.i1}	Generated P		T	Receive PPR Creation from User		{P2.2.2.i}	{P2.3.2.i} {P3.5.i} {P4.5.i}	T	Receive PPR Creation from User		{MS3} PPR		
{P2.2.2.i2}	Generated P		T	Provide access to PPR API		{P2.2.2.i}	{P2.3.2.i} {P3.5.i} {P4.5.i}	T	Provide access to PPR API		{MS3} PPR		
{U.2.2.3}	Add/Edit Service Request	cdi											
{P2.2.3.c}	Generated P		T	Generate Serv Req Email		{P2.2.3.c}		T	Generate Serv Req Email		{MS4} Service Request		
{P2.2.3.d}	Generated P		T	Service Requests Info	ter data de aprovação, valor em esforço, receita, despesas, receita e margem	{P2.2.3.d}	{P2.3.3.d} {P3.6.d} {P4.10.d}	T	Service Requests Info	ter data de aprovação, valor em esforço, receita, despesas, receita e margem	{MS4} Service Request		
{P2.2.3.i}	Generated P		T	Receive New Service Requests from User	Registar pedidos de clientes.	{P2.2.3.i}	{P2.3.3.i} {P3.6.i} {P4.10.i}	T	Service Requests Add/Edit Form	Registar pedidos de clientes.	{MS4} Service Request		
							{P2.3.3.i}						

Figure 112. 4SRS method execution within ISMPM project

As described in Section 5.2, the logical viewpoint should be complemented with additional architectural views. In this project, the logical viewpoint was design together with the deployment viewpoint, which allowed specifying the infrastructure that would support the microservices architecture execution and deployment.

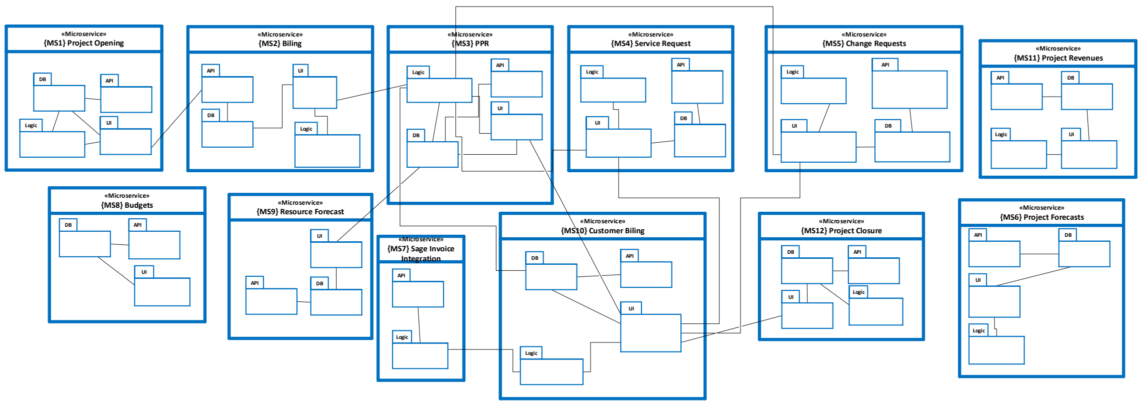

«Microservice» {MS1} Project Opening
«Microservice» {MS2} Biling
«Microservice» {MS3} PPR
«Microservice» {MS4} Service Request
«Microservice» {MS5} Change Requests
«Microservice» {MS11} Project Revenues
«Microservice» {MS8} Budgets
«Microservice» {MS9} Resource Forecast
«Microservice» {MS7} Sage Invoice Integration
«Microservice» {MS10} Customer Biling
«Microservice» {MS12} Project Closure
«Microservice» {MS6} Project Forecasts
DB
API
UI
Logic

Figure 113. IMSPM MSLA model

The deployment viewpoint is typically adopted in these kinds of projects, as mainly contributing to specify how the infrastructure will promote the automatization that the microservices architectures paradigm ease and that contribute to the "continuous" agenda of software development, which includes continuous testing, integration, deployment and delivery as well as DevOps.

Figure 114 depicts the deployment diagram for the IMSPM project. Since each microservice will be executed in one different (Docker) container, that layer refers to the designed components from the logical viewpoint, i.e., the MSLA. Remaining layers refer to applying other widely adopted microservices patterns – application patterns (namely, database per service, client-side UI composition), application infrastructure patterns (namely, messaging, circuit breaker) and infrastructure patterns (namely, service per container, service registry and API gateway). Such design was afterwards operationalized under a set of software tools, which compose the continuous delivery setting. This setting is depicted in Table 23.

Table 23. Deployment setting for IMSPM

Infrastructure/deployment	Used software tool
Deployment	Docker containers
Monitoring	Portainer tool
Version control	Gitlab
Continuous delivery	Jenkins
Inter-service communication	HTTP
API documentation	Swagger
API Gateway	Ocelot

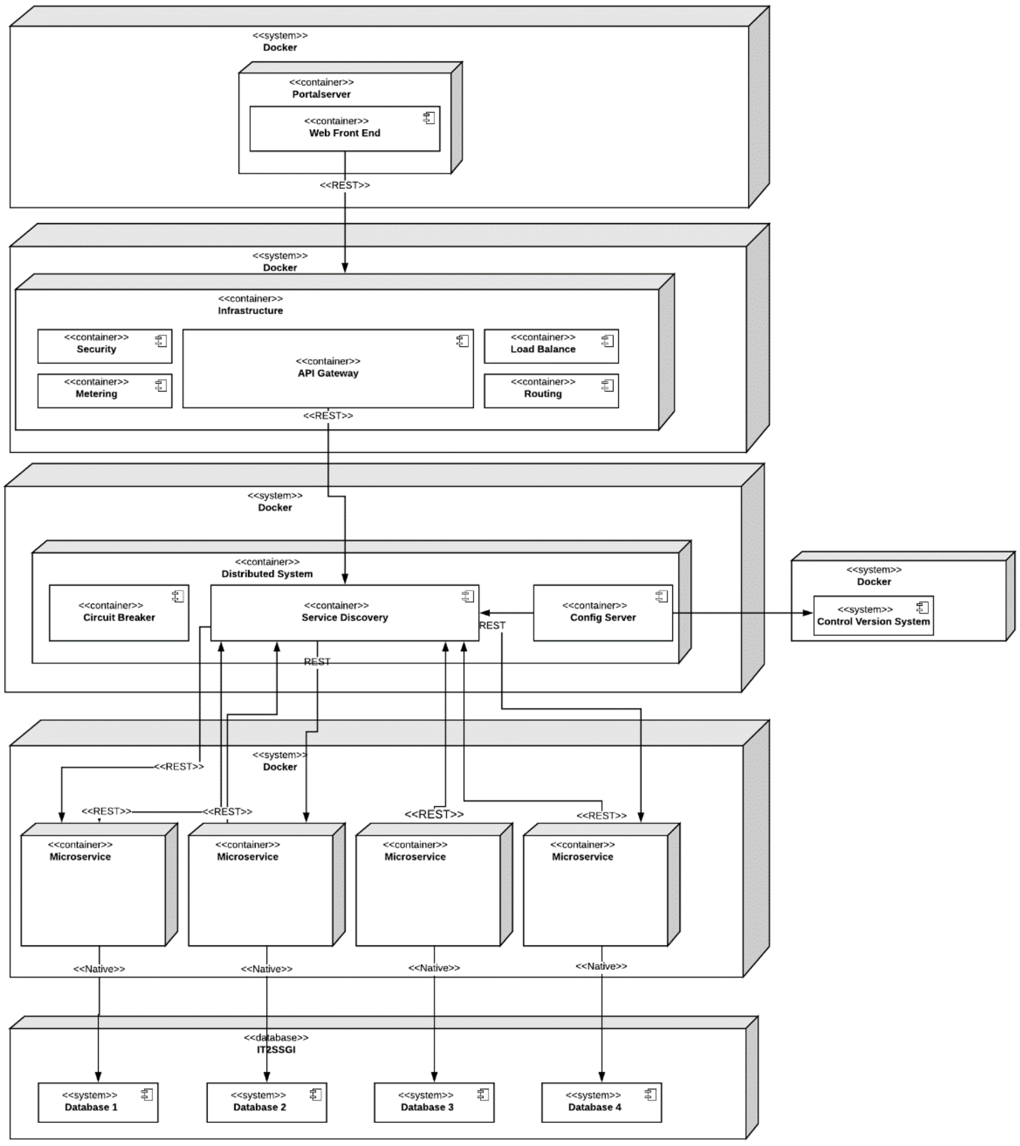
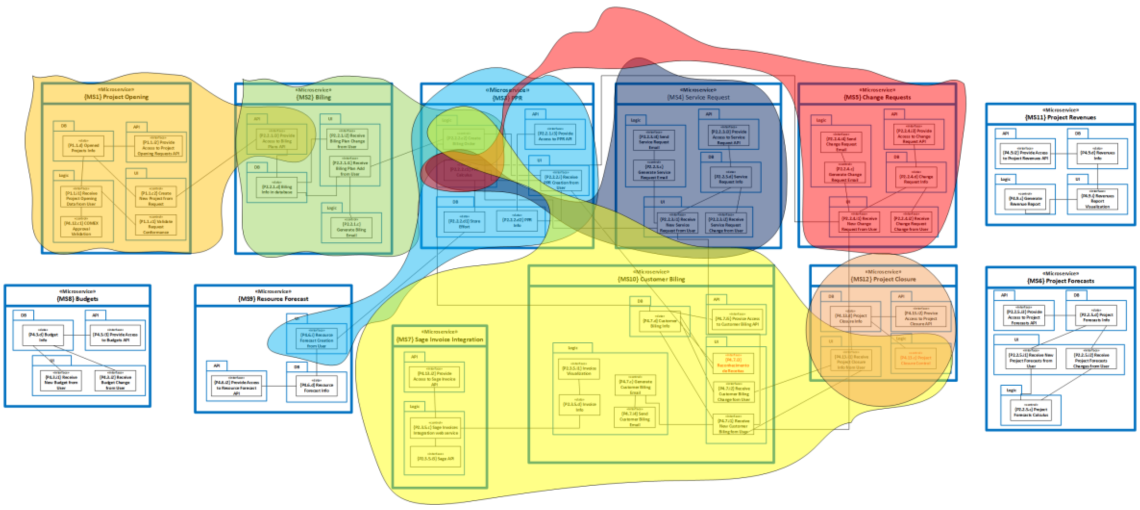


Figure 114. IMSPM Deployment diagram

Finally, it is worth referring that the MSLA is able to define the scope of development work for each microservice. Figure 115 depicts the scope of eight microservices (the remaining four were not defined as priority for now), referring that, for implementing a given microservice, the scope of the work refers to the components covered by a spot within Figure 115. Namely, the spot not only may cover the components from the microservice, but also, if applicable, cover work to be done related to service communication necessary for the microservice proper execution. Thus, if it is identified that a microservice needs to invoke another microservice for its own purpose, one or more components from another microservice (typically, the ones related to interfaces or APIs) must be covered by the spot. Only when all the components from the spot are implemented, it is possible to properly test the microservice and deploy it.



«Microservice» {MS1} Project Opening
«Microservice» {MS2} Billing
«Microservice» {MS3} PPR
«Microservice» {MS4} Service Request
«Microservice» {MS5} Change Requests
«Microservice» {MS11} Project Revenues
«Microservice» {MS8} Budgets
«Microservice» {MS9} Resource Forecast
«Microservice» {MS10} Customer Billing
«Microservice» {MS12} Project Closure
«Microservice» {MS6} Project Forecasts
«Microservice» {MS7} Sage Invoice Integration

Figure 115. Spots representation of the ISMPM MSLA model

Discussion

Candidate architecture design using the 4SRS method

Especially when new technological paradigms arise, stakeholders have many difficulties in eliciting technical design decisions. In opposition to waterfall-based frameworks, where all the requirements and design tasks are performed only upfront, within ASD projects these tasks should be performed continuously.

Starting from understanding the problem domain and existing references, the architect is able to specify functionalities and design a logical architecture. This logical architecture then allows defining a Backlog that scopes the software development in an ASD manner.

AMPLA provides a method for deriving a candidate logical architecture based in UML Use Cases, the 4SRS method. There is not any difference within the steps of the method, in comparison with the original method, to derive a candidate architecture. Rather, as the input are high-level requirements in opposition to more refined ones, one may experience difficulties in identifying a proper classification of the use case in order to decide the components to be maintained within the second step, since a more refined information helps in better define the component's nature. However, as in AMPLA the requirements will be later refined and will emerge, the 4SRS method is used as in a "living table" that is opened alongside the development Sprints, rather than a waterfall-based and one-time-execution approach, providing traceability between the requirements and the components in order to agilely respond to changes.

Incremental design for refining the logical architecture

In the UH4SP project, architecture design followed an incremental approach. Five teams composed the project. However, this research focused in only one team. After modularizing, 11 use cases from the 37, and the 15 components from the 77, compose the module under analysis. Finally, the requirements refinement output 29 use cases, *i.e.*, 18 refined functionalities, which then derived 37 architectural components from the 4SRS method. Having in mind the large-scale and complex ecosystem context of the project, these values may be perceived as acceptable. It is an increase of almost three times as the original models.

The refinement was performed incrementally and in parallel with team's Sprints (using Scrum). The requirements were refined, modeled and validated with the consortium, and only afterwards were input for the 4SRS method. The requirements validation was thus iterative as well.

After modeling in UML Use Cases, the requirements package was also composed with wireframes, to enrich the discussion. Additionally, in Scrum's Sprint Retrospective events, these models were object of feedback, and, if applicable, missing requirements were included. These validations were crucial in the project to enable a complete team buy-in.

Architecture evolution and management

The hybrid ASD performed in the iFloW project included a joint use of models within (Scrum) Sprints, combining UML Use Cases for requirements and UML Components for the logical architectural design. Besides providing an organization on the set of components, the logical architectural model was additionally used for depicting the relationships among components suggest dependencies that affect the implementation of functionalities and their inclusion in the further Sprint Backlogs (cf. Section 4.5).

During Sprints, the software logical architecture was revisited and the impacts were analyzed at the end of each iteration, in order to predict refactoring efforts. Additionally, when a change was identified, the logical architecture representation allowed analyzing which components are targeted with impacts from those changes.

Later, within the UH4SP project, the method for logical architecture assessing was revisited, namely with a focus in component traceability.

Defining a traceability method for supporting CIA for architectural decision-making from feature adding and changing arose many challenges. We believe that the model traceability between architecture and requirements strengthened AMPLA in a sense of architecture maintenance during ASD. However, implementing the method was a learning process, with advantages and disadvantages, which are detailed in this section.

Applying CIA within AMPLA allowed supporting architectural decision-making with the following advantages:

Identify architecture value proposed for change: is a feature is proposed for change, traceability to architectural models allows identifying if that change affects components that are ASR's, if the requirements relates to a high customer value feature or if it relates to an imposition from standards and policies compliance.

<u>Depicting dependencies</u>: revisiting the 4SRS method execution allows identifying dependencies with other components from architecture, which are affected by a given change

<u>Adding and refining requirements</u>: the decision to add or refine requirements - either derived from new needs from stakeholders or from technical debts - is eased by modeling the new requirements and derive the resulting new components by performing the 4SRS method.

CIA practices validation: in this research, the presented scenarios included using all the CIA practices proposed in section 5.3 (ASR, Quality characteristic, Business and customer value of the requirement, which components are affected, Compliance with standards, Requirements emerge, and Managing architectural debt).

However, applying the method faces some obstacles/disadvantages, namely:

Quantity of models for Sprint 0 and refinements: AMPLA proposes activities and artifacts for modeling (sequence, use cases, 4SRS and components diagrams) that are additional to typical work in Sprints, which may be perceived as a disadvantage. However, the method provides traceability between models, which eases agility of analyzing changes.

Need for revisiting a set of models within each change: whenever a change is proposed, CIA may result considering changes in more than one model (e.g., change use cases, then perform 4SRS and derive new logical architecture), which can be time consuming.

It also needs to be pointed out some threats to validity of this research, namely:

Method's learning curve: AMPLA was only applied by the method's designers, thus it was not possible to depict in the UH4SP project any possible data regarding the adoption of the method by other teams.

Team's involvement: it was difficult to commit the entire UH4SP consortium towards the approach. The candidate architecture was proposed and the sub-systems delivered to all entities, however it was not possible to assess CIA and possibly identify new change scenarios besides the one's our team addressed. Other teams used Scrum for their sub-systems but did not follow a refinement approach like AMPLA, thus the traceability was not assured.

Newly-formed team: the fact that it was a completely new development team and, mainly, first time implementing microservices architectures, may have led to situations that the team was unable to identify a need for changing in how a microservice should have been implemented and hence reporting such change during Sprint Reviews.

Identified practices: Although the scenarios refer to three different change needs - in fact other changes were proposed during the UH4SP project, but they fell under the scope of these three scenarios, thus for demonstration purposes they were already represented - it is impossible to assure that additional scenarios may occur in the future that require other CIA practices than the ones here proposed. The proposed CIA practices were based in the scenarios, and the literature review did not perceived the need for additional practices, however missing practices cannot be ruled out.

Microservice modeling in SoaML

The research in microservices architecture design using AMPLA encompassed the settings where projects from such nature, namely breaking legacy and monolith systems (brownfield) and greenfield projects. IMP_4.0 and IMSPM for brownfield and UH4SP for greenfield

The proposed 4SRS-MSLA method was used in a project setting where a software team from F3M aimed to break an existing monolith application to microservices, but struggling to define a proper strategy. Although the application was structured in modules, its processes were deeply dependent and data models between the modules were shared.

The application of the presented method had as main contributions to the F3M team in:

- *Identifying the required domains* for decomposing the existing F3M's monolith application into a set of microservices;
- Providing a modeling support for specifying the *intra-service behavior*, after identifying required *i-*, *c-* and *d-type* components required for the microservices;
- Depicting required separations with the *web app interfaces* and the *data models*, when packaging *i-* and *d-type* components; and
- Identifying *inter-service communication* and messaging that a microservice requires to perform its mission.

Additionally, it is also worth referring some additional advantages of the method reported during the IMP_4.0 project. Although not included explicitly in the method, the models were origin of some discussions about MSA development and deployment, as following.

- ***Database architecture***: the decisions regarding *d-type* components after performing Steps 3 and 4 of 4SRS-MSLA originated discussions about if the service had shared database with another service rather than a dedicated one. Deciding which package in Step 3 for a given *d-type*, or use case flow-related associations to *d-types*, were the main reasons for starting those discussions.
- ***Querying (API Composition and CQRS)***: The dependencies between microservices from the associations in Step 4 and depicted in Service Channels or service ports were input for discussions of patterns like *API Composition*, *Command Query Responsibility Segregation* (CQRS) and *Saga*.
- ***Testing:*** Component testing is enabled by validating the microservice behavior as described its composing components. Service integration contract testing is enabled by validating the

scenarios where services invoke other services. These invocations are represented as ServiceChannels by the associations described in Step 4. This testing and valuable for a proper continuous integration/deployment (CI/CD) pipeline.

- ***Communication style:*** The MSLA only defines the necessity of existence for a flow between microservices. However, any design decisions on adopting a given pattern (API gateways, remote procedure invocation, messaging or a domain-specific protocol, etc.) may be directly included in the components specification.

Nevertheless, the method still has its limitations. In terms of database partition, the derived model was still far from the required specifications on implementing distributed relational and non-relational databases, mainly regarding consistency problems.

In the UH4SP project, stakeholders elicit requirements regarding front-end functionalities, since they are more aware of the business and not so much of the technology. Hence, starting in modeling a logical architecture based in business requirements allowed using stakeholder inputs for an initial stage and afterwards refine the information necessary to specify the MSLA.

The UH4SP was composed by five teams, where each one was assigned to a module from the architecture. Since a module could have one or more microservices, this research allowed validating loosed development from different teams. Sequence diagrams were also useful for discussing and developing microservice communications that were developed by different teams.

As a disadvantage, the diagrams were only the starting point for developing and deploying the microservices. In terms of data management, inter-service communication, messaging/brokers, deployment and infrastructure, the diagrams do not provide still the necessary detail for implementing application, infrastructure application and infrastructure patterns (Richardson, 2018).

5.5. Conclusions

Agile Architecting Lifecycle (AAL) encompasses software design that evolves from architectural, mechanistic, and detailed design to development and deployment. The AAL pathway goes through three stages: Grooming, Backlog and Delivery. Throughout the stages, the architecture is designed under eleven viewpoints, grouped in categories of Concepts, Information Systems, Software Systems and Infrastructure. The viewpoints are more suited depending on the stage they are performed.

Grooming encompasses most architecture design. Starting from understanding the problem domain and existing references, the architect is able to specify functionalities and design a logical architecture. This logical architecture then allows defining a Backlog that scopes the software development in an ASD manner. As software increments are developed and the architecture emerges, the Delivery stage encompasses design at the deployment level.

Microservices architectures are seen with great advantages in software development, and especially for cloud applications. Although its advantages, teams struggle with properly designing, developing, and deploying this technology. Performing traditional techniques in software engineering, in terms of requirements modeling, is still far from providing a proper level of detail for developing microservices. However, there is room for specifying services based in the elicited requirements.

This research proposed defining a method for deriving a microservices logical architecture from functional requirements. The method has as input an UML logical components diagram, where domains (DDD) were identified within the architecture. That information was used for iterative refinement of the architecture, enabling deriving microservices specifications, afterwards modeled in SoaML diagrams, which are complementary for a proper specification of microservices behavior and associations, such as Service Participants and Contracts. The traceability associated to 4SRS-MSLA method assures an alignment between the initial Use Case model and the derived architecture proposed solution. The 4SRS-MSLA steps were adapted to meet widely known microservices characteristics.

This chapter discussed the *agile logical architecting* topic addressed within the AMPLA process. This research also included a discussion on describing agile logical architecting by proposing an architecture classification framework. With inputs from business needs and software requirements – using "Decomposing User Agile Requirements ArTEfacts" (DUARTE) - the candidate logical architecture is derived from the 4SRS, in form of a V-model-based model derivation. As key results for this topic, this chapter proposed:

- AAL pathway with three stages: Grooming, Backlog and Delivery.
- Adaptations of the 4SRS method for candidate logical architecture design contexts.

This chapter also addressed Backlog and Delivery stages of the AAL pathway, namely how the candidate architecture is incrementally refined throughout (Scrum) Sprints, aiming to provide required component specifications for software addressing from development teams. Such addressing during Sprints was also continuously managed, where this research proposed traceability mechanism for allowing an impact-analysis in aspects such as ASR, Quality characteristic, Business and customer value of the requirement, Which components are affected, Compliance with standards, Requirements emerge, and Managing architectural debt. The iterative refinement of the architecture is used for deriving microservices specifications, by proposing the 4SRS-MSLA, allowing to model SoaML diagrams. As key results for this topic, this chapter proposed:

- Architecture refinement within Sprints
- Architecture evolution and management, with techniques of change-impact analysis
- Modeling of microservices logical architectures (4SRS-MSLA)

For multiteam projects, in LSA settings, the candidate architecture derived from the first execution of the 4SRS is used for modularization. Chapter 6 discusses how architectural modules from AMPLA are used to assign work to multiteams and support the evolution of models and of the developed software through inter-team management and communication.

Further reading

The "Agile Software Architecture" from Muhammad Ali Babar, Alan W. Brown and Ivan Mistrik includes several contributions in the topic of agile architecting, ranging from architecture design, combination with agile methods and change-impact analysis (Ali Babar, Brown, & Mistrík, 2014).

Designing microservices architecture includes addressing the services' scope, database, inter-service communication, security, observability, among others. Kasun Indrasiri and Prabath Siriwardena discuss a plethora of concerns and technologies for different approaches on designing, developing and deploying microservices architectures (Indrasiri & Siriwardena, 2018). Additionally, Richardson presents patterns and anti-patterns on developing microservices (Richardson, 2018).

References

Abrahamsson, P., Babar, M. A., & Kruchten, P. (2010). Agility and architecture: Can they coexist? *IEEE Software*, *27*(2), 16–22. https://doi.org/10.1109/MS.2010.36

Ali Babar, M., Brown, A. W., & Mistrík, I. (2014). *Agile Software Architecture: Aligning Agile Processes and Software Architectures*. Elsevier.

Azevedo, S., Machado, R. J., Muthig, D., & Ribeiro, H. (2009). Refinement of Software Product Line Architectures through Recursive Modeling Techniques . (R. Meersman, P. Herrero, & T. Dillon, Eds.), *On the Move to Meaningful Internet Systems: OTM 2009 Workshops*. Springer Berlin / Heidelberg. https://doi.org/10.1007/978-3-642-05290-3_53

Azevedo, S., Machado, R., & Maciel, R. (2012). On the Use of Model Transformations for the Automation of the 4SRS Transition Method. In M. Bajec & J. Eder (Eds.), *Advanced Information Systems Engineering Workshops* (Vol. 112, pp. 249–264). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-31069-0_22

Beck, K., & Andres, C. (2004). *Extreme programming explained: embrace change*. Addison-Wesley Professional.

Bragança, A., & Machado, R. (2007). Automating mappings between use case diagrams and feature models for software product lines. *Software Product Line Conference, ...*.

Bragança, A., & Machado, R. (2009). A model-driven approach for the derivation of architectural requirements of software product lines. *Innovations in Systems and Software Engineering*, *5*(1), 65–78. https://doi.org/10.1007/s11334-009-0078-3

Bragança, A., & Machado, R. J. (2005). Deriving Software Product Line's Architectural Requirements from Use Cases: An Experimental Approach. In *2nd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES'05)* . Rennes, France.

Cockburn, A. (2006). *Agile software development: the cooperative game*. Pearson Education.

Conway, M. E. (1968). How Do Committees Invent? Datamation, 28–31.

Coplien, J. O., & Bjørnvig, G. (2011). *Lean architecture: for agile software development*. John Wiley & Sons.

Cruz, E., Machado, R., & Santos, M. (2014). From business process models to use case models: A systematic approach. *Advances in Enterprise Engineering …*. Retrieved from http://link.springer.com/chapter/10.1007/978-3-319-06505-2_12

Díaz, J., Pérez, J., & Garbajosa, J. (2014). Agile product-line architecting in practice: A case study in smart grids. *Information and Software Technology*, *56*(7), 727–748. https://doi.org/10.1016/j.infsof.2014.01.014

Douglass, B. (1999). *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns*. Addison-Wesley Professional.

Erder, M., & Pureur, P. (2015). Continuous architecture: Sustainable architecture in an agile and cloud-centric world.

Erdogmus, H. (2009). Architecture meets agility. *IEEE Software*, *26*(5), 2–4. https://doi.org/10.1109/MS.2009.121

Evans, E. (2004). Domain-driven design : tackling complexity in the heart of software. Addison-Wesley

Farhan, S., Tauseef, H., & Fahiem, M. A. (2009). Adding agility to architecture tradeoff analysis method for mapping on crystal. In *WRI World Congress on Software Engineering (WCSE'09) - Volume 04* (Vol. 4, pp. 121–125). IEEE. https://doi.org/10.1109/WCSE.2009.405

Fernandes, J. M., & Machado, R. J. (2016). *Requirements in Engineering Projects*. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-18597-2

Fernandes, J. M., Machado, R. J., Monteiro, P., & Rodrigues, H. (2006). A Demonstration Case on the Transformation of Software Architectures for Service Specification. In B. Kleinjohann, L. Kleinjohann, R. Machado, C. Pereira, & P. S. Thiagarajan (Eds.), *From Model-Driven Design to Resource Management for Distributed Embedded Systems* (Vol. 225, pp. 235–244). Springer US. https://doi.org/10.1007/978-0-387-39362-9_25

Ferreira, N., Santos, N., Machado, R., Fernandes, J. E., & Gasević, D. (2014). A V-Model Approach for Business Process Requirements Elicitation in Cloud Design. In A. Bouguettaya, Q. Z. Sheng, & F. Daniel (Eds.), *Advanced Web Services* (pp. 551–578). Springer New York. https://doi.org/10.1007/978-1-4614-7535-4_23

Ferreira, N., Santos, N., Machado, R. J., & Gasevic, D. (2012). Derivation of Process-Oriented

Logical Architectures: An Elicitation Approach for Cloud Design. (A. J. O. Dieste and N. Juristo, Ed.), *13th International Conference on Product-Focused Software Development and Process Improvement - PROFES 2012*. Madrid, Spain: Springer-Verlag, Berlin Heidelberg, Germany .

Fitzgerald, B., & Stol, K.-J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, *123*, 176–189. https://doi.org/10.1016/J.JSS.2015.06.063

Fowler, M. (2018). *Refactoring: improving the design of existing code (2nd edition)*. Addison-Wesley Professional.

George, M. L., & George, M. (2003). *Lean six sigma for service*. New York, NY: McGraw-Hill.

Harvick, R. (2012). *Agile Architecture for Service Oriented Component Driven Enterprises: Encouraging Rapid Application Development using Agile*. DataThunder Publishing. Retrieved from http://dl.acm.org/citation.cfm?id=2331400

Indrasiri, K., & Siriwardena, P. (2018). *Microservices for the enterprise: Designing, Developing, and Deploying*. Apress.

Jeon, S., Han, M., Lee, E., & Lee, K. (2011). Quality attribute driven agile development. In *9th International Conference on Software Engineering Research, Management and Applications (SERA)* (pp. 203–210). IEEE. https://doi.org/10.1109/SERA.2011.24

Kanwal, F., Junaid, K., & Fahiem, M. A. (2010). A hybrid software architecture evaluation method for fdd-an agile process model. In *International Conference on Computational Intelligence and Software Engineering (CiSE)* (pp. 1–5). IEEE. https://doi.org/10.1109/CISE.2010.5676863

Kazman, R. (2013). Foreword - Bringing the Two Together: Agile Architecting or Architecting for Agile? In *Agile Software Architecture: Aligning Agile Processes and Software Architectures* (pp. xxix–xxx). Elsevier.

Kruchten, P. (1995). The 4+1 View Model of Architecture. *IEEE Software*, *12*(6), 42–50. https://doi.org/10.1109/52.469759

Loukides, M. (2012). What is DevOps?

Machado, R. J. (2002). Heterogeneous Information Systems Integration: Organizations and Methodologies. In Springer (Ed.), *4th International Conference on Product Focused Software Process Improvement - PROFES'02*. Rovaniemi, Finland.

Machado, R. J., Fernandes, J. M., Monteiro, P., & Rodrigues, H. (2005). Transformation of UML Models for Service-Oriented Software Architectures. *Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*. IEEE Computer Society. https://doi.org/http://dx.doi.org/10.1109/ECBS.2005.73

Machado, R. J., Fernandes, J., Monteiro, P., & Rodrigues, H. (2006). Refinement of Software Architectures by Recursive Model Transformations. (J. Münch & M. Vierimaa, Eds.), *Product-Focused Software Process Improvement*. Springer Berlin / Heidelberg. https://doi.org/10.1007/11767718_38

Madison, J. (2010). Agile architecture interactions. *IEEE Software*, *27*(2), 41–48. https://doi.org/10.1109/MS.2010.35

Mancl, D., Fraser, S., Opdyke, B., Hadar, E., & Hadar, I. (2009). Architecture in an agile world. *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*. Orlando, Florida, USA: ACM. https://doi.org/10.1145/1639950.1639981

Martini, A., Pareto, L., & Bosch, J. (2014). Role of Architects in Agile Organizations. In J. Bosch (Ed.), *Continuous Software Engineering*. Springer Cham.

Moser, R., Abrahamsson, P., Pedrycz, W., Sillitti, A., & Succi, G. (2008). A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team. In *IFIP Central and East European Conference on Software Engineering Techniques* (pp. 252–266). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-85279-7_20

Neiva, R., Santos, N., Martins, J. C. C., & Machado, R. J. (2015). *Deriving UML logical architectures of traceability business processes based on a GS1 standard*. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 9158). https://doi.org/10.1007/978-3-319-21410-8_41

Newman, S. (2015). *Building microservices - Designing fine-grained systems*. O'Reilly Media, Inc.

Nord, R. L., & Tomayko, J. E. (2006). Software architecture-centric methods and agile development. *IEEE Software*, *23*(2), 47–53. https://doi.org/10.1109/MS.2006.54

OMG. (2012). Service Oriented Architecture Modeling Language™ (SoaML®). http://www.omg.org/spec/SoaML/.

Pahl, C., & Jamshidi, P. (2016). Microservices: A Systematic Mapping Study. In *6th International Conference on Cloud Computing and Services Science (CLOSER)* (Vol. 1, pp. 137–146). SCITEPRESS - Science and and Technology Publications. https://doi.org/10.5220/0005785501370146

Pérez, J., Díaz, J., Garbajosa, J., & Yagüe, A. (2014). Bridging User Stories and Software Architecture: A Tailored Scrum for Agile Architecting. In A. W. . Ali Babar, Muhammad ; Brown & I. Mistrik (Eds.), *Agile Software Architecture - Aligning Agile Processes and Software Architectures* (pp. 215–241). Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-407772-0.00008-3

Richardson, C. (2018). *Microservice Patterns* (1st ed.). Manning.

Ries, E. (2011). *The lean startup: How today's entrepreneurs use continuous innovation to create radically successful businesses*. Crown Books.

Salgado, C. E., Teixeira, J., Santos, N., Machado, R. J., & Maciel, R. S. P. (2015). A SoaML Approach for Derivation of a Process-Oriented Logical Architecture from Use Cases. *Exploring Services Science*, 80–94. Retrieved from http://link.springer.com/chapter/10.1007/978-3-319-14980-6_7

Santos, M. Y., & Machado, R. J. (2010). On the Derivation of Class Diagrams from Use Cases and Logical Software Architectures. In *2010 Fifth International Conference on Software Engineering Advances* (Vol. 0, pp. 107–113). Nice, France: IEEE. Retrieved from http://doi.ieeecomputersociety.org/10.1109/ICSEA.2010.24

Santos, N., Duarte, F. J., Machado, R. J., & Fernandes, J. M. (2013). *A transformation of business process models into software-executable models using MDA*. *Lecture Notes in Business Information Processing* (Vol. 133 LNBIP). https://doi.org/10.1007/978-3-642-35702-2_10

Schwaber, K. (1997). Scrum development process. In *Business Object Design and Implementation* (pp. 117–134). Springer. https://doi.org/10.1007/978-1-4471-0947-

1_11

Sharifloo, A. A., Saffarian, A. S., & Shams, F. (2008). Embedding architectural practices into Extreme Programming. In *9th Australian Conference on Software Engineering (ASWEC)* (pp. 310–319). IEEE. https://doi.org/10.1109/ASWEC.2008.4483219

Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. *IEEE Cloud Computing*, *4*(5), 22–32. https://doi.org/10.1109/MCC.2017.4250931

Thönes, J. (2015). Microservices. *IEEE Software*, *32*(1), 116–116. https://doi.org/10.1109/MS.2015.11

Waterman, M., Noble, J., & Allan, G. (2012). How much architecture? Reducing the up-front effort. In *AGILE India* (pp. 56–59). IEEE. https://doi.org/10.1109/AgileIndia.2012.11

Zhang, X., Hu, Y., Lu, Y., & Gu, J. (2011). University Dormitory Management System Based on Agile Development Architecture. In *International Conference on Management and Service Science*. IEEE. https://doi.org/10.1109/ICMSS.2011.5998992

Chapter 6 – Inter-team management within an LSA process based in logical architectures

This chapter discusses how the logical architecture model from the previous chapter is able to support teams' management in an LSA setting. The approach proposes an architecture modularization, where the modules encompass the work scope of different teams. Afterwards, the chapter describes how the models regarding the module are basis for inter-team management and communication. The same models are the starting point for deriving product backlogs, whether based in use cases, user stories, or other items. The chapter includes three demonstration cases, one regarding the use case-driven backlog case, and two regarding the user stories-driven backlog cases, as well as a discussion of the results. This chapter ends with the conclusions.

Chapter 6 – Inter-team management within an LSA process based in logical architectures

"*Depending on where you're looking,*
one person's system is another's subsystem"
Grady Booch*,* co-author of UML

6.1. Introduction

Digital transformation of businesses has increased the rate of creation of new software ecosystems. Additionally, software solutions allow third-party integration (e.g., using Application Programming Interfaces – APIs) towards full support of the supply chain. Many times this means that software development teams are no longer developing software "alone", rather cooperating with distributed teams belonging to other companies. While agile software development (ASD) has been adopted to optimize how a team delivers software, its use in scaled and distributed (i.e., not co-located) contexts is still object of research, with some emphasis in planning and inter-team coordination (Moe & Dingsøyr, 2017).

Software development processes in these contexts need to address how software delivered by a team fits in the overall solution, but also how teams must define their boundaries, interfaces, dependences and priorities. Only then, it is possible to apply ASD practices at scaled context, i.e., the concept of "*large-scale agile*" (LSA) (Dingsøyr & Moe, 2014).

Managing projects that include multiple teams is a complex task in large-scale software projects. The process of delivering software using more than one development team, often distributed, faces issues of dependencies, boundaries, coordination and/or synchronization. The challenges of making decisions, setting goals, communicating, building trust and managing the team are far harder (Owen, 2016). With ASD, such task had to be rethought (Dingsøyr, Bjørnson, Moe, Rolland, & Seim, 2018).

In process management, architectures are an artefact capable of supporting a set of coordination decisions. Additionally, architecture is a central artefact when scaling up agile methods, as it is explicitly present in popular "commercial" LSA frameworks, like Scaled Agile Framework (SAFe), Large-Scale Scrum (LeSS), Disciplined Agile Delivery (DAD), Scrum@Scale, Nexus or Enterprise Scrum. Communities such as Industrial XP include "Evolutionary Design" practices, and “Spotify model” have specific architecting roles. “Scientific” LSA proposals like Agile Product Line Architecting (APLA) (Díaz, Pérez, & Garbajosa, 2014), a tailored XP for large-scale projects (Cao, Mohan, Xu, & Ramesh, 2004), or a hybrid RUP+Scrum (Cho, 2009) also include explicit architecture practices.

Although acknowledging the importance of architecture in managing inter-team processes in an LSA context, these approaches lack of a structured approach for using such information to manage the software delivery process. Models are about presenting an abstraction of reality towards a shared understanding of the problem, but a proper analysis allows depicting their input in assigning work, derive dependencies, and manage inter-team communication and coordination.

This chapter describes how a logical architectural artefact is used as basis for managing the process of setting delivery boundaries, communicating the requirements, coordinating and synchronizing multiple teams. The approach presented in further sections is an integrating part of AMPLA, after the candidate version of the logical architecture is derived (cf. Section 5.3).

The research addressed in chapter is the result of using a logical architecture diagram as basis for managing work of multi-teams in ASD and LSA settings, from architecture modularization, requirements communication and inter-team coordination. This study was first applied in ISOFIN and afterwards in UH4SP project. Then, this research addressed defining work instructions for these teams by deriving backlogs. First, using a use case-driven backlog in the iFloW project. Then, deriving user stories statements (in the ISOFIN project) and afterwards deriving other agile product backlog items (in the UH4SP project). In addition, in the UH4SP project, the derived product backlog progress was monitored using a set of agile metrics. The contributions of the projects are summarized in Table 13.

Table 24. Contributions of projects in candidate architectures

Research contribution \ demonstration case	UH4SP	ISOFIN	iFloW
Modularization, coordination and communication from separation of concerns		X	
Modularization, coordination and communication from DDD bounded contexts	X		
Use case-driven backlogs			X
Deriving user story statements		X	
Deriving product backlog items	X		
ASD metrics monitoring	X		

This chapter is structured as follows:

- Section 6.2 discusses architecture modularization, inter-team communication and coordination;
- Section 6.3 describes approaches for backlog definition;
- Section 6.4 describes the demonstration cases and main discussions around use case-driven backlogs, LSA process based in logical architecture, and agile product backlog items derivation;
- Section 6.5 presents the chapter's conclusions;
- The chapter ends with complimentary reading.

6.2. On modularization, communication and coordination

As described in previous chapters, the AMPLA approach is the process for architecture design, based on successive and specific artefacts generation. AMPLA is composed by discovery of user needs, *A-type* sequence diagrams, use case models, a logical architecture, feedbacks and issues, and the consequent software delivery. The artefacts are generated based in the information existing in previously defined artefacts. When software delivery begins, the process is performed in typical cycles, whether in Scrum, Kanban, or other frameworks.

AMPLA has four established phases: (i) Requirements Elicitation (ii) Requirements Analysis & Modelling, (iii) Architecture Design, and (iv) Delivery Cycles. Chapter 4 covered phases (i) and (ii), while Chapter 5 covered phases (iii) and (iv). This section also describes team management-driven work that is performed within phases (iii) and (iv).

The derived logical architecture from AMPLA's V-Model is then the foundational artefact for a distributed agile team framework. The framework, depicted in Figure 116, addresses the architecture modularization, team assignment, dependencies, requirements modelling towards coordination and communication within distributed teams.

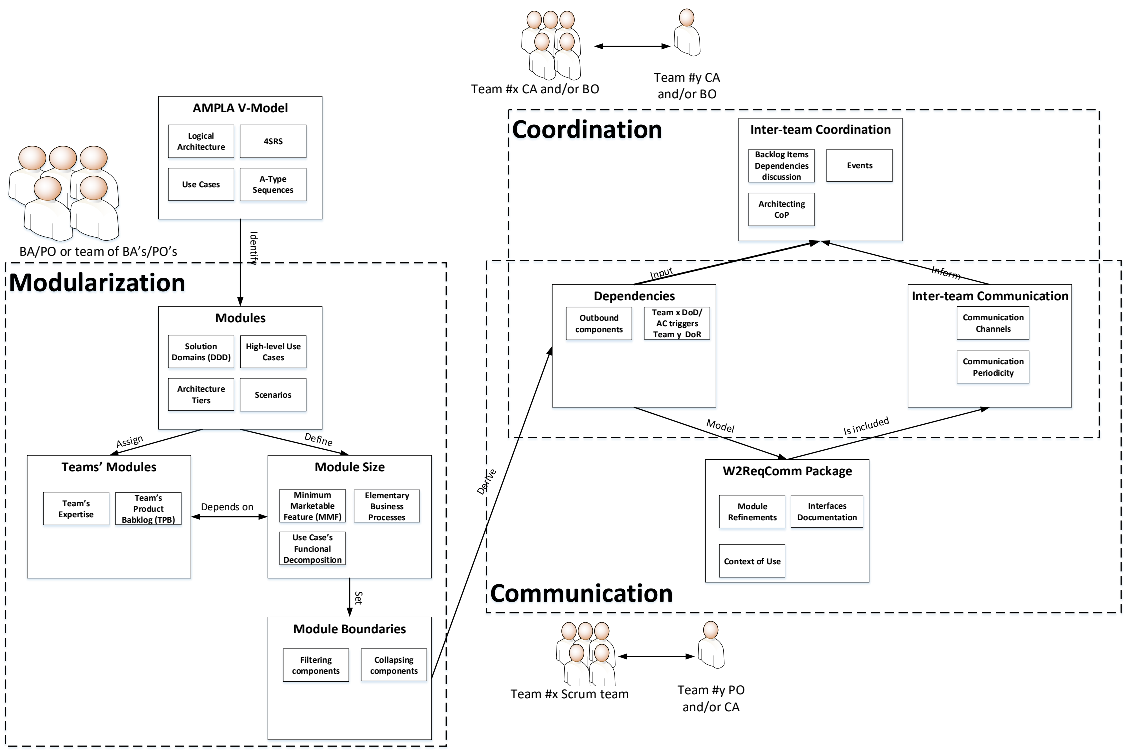


Figure 116. Logical architecture-based distributed agile teams management framework

Modularization

Like in most cases, the best approach to issue complex problems is to divide them into smaller ones and address them one by one, ultimately addressing the big solution. With the purpose of modularizing the architecture, the logical architecture is partitioned into sub-systems. Properly addressing the boundaries across teams is crucial (Rolland, Fitzgerald, Dingsoyr, & Stol, 2016). This is applicable for multiteams systems (MTS) (Mathieu, Marks, & Zaccaro, 2001). It is aligned with the feature teams concept (Larman & Vodde, 2008). A feature team works independently by being given the responsibility for a whole feature. One well-known case of assigning a subset of the architecture is the definition of Tribes in the "Spotify model" (Kniberg & Ivarsson, 2012). The framework in modularization is depicted in Figure 117.

The first concern in modularizing the architecture is in identifying architecture modules by the entities' core competencies. Then, each team is assigned with a part of the solution (e.g., payments, accounts, integration, mobile app, etc.), tiers (front-end, back-end, middleware, data repositories, etc.), “branch” from high-level Use Cases, or scenarios. In this research, use cases are grouped by the (sub-)domains (DDD). This means that each of the tree’s “branches” relate only to a given domain, which also assures that the contexts are properly bounded (Figure 118), allowing teams to work independently. Mapping the use cases to the components enables the identification of the module’s component coverage.

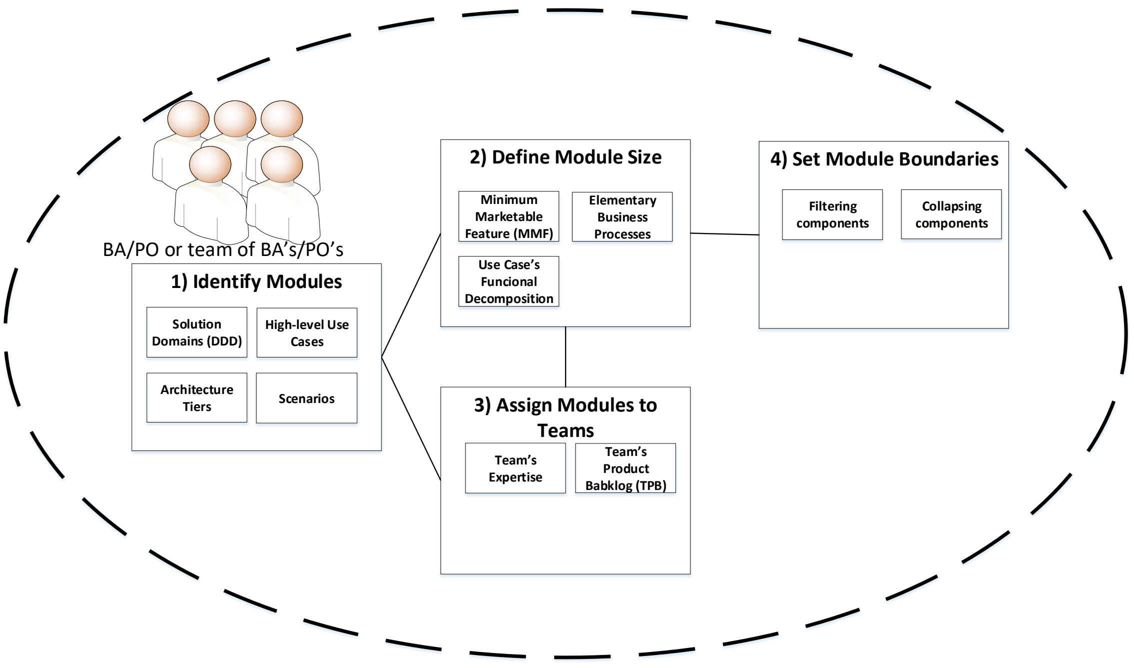


Figure 117. Decision framework within Modularization

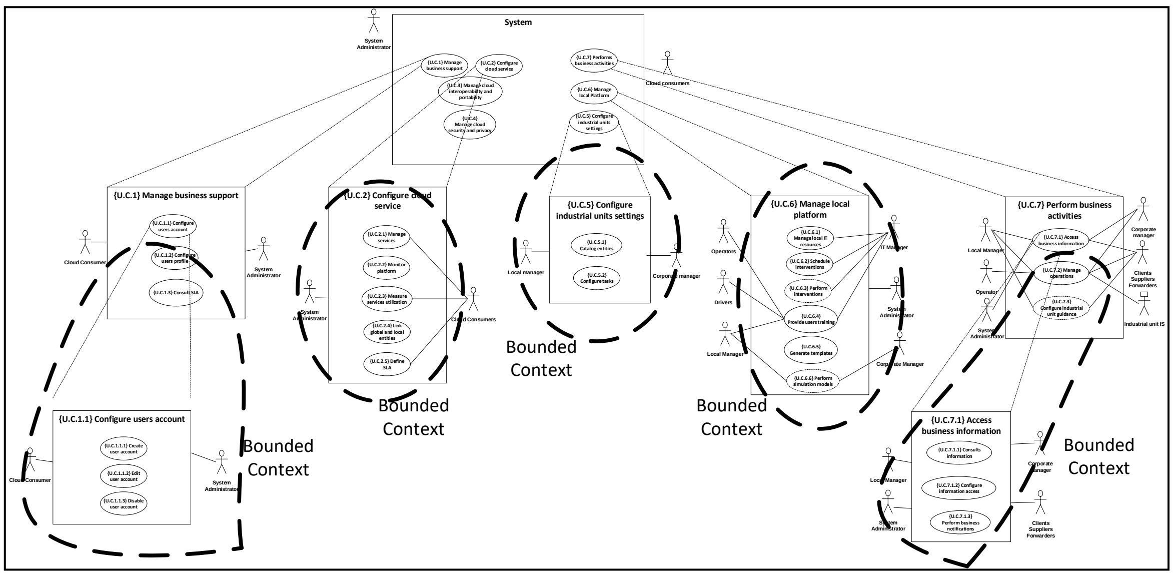


Figure 118. Domain's and sub-domain's bounded contexts (DDD)

In Scrum of Scrums (SoS) literature, the distribution of work between teams shows that each team can self-assign them to any stories from the product backlog. The difference for this approach is that a team product backlog (TPB) is proposed, i.e., a subset of product backlog, at the outset. The predefinition of a subset (or a given feature) of the backlog is also present in approaches like SAFe, DAD, Enterprise Scrum, and Spotify model. In opposition, the team assignment of items from the backlog in LeSS, APLA, Scrum@Scale and Nexus do not follow any grouped stories or features.

Hence, architecture modularization within this research aims at partitioning the logical architecture to define architectural subsets or a group of components will be assigned for a team to implement. It must be assured that a module has composing software components that, together, deliver working software. Only if the set of components are able to deliver working software, it is also possible to perform acceptance testing and, afterwards, integration testing within the code deployment.

However, a module should be bounded not only by representing working software but by also assuring business value delivery. In this research, modularization applies the concepts of a Minimum Marketable Feature (MMF) (Denne & Cleland-Huang, 2003) and an Elementary Business Process (EBP) (Larman, 2004). An MMF is defined as a small, self-contained feature that can be developed quickly and delivered significant value to the user. EBP refers to a single task that adds measurable business value. A module should also be able to support at least one of the scenarios previously modelled in *A-Type* Sequence diagrams during the V-Model execution (including interfaces for components referring to inputs and outputs of the scenario). These decisions in mind output a selection for inbound components. The logical architecture is now a group of architectural modules, like a SoS.

Then, with these decisions in mind, the components to be included in the module are selected. The logical architecture is now a group of architectural modules, as depicted in Figure 119 as a group of “spots”.

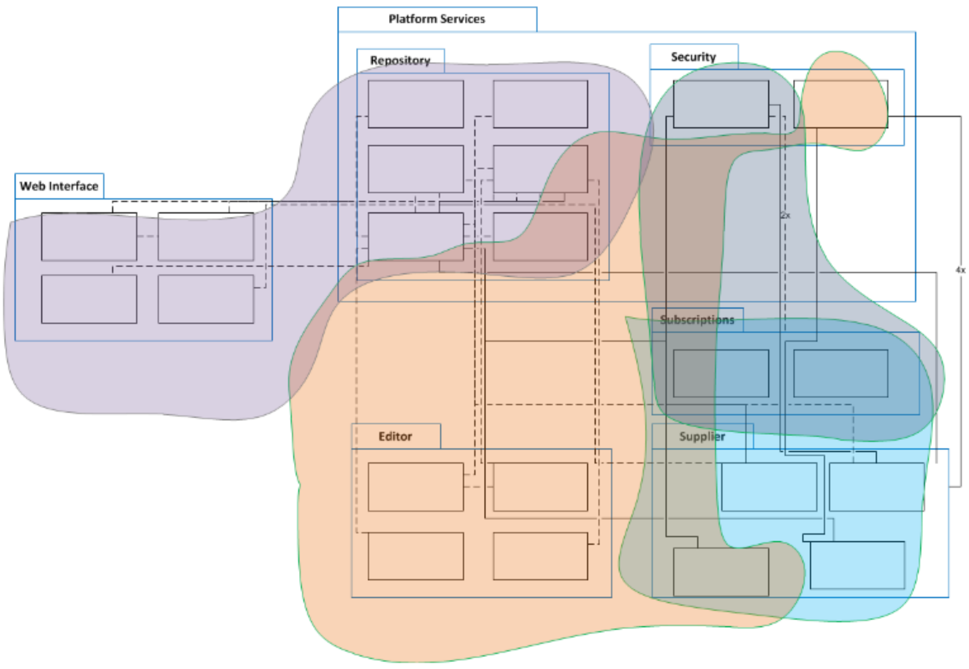


Figure 119. Architecture modularization example

The application of filtering and collapsing techniques (cf. Section 5.3) redefine the system borders. During the filtering process, all components not directly connected to the module must be removed. The inbound components are maintained. The components with direct connections to the module are maintained (as outbound), and the ones without direct connection are removed. The spot represents the sub-system borders, where the software components from the module, as well as the components that directly interact with the model.

Inside the system border defined for the given module, through the respective coverage, the components were maintained as originally characterized. The components with direct connections to the module are maintained, and the ones without direct connection are removed. On the other hand, for representing the interfaces that are outside the system border, we adopted the UML notation for components, to represent inputs and outputs of the functionality. The component-based diagram uses a typical representation of UML component graphic nodes (OMG, 2009). A connector may be notated by a “ball-and-socket” connection between a provided interface and a required interface.

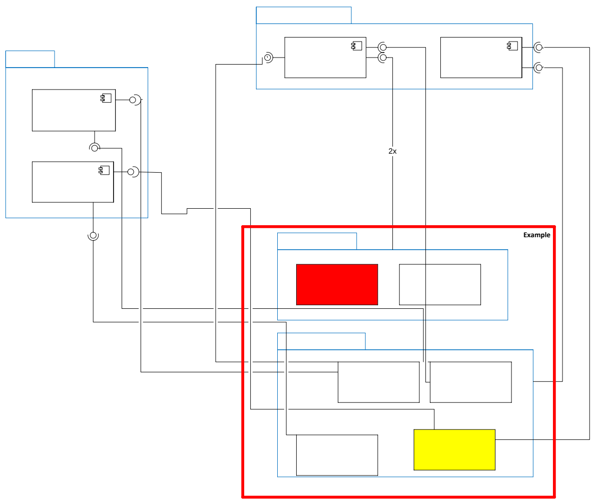


Figure 120. The module representation

The outbound components relate to interface and communication (e.g., APIs) needs. Additionally, they derive dependencies between teams. The dependency is identified within *A-type* Sequence diagrams (see Inter-team management section). A given scenario is only feasible if the constituent components, inbound and outbound are implemented and integrated. Hence, only when the component or story meets its "Definition of Done" (DoD) / "Acceptance Criteria" (AC), the TPB item meets its "Definition of Ready" (DoR) to start the implementation.

Communicating the requirements

While a given team is responsible for delivering working software related to the assigned module, the associations between outbound components require that the team work together with other teams when the integration is needed. Along with managing the dependencies (see next section), a proper communication of "what" is being delivered by the team and required integration is also advisable.

In MTS, global or co-located, the knowledge possessed by each other must be properly communicated with other team in interest. It is not an easy task, since many issues arise due to geographic, temporal, or sociocultural distance.

Knowledge can be shared in Communities of Practice (CoP) - groups of experts who share a common interest or topic and gather to promote discussions (Paasivaara & Lassenius, 2014) – and other gatherings, meetings and informal meetups.

Shared Mental Models or mini demos (Bjørnson, Wijnmaalen, Stettina, & Dingsøyr, 2018; Dingsøyr, Bjørnson, et al., 2018), as well as Video and Audio conferences, online chat, documentation and email (Ahmad, Lenarduzzi, Oivo, & Taibi, 2018) are the most common used tools for communicating knowledge between teams.

AMPLA seeks addressing knowledge sharing using models, where the main purpose is to design the artefacts related to implementing features, to be incorporated in the presented events and tools. The outbound components derived from the filtering and collapsing exercise in Figure 120 allows identifying the dependencies.

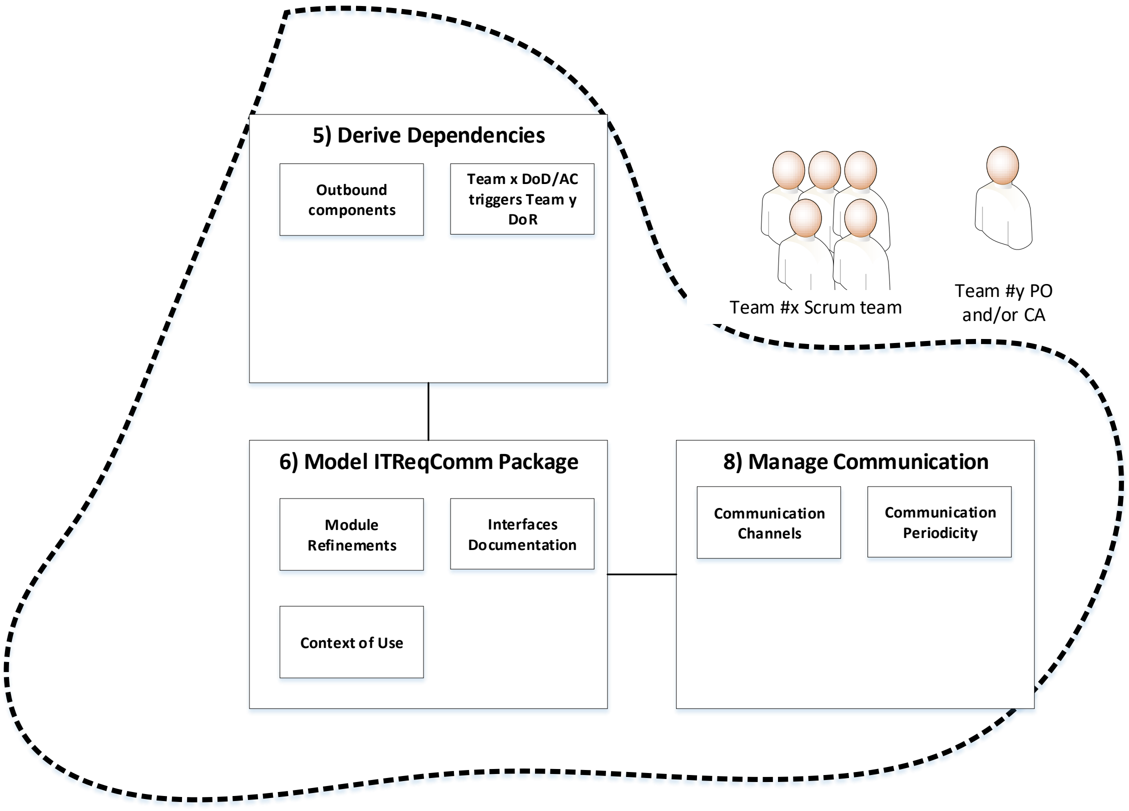


Figure 121. Requirements communication theory

We propose an organization of the information, namely a multi-view perspective of the module to be delivered to implementation teams, the “What? and Why? Requirements Communication” (W2ReqComm) package. The W2ReqComm provides the development teams not only the information regarding the functionalities of the module but also to describe how, and in which scenarios, their future users will use them. This artefact is composed by the software components from the logical architecture that compose the module and their interfaces with external modules.

Additionally, the W2ReqComm includes information regarding the modules usage in the real world. Such information may be described though one or more scenarios, or by including a scenario representation through an *A-type* sequence diagram previously modelled. The scenario should include the functionalities that the module relates to, but also include the functionality belonging to another team module, in order to provide the implementation team with much context information as possible. An example of a W2ReqComm is depicted in **Figure 122**.

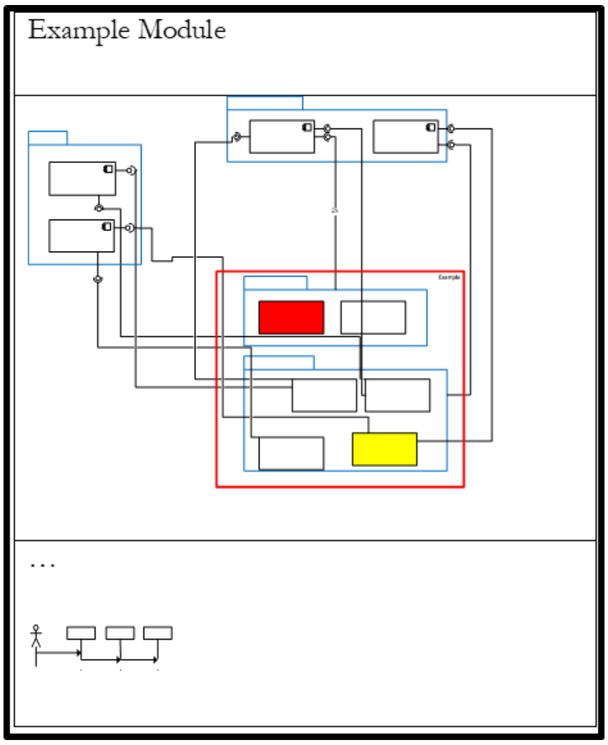


Figure 122. W2ReqComm example

Inter-team management

The process for managing inter-teams development from this research aims identifying coordination needs for addressing dependencies between components, as well as structuring of roles and events. The coordination and management events involve team representatives – architects, PO's, BO's – instead of the entire team.

The identified dependencies trigger the discussions between the representatives, which base in the artefacts from Communication. Discussions take place in planned or unplanned events, using the available communication channels.

It is also worth referring that dependency does not only relate to synchronizing a component's DoD/AC to another component's DoR. They are also used to define team interfaces, where, for instance, if one team has some doubts in implementing a component from a module boundary, practices such as CoP between the predefined representatives promote discussions in overcoming the given doubts. These associations for this phase are depicted in Figure 123.

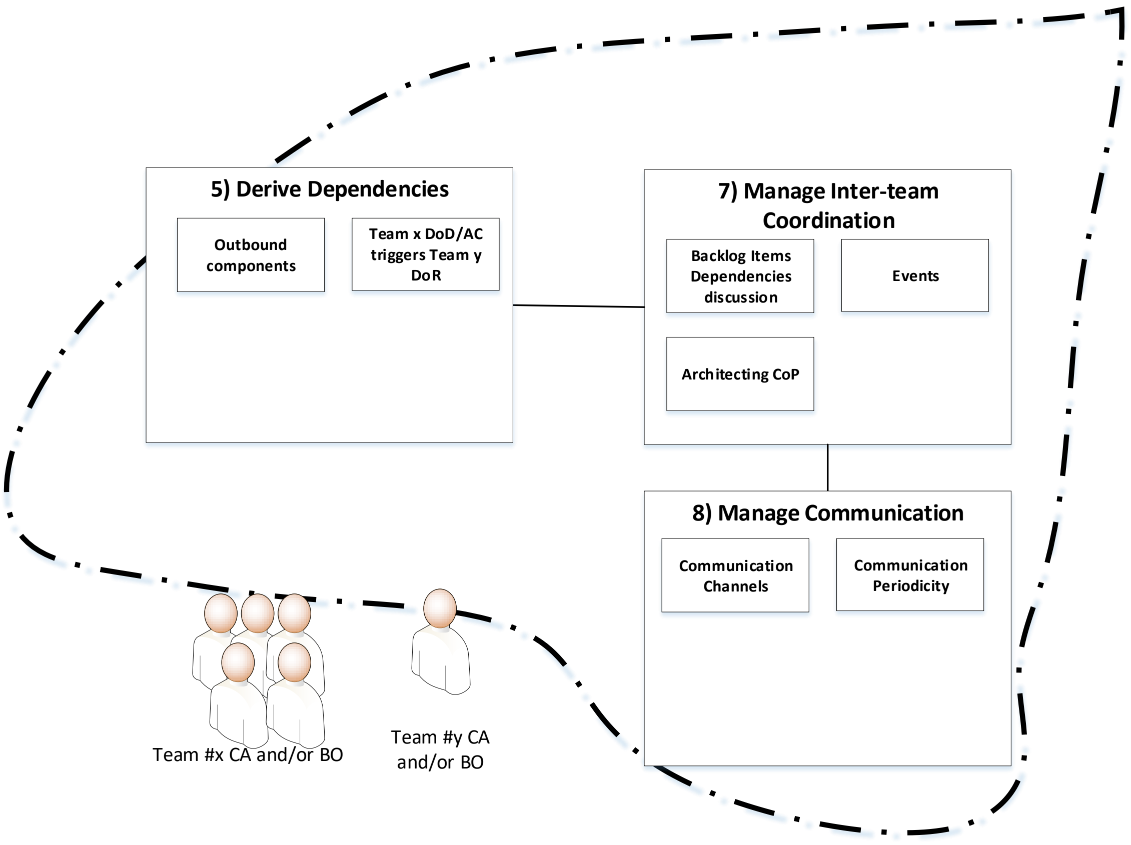


Figure 123. Inter-team management theory

By modelling some processes to validate the flow between components (including components from different modules), e.g., using *A-type* sequence diagrams, dependencies can be depicted, namely some functionalities that must be implemented and executable in order for other functionalities to proper execute. In fact, *A-type* sequence diagrams are powerful tools for bordering the modules, as well as validating (not just the modules but as well the whole) architecture.

Coordination arenas, inspired by Dingsøyr *et al.* (Dingsøyr, Moe, Fægri, & Seim, 2018; Dingsøyr, Rolland, Moe, & Seim, 2017), structure how teams involve with each other, from collaborative tools, communication, chats, but also events (or ceremonies). These arenas are enablers for team cooperation, where the models are the core artefact within the discussion.

6.3. Delivering work items

As the presented framework addressed mechanisms for modularization, communication and coordination of multi-teams, this section rather addresses how a team under analysis manages and controls the work items that they have to deliver.

In ASD approaches, these work items are a composing part of a backlog that the team uses to define the work to be done within the overall project/product and within a given iteration (e.g., a Scrum Sprint).

This section introduces defining backlogs, and its composing items, from requirements models (namely UML Use Cases, Components and Sequence Diagrams) using rules that assure the backlog items cover the gathered requirements.

This research addressed three possible ways to define a backlog from the requirements models: using a backlog composed by use cases directly from those models, deriving user stories statements from use cases and architectural components, and deriving additional backlog items (themes, epics, user stories, details and acceptance criteria). Each one is further described.

Approach for using Use cases as basis for Scrum backlogs

As already proposed in Section 4.2, one approach for delivering work items in a backlog is by composing it directly with the Use Cases modeled during Requirements stage (i.e., Initialization phase of the hybrid approach in section 4.2).

These tasks are represented in a SPEM diagram in Figure 51, depicting tasks that output work products (Use Case Prioritization and Use Case Estimation) and deliverables, namely '*Project Scope*', '*As-Is report*', '*To-Be Report*' and the '*Product Backlog*'. The Business Modeling results are documented in a report designated as '*As-Is report*'. Requirements results are modeled in the form of UML use cases. Design results regard the proposal of the logical architecture (UML component diagram). UML use cases (output of Requirements) and UML component (output of Design) diagrams compose the '*Solution Requirements Specification*' that result in the '*To-Be Report*'. Use case models are used as basis to define a '*Product Backlog*'. This kind of backlog is demonstrated further in Section 6.5.

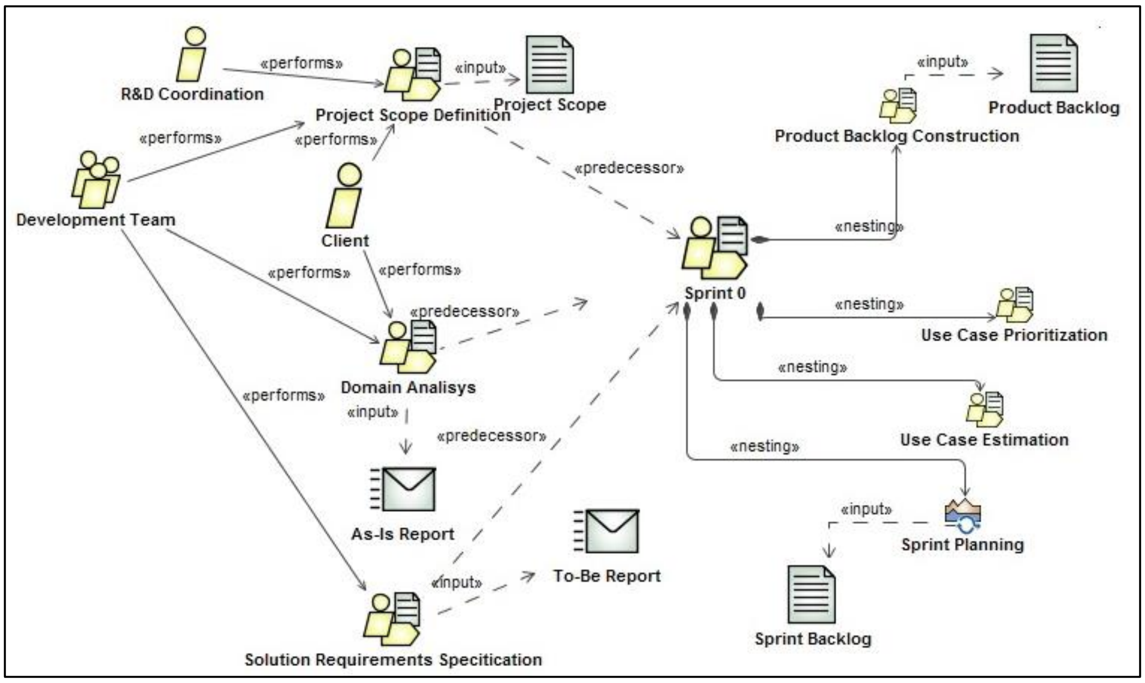


Figure 124. SPEM diagram for Initialization phase

Deriving User stories from components

Using logical architectures for establishing initial requirements allows to combine requirements from backlogs (that focus only on functional features) with the quality attributes of the software (Jeon, Han, Lee, & Lee, 2011). This research proposes including some upfront design in the set-up phase (e.g., Sprint 0, for Scrum projects) of the project - by some we do not mean BDUF, rather "just-enough" (Ambler, 2007) for a candidate architecture - and to use the architecture as input for an ASD approach (back to requirements again) to build almost the totality of the Product Backlog (illustrated in Figure 125). The 4SRS method allows deriving logical architectures aligned with the corresponding, and previously elicited and modeled, user requirements. A logical architecture is a view that primarily supports the functional requirements, taken mainly from the problem domain (Kruchten, 1995). The conventional version of the 4SRS method is typically applied in large-scale projects, but demands high quantity of information (use cases, textual descriptions), which is often time consuming and, in every way, misaligned with the general paradigm adopted by ASD.

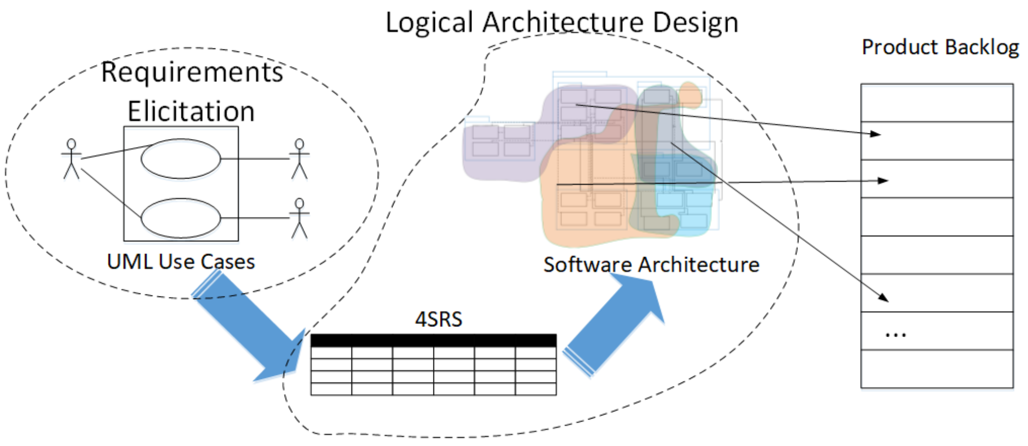


Figure 125. Approach for delivering backlog items requirements

The starting point for the User Stories derivation is the logical architecture diagram that results from the 4SRS method execution. In some cases regarding very large products, it is easy to see that these models can be extremely large and heavy to be analyzed as a whole, because these diagrams represent all the modules needed to run all desired functional requirements. Moreover, it is unlikely that, for large systems, only one Scrum team will perform all the work. In a model where there may be hundreds of modules, a Scrum team could take an amount of time not feasible with the needs of a dynamic market.

Thus, deriving User Stories from the modules presented in the previous section allows that several Scrum teams can work in parallel, reducing the time required to implement and deliver the solution to the customer.

The first critical decision related to the development of our approach was to understand what should be the relation between the components and the User Stories. In the 4SRS method, the components are derived through the decomposition of Use Cases in three different types (interface, data and control).

In a first hypothesis, we decided to create one User Story for each components, as depicted in Figure 126. This decision intends to maintain the core principles for writing User Stories (*i.e.*, the INVEST characteristics – Independent, Negotiable, Valuable, Estimable, Small and Testable). Additionally, it complies with the greater flexibility for the Product Owner to follow the team's work. If the User Stories are always complex and require great effort to implement, there is the risk of diluting one of the main advantages recognized of agile methodologies: the ease of changing the direction of the team and the ability to see, in real time, which is state of commitment of the team to a Sprint. When implementing User Stories of great complexity, which may occupy the entire Sprint, only one can draw conclusions about the speed and commitment of the team at the end of the Sprint, when work (supposedly) must be completed. These arguments all meet the characteristic of having small User Stories (Small) and, thus, simplifying the implementation effort estimation (estimable).

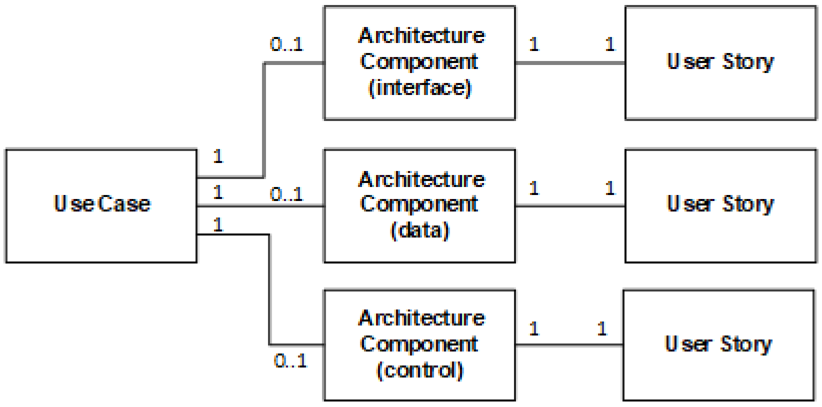

Architecture Component (interface)
Architecture Component (data)
Architecture Component (control)
Use Case
User Story
User Story
User Story
0..1
0..1
0..1
1
1
1
1
1
1
1
1
1

Figure 126.Relation between Use Cases, Components and User Stories

The information from the 4SRS method execution, mainly micro-steps 2i - "Use Case Specification", 2iii - "Component naming" and 2viii - "Component specification", together with the actors associated with each use case (where each component was derived) are key elements in the generation of User Stories, since in them are encapsulated information required to write User Stories respecting the INVEST principles. The proposed technique for deriving user stories is composed by three steps, as follows:

Step 1 – Group Components.

The first step is to group components and analyze their functionality and their interfaces. This step has as input the components from the logical architecture or, in case of an architecture modularization as the one presented in the previous section, from a given module.

Step 2 – Analyze component specification and use case description

In this step, we gather the information from micro-steps 2i - "Use Case Specification", 2iii - "Component naming" and 2viii - "Component specification" and the involved actor (by reversing to the use cases that derived the component). This step uses the traceability characteristic that the 4SRS method provides, by allowing to easily depicting the original use case.

All these details of each component should be stored with the same structure to give input to create a "card" for each User Story, containing all the information needed to carry out its estimation and subsequent implementation. Thus, for each component is important to obtain the following information: Name; Code; Type; Description; Package; Associations ; Direct Associations; UC Associations; Original Use Case; the Actors Involved; and the UC from the functional decomposition.

Alongside this information, it is also important to depict if the component is part of more than one module. Regarding the teams that have habits to keep information always visible from User Stories (placing its features in physical format, often in the form of cards), it was created a User Story template that includes all information collected and previously listed, as well as some information that the implementation team will generate in grooming, as the number of Story Points, acceptance criteria, or any other comment that the team find relevant register and save. Table 25 depicts a template including the information needed for any stakeholder (from the customer / Product Owner to implementation teams). The completion of the card is also intended to be basic and quick as all information regarding the component and the Use Case is available from the execution of the 4SRS method, while information on the User Story is mandatory and is defined by the implementation team during grooming.

Table 25. User story card template

User Story	US #	(name)		
	Acceptance criteria	• #1 • #2 • #3		***Story Points***
Architecture	**Component #**	(num)	(name)	
	Type	(interface/ data/ control)	***Package***	
	Specification			
	Multiple	(Yes/No)	**Module**	
	Association	**Direct Associations**		
		UC Associations		
Use Case	**UC #**	(num)	(name)	
	Description			
	UC Ass.	(num / name)		
	Actors			
	(Comments)			

Step 3 – Write the User Story

This is the final step and where the effecting output of the process is generated. The 4SRS method execution, followed by an analysis on the derived diagrams and documentation, allows triggering the procedure for mapping a logical architecture to output a set of User Stories that comply with the INVEST principles.

One of the great advantages of applying the 4SRS method to derive the component is that it quickly allows realizing the ultimate goal of the component: data manipulation, communication or logic operations, by reading the type of component. This standardization of component types simplifies the management of User Stories because, after all, they are centered on three very specific types of tasks.

The "who", the actors involved and who will perform tasks on the User Story to implement, are easily identified by analyzing their User Story Card and looking for those involved in the Use

Case that derived the component (by executing the 4SRS method). As the logic surrounding the need to represent properties of systems in Use Case and User Story is similar (to capture specific requirements in terms of interaction between users and system), it is easy to validate that those involved in the Use Cases will be benefited by the implementation actors Story of a particular User.

Regarding the "what," you can also find a direct relationship between this component and the name of component. Firstly, it is necessary to find an action, represented by a verb, to identify what you want to implement. The division between components of control, and data interface simplifies this demand, since the component interface always refer to the creation of a specific interface and is therefore an action which is fixed and constant need for the existence of a communication interface for between component and / or actors. In most cases, the name of component only indicates what kind of interface is required. Thus, in cases of interface (*i-type*) components, the actors involved just need their existence in order to use them in their workflows. By using the name of the corresponding components, and using connections want/need to have (want/need to have), the connection between the "who" (actor) and what (action) is derived. In cases of *i-type* components that do not have this syntax, the central part of the User Story for the "what" is simply left to the information “want/need to have an interface”, and the title of the component (the actions that will take place using that interface) used as part of the "why". For data (*d-type*) components the process is similar, since they usually refer to the need of the existence of repositories/storage locations or interfaces for communication with such storage spaces. Thus, the construction of the User Story follows the same rule used in *i-type* components.

Compared with the previous two types, control (*c-type*) components are disparate. They support the logic behind a system, representing all actions that can be performed by manipulating the data (represented by *d-type* components) and using interfaces for transmission (represented by *i-type* components). As they can represent any action on the system, typically *c-type* components have associated a verb that represents the action that it performs. In this case, we are deriving information of a title for a sentence. Some kind of semantic correctness of words may be required, allowing the sentence to make sense.

This approach allowed User Stories to fulfill their main purpose, which is to identify work to be done.

Deriving User stories and Product Backlog Items from Use Cases and Components

In this section is proposed a systematic transformation of model-based requirements (UML Use Cases and Component diagrams) into ASD-oriented requirements, according to the backlog items. Leftingwell also describes software requirements approaches for agile teams (Leffingwell, 2010). Additionally, he includes a metamodel for a common understanding on requirements information in agile product backlogs (Figure 127). The Agile Extension to the BABOK® Guide lists a variety of requirements artifacts and activities present in known agile frameworks (IIBA, 2017) such as Scrum, XP, Behavior-driven Development (BDD), Kanban, and Agile Unified Process (AUP).

It was based in these works that we defined the backlog structure that should result after performing a set of derivation rules. These derivation rules aimed a backlog that followed the path of Themes, then Epics, then User stories, then finally tasks. Each User story has associated Acceptance criteria (which gives the "Definition of Done" (DoD), and may have details that describe the requirements to implement the story in software. The backlog items are the following:

Deriving Themes: A theme in a Backlog item relates to a generic concept. A theme in a Backlog item relates to a generic concept, realized by a set of Epics (Leffingwell, 2010). For that reason, a theme may be derived by the identified packages. In 4SRS, a package is identified for logically grouping a set of components from the architecture.

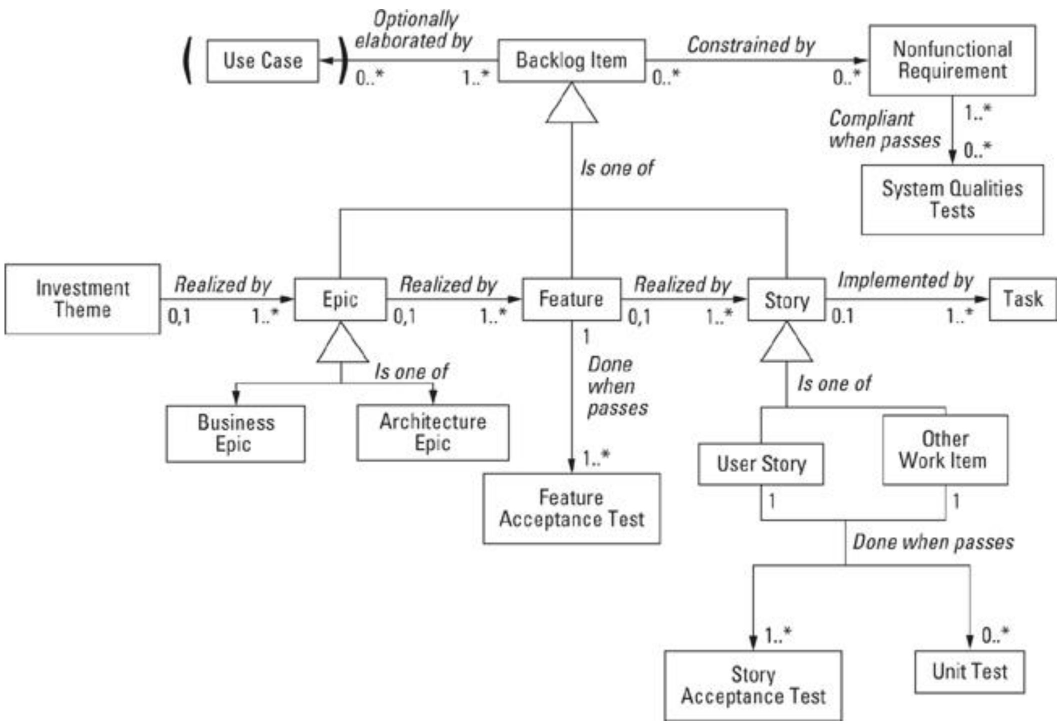


Figure 127. Agile requirements metamodel (Leffingwell, 2010)

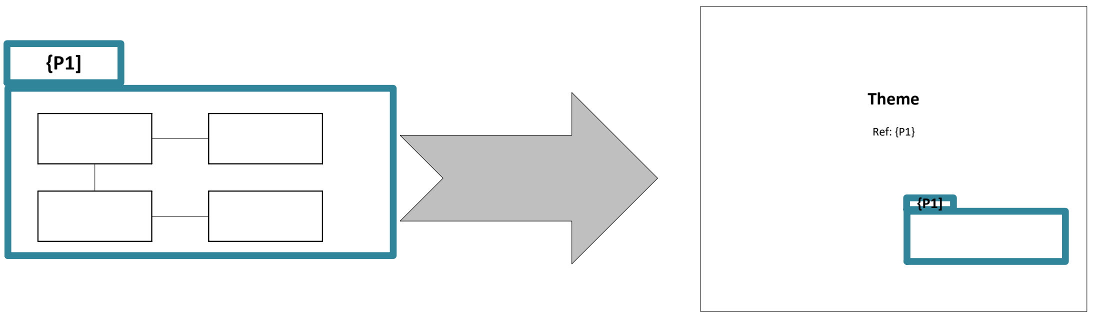


Figure 128. Rule for deriving Themes

Deriving Epics: An epic describes a requirement that further needs to be divided (in a User Story). A User Story is considered to be smaller than a Use Case. Additionally, it is discussed that a Use Case contains a set of interrelated User Stories (Cohn, 2004). For that reason, in our approach, this rule suggests that an epic is directly derived from a Use Case. Although this relationship is arguable, it is assured that this way the Epic item is stated in a Product Backlog, referring to a required development work. In addition, it needs further refinement before inclusion in a Sprint Backlog (which is basically what already happens, since an Epic is not able to be included in a Sprint Backlog unless it is refined in User Stories).

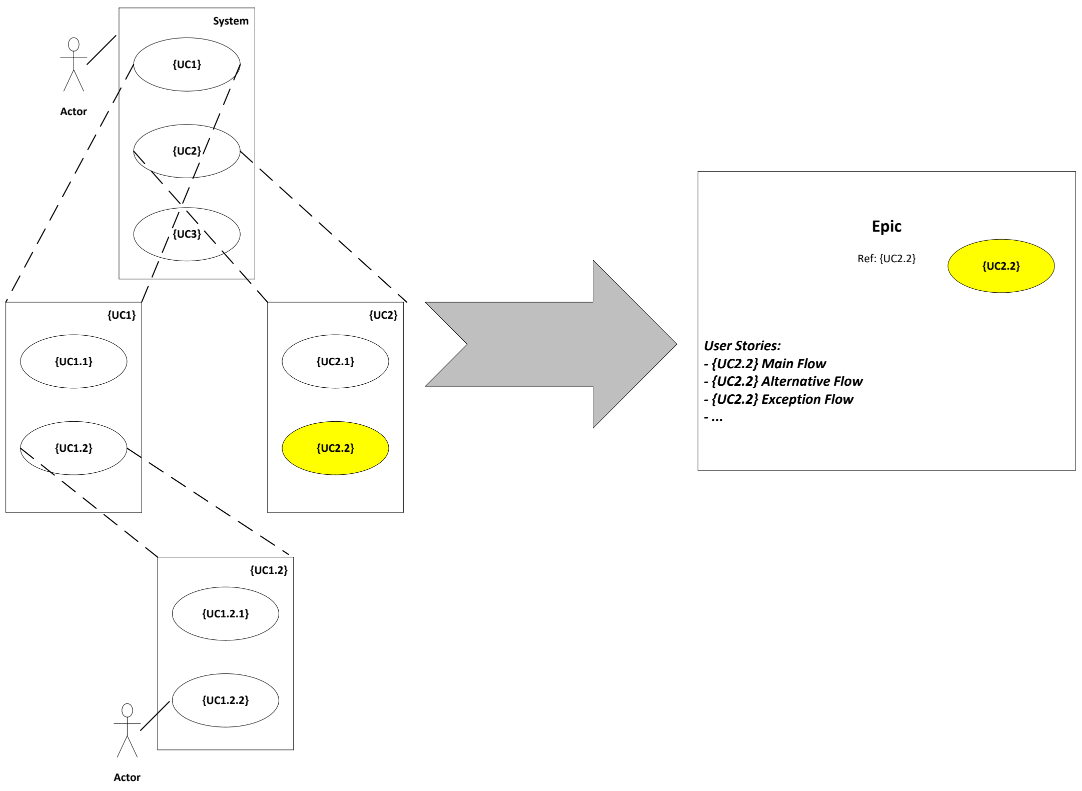


Figure 129. Rule for deriving Epics

Deriving User Stories: this rule proposes that the story creation should be based by each of the scenarios from a use case, which is in line with Cohn (Cohn, 2004) and Jacobson (Jacobson, Spence, & Bittner, 2011), for instance. The scenarios are identified in the use case description, namely as main flow and alternate flow (Cockburn, 2001). These flows in use case descriptions provide the business value of a given scenario from a use case, thus it related the requirements to the business value of the story.

In what composing the user story statement is concerned, it follows the same set of rules as described in the previous section but, instead of including the component name, it includes a part of the (use case) flow that indicates the flow purpose.

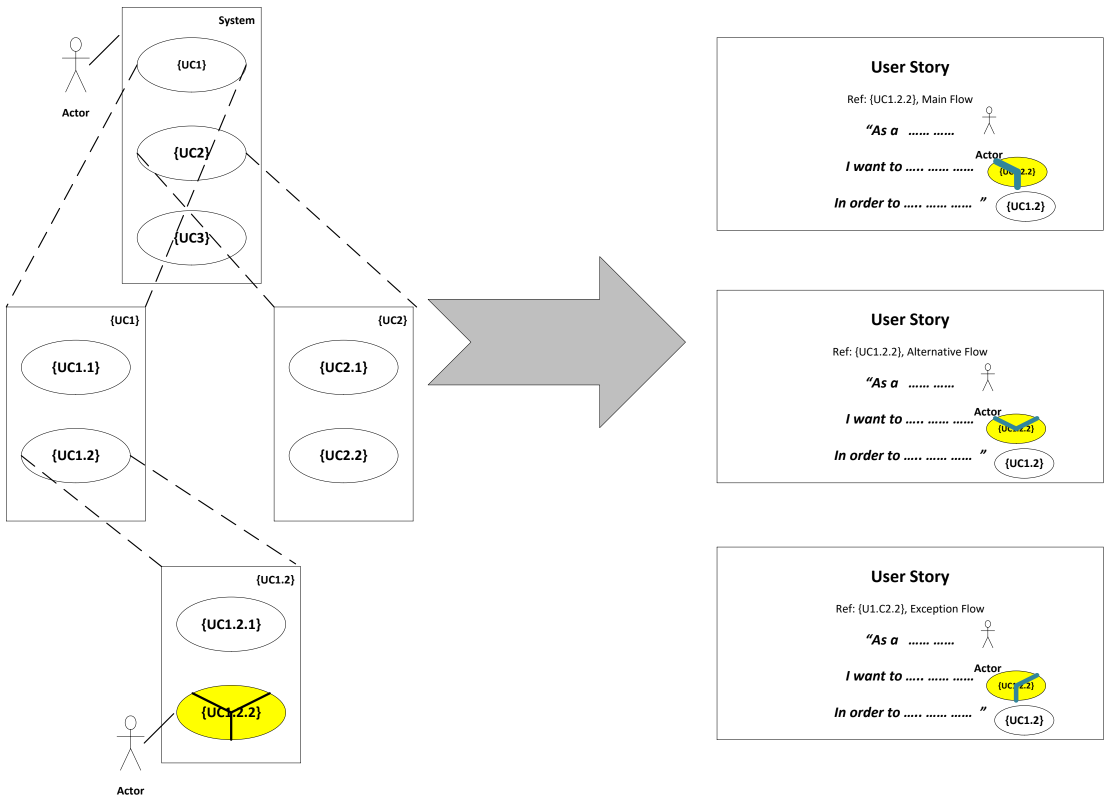


Figure 130. Rule for deriving User Stories

Deriving User Stories details: The inclusion of user stories in the backlog *per se* refers to "promises for a conversation" and not as actual requirements specifications. These specifications are commonly included separately within their '*details*'. The functional (and some of the non-functional) behavior of the story is depicted by the software components responsible for executing the given functionality. These software components are identified by the 4SRS method execution, by tracing back to the use cases, which relate to user stories, as proposed in the previous rule. Another important aspect of the detail may be the context of use, which is depicted by scenarios, modeled in sequence diagrams as suggested during the modularization. This information can be gathered and described in plain text in an informal way.

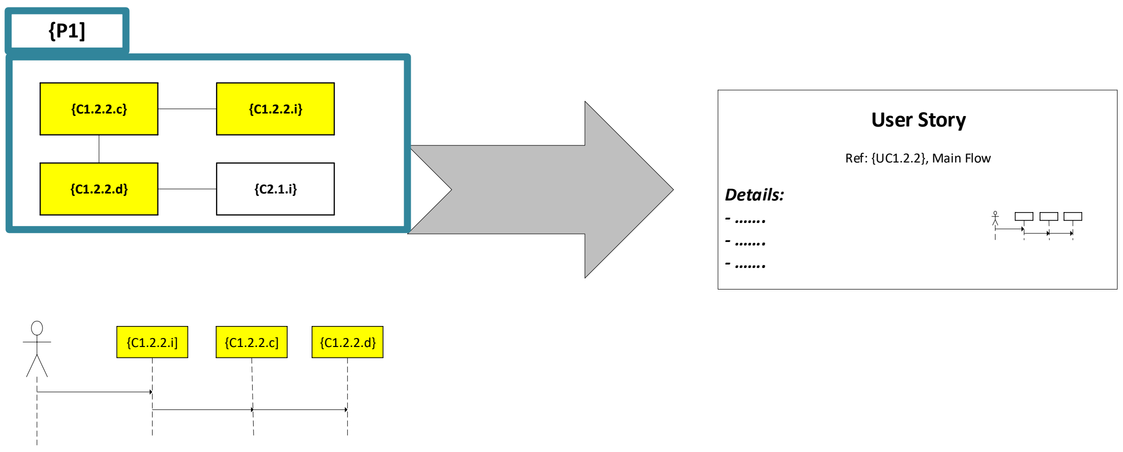
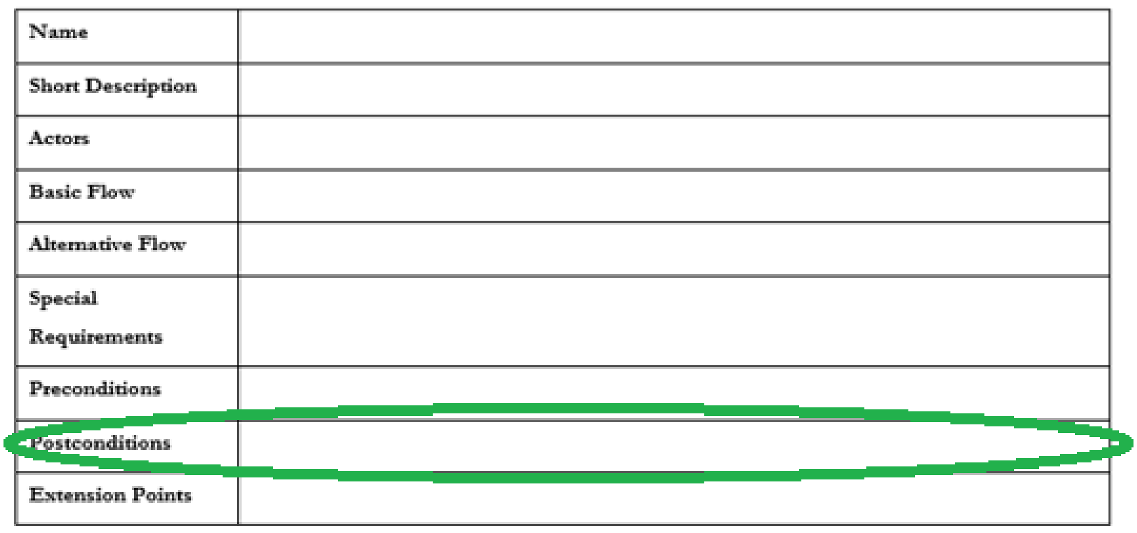


Figure 131. Rule for deriving user story details

Deriving Acceptance Criteria: The expected behavior after performing the use case scenario is described in the use case description, namely the post condition, in the template structure as suggested by Cockburn (Cockburn, 2001).



Name	
Short Description	
Actors	
Basic Flow	
Alternative Flow	
Special Requirements	
Preconditions	
Postconditions	
Extension Points	

Figure 132. Rule for deriving Acceptance Criteria

The Definition of Ready checklist

It is assumed that, since the components have been refined, they are in an improved situation to be now delivered for implementation. The combined view of user stories (along with their related use cases), software components and sequence diagrams (if applicable) allow initiating the implementation, by its ‘definition of ready’ (DoR) (Power, 2014). DoR is a set of agreements that define if an item is sufficiently prepared so that a team can start to work on it.

The combined view of the requirements models, in addition to the backlog derivation, must be validated on its ‘readyness’ for initiating the development iterations (e.g., Sprints), which is to say they meet DoR. Namely, the identification of the stories, acceptance criteria, dependencies and the integrated use of the stories (**Table 26**), mainly because allows understanding the “who”, “what” and the “why”.

Table 26. Checklist DoR for a User Story

DoR criteria	Criteria Fulfillment
User Story defined	The fields for writing the statement (“As a…, I want to…In order to…”) are filled based on UML Use Case and Component information.
User Story Acceptance Criteria defined	The Component information includes acceptance criteria.
User Story dependencies identified	The dependencies were identified within UML Use Case and Component diagrams.
User Story sized by Delivery Team	(independent from transformation rules output)
User Experience artifacts are Done and reviewed by the Team	The combined view includes UX/UI artifacts for each user story, or at least each Epic.
Architecture criteria (performance, security, etc.) identified, where appropriate	The logical architecture and its components are discussed during Grooming
Person who will accept the User Story is identified	(independent from transformation rules output)
Team has reviewed the User Story	(independent from transformation rules output)
Team knows what it will mean to demo the User Story	The ‘overall picture’ of the User Story was modeled in UML Sequence diagrams regarding the defined project core processes.

6.4. Demonstration cases

The aforementioned LSA team management approaches (modularization, coordination, communication and backlog item derivation – use cases, user stories from components, and backlog items from use cases and components) are instantiated in this section within a demonstration case. The team management approach based in a backlog with use cases is demonstrated by the iFloW project. The team management approach based in a backlog built from the derivation of user stories statements is demonstrated by the ISOFIN project. Finally, the team management approach based in a backlog from the derivation of themes, epics, user stories and acceptance criteria is demonstrated by the UH4SP project. Additionally, this section includes demonstration of progress control, using a set of agile-oriented metrics, by a team from the UH4SP project that used the derived backlog items in their software development process.

Team management approach based in a use case-driven backlog: the iFloW case

Team settings

In the iFloW project, Bosch mainly performed as a software customer and UMinho as a contracted software development entity. The team was co-located, but some integration support was provided by a third-party remote team (cf. Section 6.5).

In this project, the core iFloW team was composed of nine collaborators with multidisciplinary backgrounds:

- Bosch:
  - one Product Owner, that was representing other eight elements from the Logistics department, which formally dictated the requirements.
  - one member of the IT department, responsible for validating that each developed product increment could be easily integrated within Bosch information system;
- UMinho:
  - three R&D coordinators, with the role of assuring that the scientific rigor (from both the system and the software development process) and deadlines of the project are met;
  - four software developers with methodological and technological competences (like analysis, requirements, design, database modeling, programming, testing, deployment, etc.).

The entire software development was performed within Bosch's premises, where the iFloW team elements (in exception of R&D Coordinators) were located on a daily basis. The elements from UMinho had no previous knowledge of the domain (in this case, logistics), so the team decided that the project kicked-off by gathering and documenting requirements in a waterfall-based approach.

After the requirements engineering was performed, and since iFloW aimed developing a software system for an industrial context, the team decided to follow the Scrum framework as the iterative approach for the implementation phase. This phase was performed by development iterative cycles in form of Scrum Sprints. Based in incremental software deliveries, both UMinho and Bosch could manage their project's expectations.

As a collaborative University-Industry R&D software project, the previously presented roles are slightly different from the roles defined by the Scrum framework (namely, Product Owner, Scrum Master and Development Team) (Schwaber & Beedle, 2001), however easily mapped, as depicted in Table 27.

Table 27. Mapping between iFloW roles and typical Scrum roles

Scrum Role / iFloW Role	Product Owner	Scrum Master	Development Team
Bosch			
Product Owner	✓		
Bosch IT	✓		✓
UMinho			
R&D Coordinators		✓	
Software Developers			✓

Use Cases that compose the backlog

During the initialization phase of the hybrid method, the iFloW requirements gathering output were modeled in a set of UML Use Cases, depicted in Figure 133. Each of the use cases were functionally decomposed, which resulted in 90 lower level use cases.

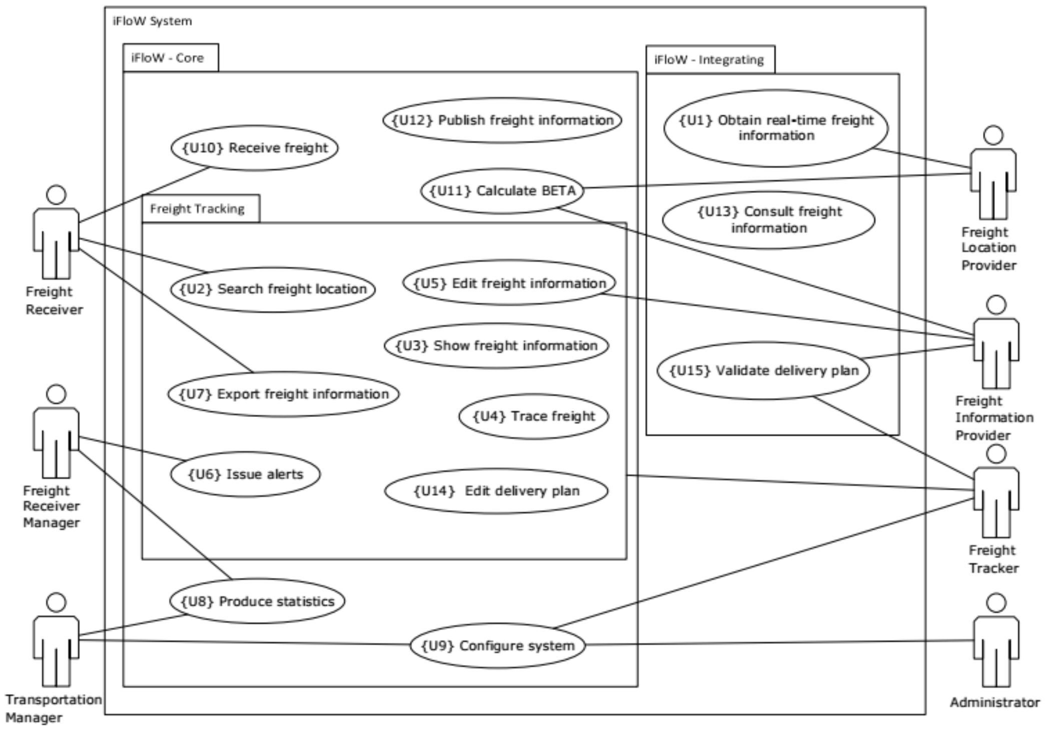
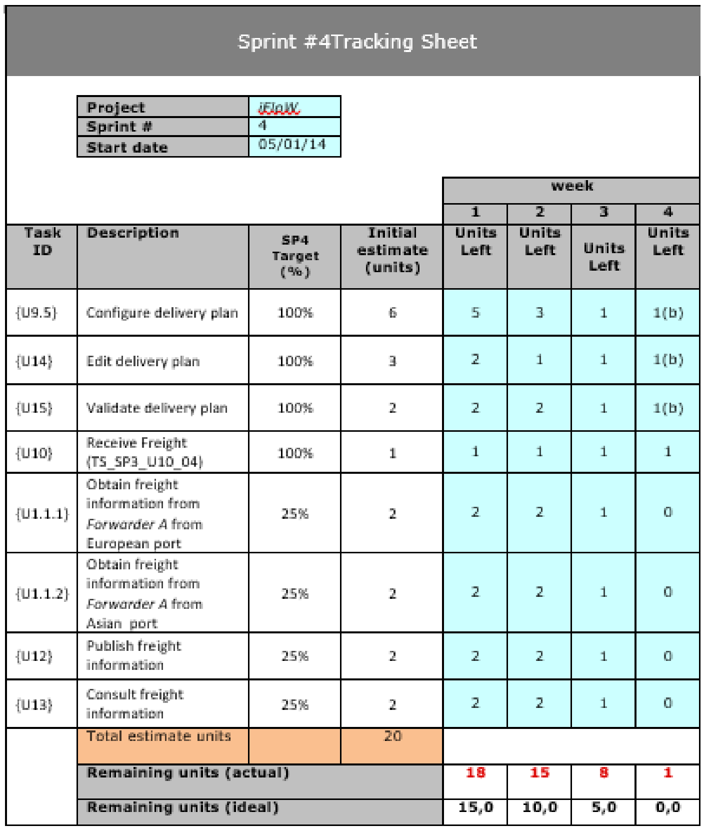

iFloW System
iFloW - Core
iFloW - Integrating
Freight Tracking
{U12} Publish freight information
{U10} Receive freight
{U11} Calculate BETA
{U1} Obtain real-time freight information
{U13} Consult freight information
{U2} Search freight location
{U5} Edit freight information
{U3} Show freight information
{U15} Validate delivery plan
{U7} Export freight information
{U4} Trace freight
{U6} Issue alerts
{U14} Edit delivery plan
{U8} Produce statistics
{U9} Configure system
Freight Receiver
Freight Receiver Manager
Transportation Manager
Freight Location Provider
Freight Information Provider
Freight Tracker
Administrator

Figure 133. Use Case diagram of the iFloW project

Afterwards, within the implementation phase, the use cases from the '*Product Backlog*' were implemented iteratively and incrementally during eight four-week (Scrum) Sprints. In this phase, typical Scrum iterations were performed, where each '*Sprint Backlog'* is a selected subset from the '*Product Backlog*'. In Figure 134 is depicted a '*Sprint Backlog'* tracking sheet, composed by the iFloW use cases and whose progress was monitored.



Sprint #4Tracking Sheet

Project	iFloW
Sprint #	4
Start date	05/01/14

				week			
				1	2	3	4
Task ID	Description	SP4 Target (%)	Initial estimate (units)	Units Left	Units Left	Units Left	Units Left
{U9.5}	Configure delivery plan	100%	6	5	3	1	1(b)
{U14}	Edit delivery plan	100%	3	2	1	1	1(b)
{U15}	Validate delivery plan	100%	2	2	2	1	1(b)
{U10}	Receive Freight (TS_SP3_U10_04)	100%	1	1	1	1	1
{U1.1.1}	Obtain freight information from *Forwarder A* from European port	25%	2	2	2	1	0
{U1.1.2}	Obtain freight information from *Forwarder A* from Asian port	25%	2	2	2	1	0
{U12}	Publish freight information	25%	2	2	2	1	0
{U13}	Consult freight information	25%	2	2	2	1	0
	Total estimate units		20				
	Remaining units (actual)			18	15	8	1
	Remaining units (ideal)			15,0	10,0	5,0	0,0

Figure 134. A Sprint Backlog based in Use Cases from the iFloW project

Team management approach based in a User stories-driven backlog derivation: the ISOFIN case

Team settings

The ISOFIN project was composed by eight entities (private companies, public research centers and universities). During the requirements stage, the consortium defined that the entire ISOFIN solution would include a cloud platform and a set of local services. For the case of the cloud platform, one entity formed a team specifically to model requirements in order to deliver them to other three entities that would have Scrum teams to implement them. For the case of the local services, another entity modeled requirements that the remaining two would implement. This research focused in the team responsible for gathering requirements for the cloud platform.

Modularization

Previously in Section 4.4, the modeling process performed within the ISOFIN project has been already described. By following a V-Model approach, models were derived in succession (Figure 53). This derivation path includes the solution's business processes, A-type sequence diagrams, use cases, logical architecture and B-type sequence diagrams. In its vertex, the 4SRS method assures that user requirements models (in the left side) are aligned with system requirements models (in the right side).

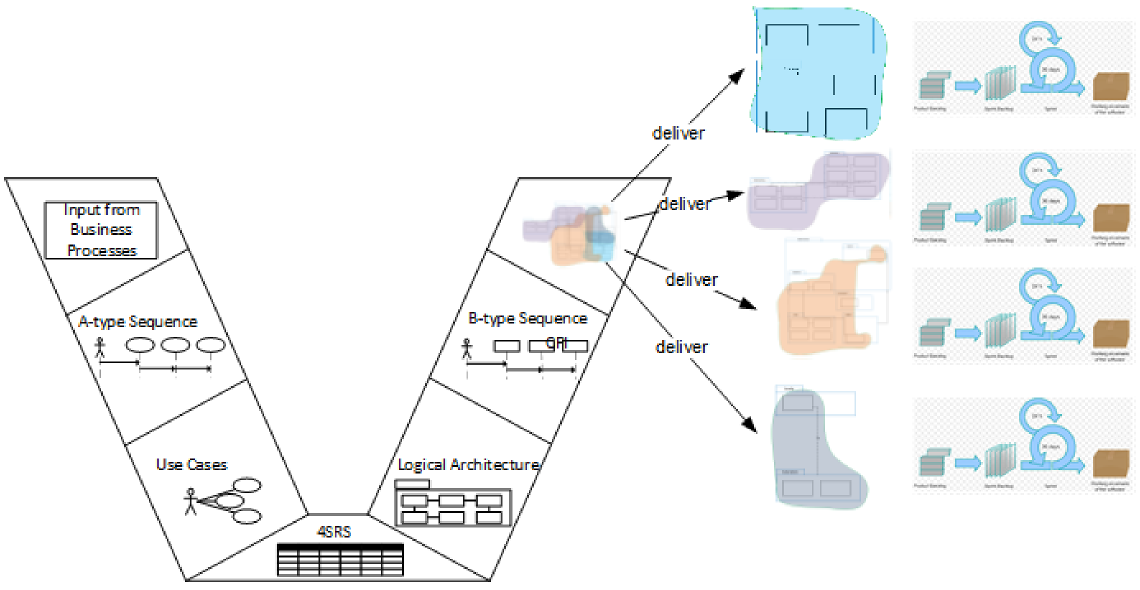


Figure 135. The result of the V-Model to be delivered to multiple Scrum teams

The ISOFIN logical architecture is composed by 105 components. The overall architecture referred to software development functionalities that were to be delivered by three software teams, each belonging to a different entity.

Having as basis the logical architecture and, additionally, *B-type* sequence diagrams and the solution's business processes, the analysts discussed, depicted and proposed a set of application/modules to be developed. The proposed applications/modules are composed by components from the logical architecture, and the composition of each application is depicted in Figure 136 through a set of "spots" that traverse the logical architecture.

The components that are "covered" by the *spot* represent the expected functionalities for a given application. However, it is not possible yet to depict sequences and flows for the application execution (such is provided by *B-type* sequence diagrams) as well as the components that interface with the application.

The logical diagram was partitioned in seven "spots" covering the components (Figure 136), representing applications to be developed: Integrated Business Services (IBS) Management; ISOFIN Applications Management; Alerts Management; Subscription Management; Security

Management; Policies Management; and Logs Management. An overview of the “spots” is presented in Annex B.

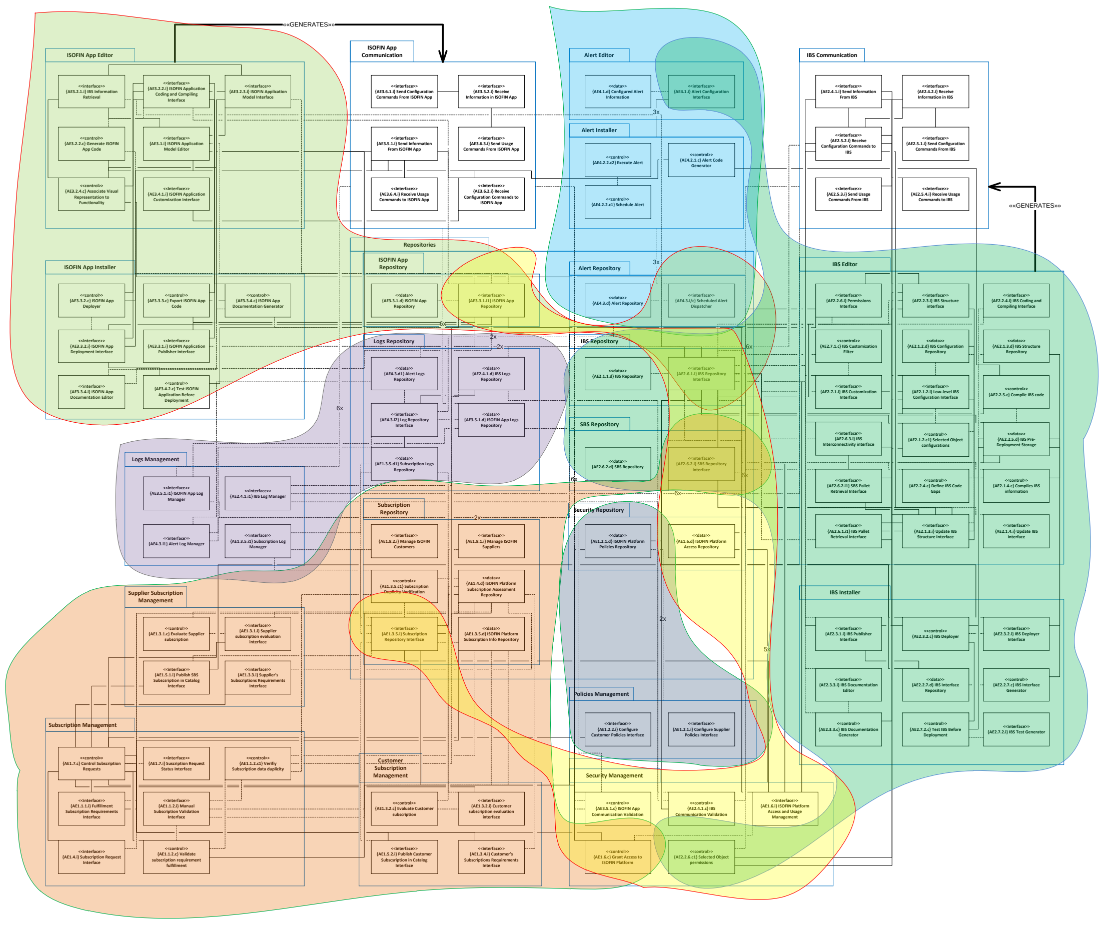


Figure 136. ISOFIN architecture modularization

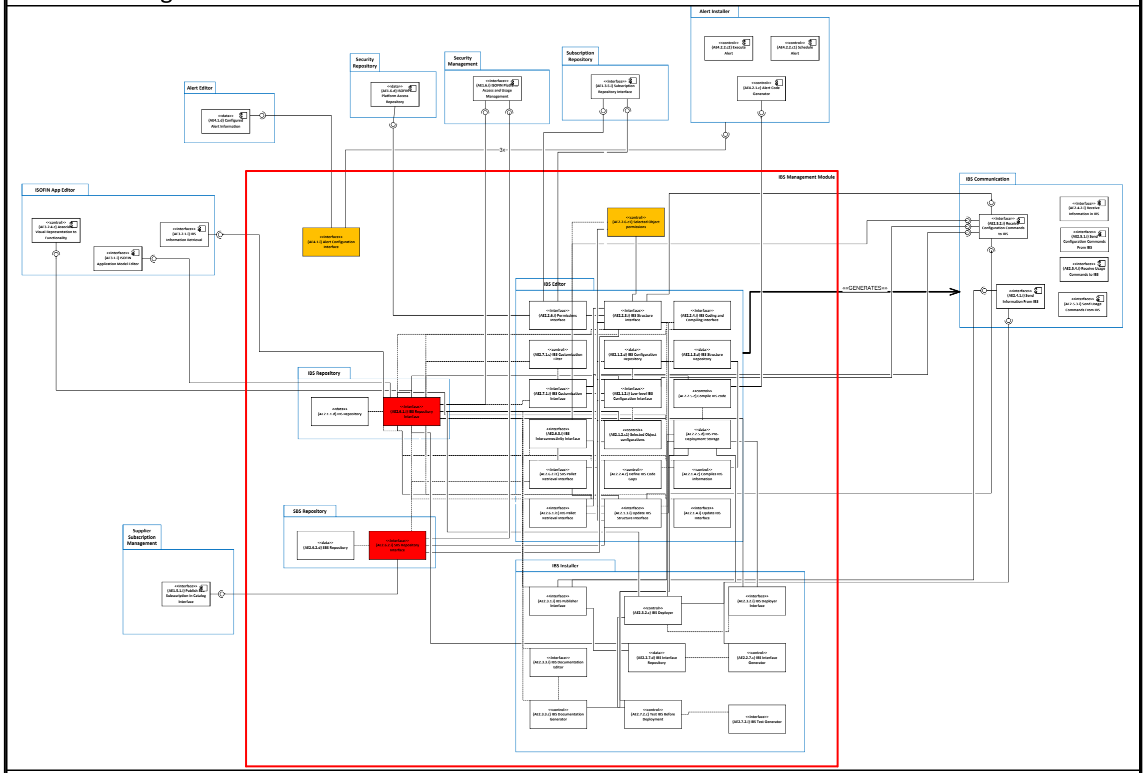
We present in this section one of the modules to illustrate the demonstration case: the IBS Management module. By executing this module, An IBS Developer develops a new IBS by modeling the IBS, selecting the available IBSs from the pallets. Besides modeling its structure, the IBS Developer is also responsible for defining permissions, manually filling gaps in the IBS code, publishing the information in the catalog and deploying the IBS in the ISOFIN Platform.

Communication

The filtering and collapsing technique that was applied within the logical architecture allowed depicting the components that compose the module and their interfaces, and then depict its W2ReqComm. The W2ReqComm for the IBS Management module is presented in Figure 137.

Regarding the overlapped components in more than one module, one of the teams is nominated to be responsible for implementation of the module and assure that the teams responsible for modules with dependencies with that particular one have all required documentation and provide updates on its implementation.

IBS Management Module



An IBS Developer develops a new IBS by modeling the IBS, selecting the available IBSs from the pallets. Besides modeling its structure, the IBS Developer is also responsible for defining permissions, manually filling gaps in the IBS code, publishing the information in the catalog and deploying the IBS in the ISOFIN Platform. If necessary, the IBS Developer performs test and fixes coding errors. The IBS to be developed may require setting system alerts.

Figure 137. W2ReqComm for IBS Management module

User Story derivation

We present in Table 28 an example of the use of user story card, using the *Test Before IBS Deployment* component example. Information regarding the name of the User Story presupposes the execution of the next step of this method. The acceptance criteria and the story points fields are not defined at the time of the User Story derivation. They are defined later during Sprints, so these fields were not yet defined in the card in Table 28, thus defined as “not applicable” (*N/A*) at this time.

Table 28. User Story card for "Test IBS Before Deployment"

User Story	US #1	As a Business User or a IBS Developer, I want to test IBS before deployment, in order to render IBS in pre-runtime.			
	Acceptance criteria	N/A			***Story Points***
					N/A
Architecture	**Component #**	2.7.2.c	Test IBS Before Deployment		
	Type	Control	***Package***	IBS Installer	
	Specification	This component allows testing of the IBS before deployment. This component will be required before the execution of *{C2.3.2.c} IBS Deployer* to verify that no problems occur during the execution of the IBS. All information need for the execution is provided by *{C2.2.5.d} IBS Pre-Deployment Storage*			
	Multiple	No	**Module**	IBS Management	
	Association	**Direct Association**	2.7.2.i – IBS Test Generator		
		UC Association	2.3.2.c – IBS Deployer 2.2..5.d – Pre-Deployment Storage		
Use Case	**UC #**	2.7.2	Render IBS Pre-Runtime		
	Description	Configure and defines the pre-runtime of the IBS. This use case allows testing of the IBS before deployment.			
	UC Ass.	2.7 – Configure IBS			
	Actors	Business User IBS Developer			
Others					

The User Story must provide the "why" of a particular actor ("who") may need to perform a certain action ("what"). This information, often induced by the very title of the corresponding components, can be complemented with a description of the Use Case from which the component was derived. As User Stories relate to a lower level than uses cases, the description of the use case itself can justify the need for existence of a particular User Story. In cases where the name of components is quite similar to use case from which the component was derived, the significance of User Story can be found in the description of the use case. Using these rules, remaining User Stories were derived that are listed in Table 16. In Figure 138 is represented a User Story sentence based in the derivation from Table 16.

Table 29. User Stories derived from *c-type* components

#	Component	As a(n) <actor>	I want/need <description>	In order to <outcome>
2.1.2.c1	Selected Object Configurations	ISOFIN Customer / IBS Developer	select object configurations	change (IBS Structure) configurations
2.1.4.c	Compiles IBS information	IBS Developer	compile IBS (changes and) information	create a new IBS
2.2.4.c	Define IBS Code Gaps	IBS Developer	(automatically generated code) and define IBS code gaps	create IBS code
2.2.5.c	Compile IBS code	IBS Developer	compile IBS code (and create new IBS catalog)	(keep IBS catalog and store) compile(d) IBS Code
2.2.6.c1	Selected Object Permissions	IBS Developer	select object permissions	set(/manage) permissions (and create IBS)
2.2.7.c	IBS Interface Generator	IBS Developer	(automatically) Generate IBS Interface	(store the) generate(d) IBS interface
2.3.2.c	IBS Deployer	IBS Developer	deploy IBS	execute IBS deployment
2.7.1.c	IBS Customization Filter	Business User	filter IBS (configuration and) customization	customize IBS
2.7.2.c	Test IBS Before deployment	Business User / IBS Developer	test IBS before deployment	render IBS Pre-Runtime

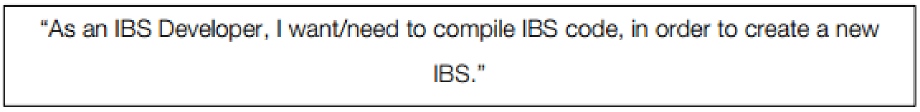

"As an IBS Developer, I want/need to compile IBS code, in order to create a new IBS."

Figure 138. User Story from 2.1.4.c

Inter-team management

Now that the User Story derivation is complete, there are just some issues that are dealt in the multiple teams' management. In the case of the ISOFIN project, the teams were distributed but belong to the same organization and were not geographically distributed. The quantity of teams were not as many as the modules identified, but the total quantity of teams is not relevant, since they belong to the same organization. Thus, the organization chose to nominate a single Scrum Master to work closely with all teams. The Product Owner was responsible for the decisions during the implementation, like detecting potential delays and decisions on critical issues across the teams.

Regarding the overlapped components in more than one module, it is then the Product Owner's responsibility to nominate a team to be responsible for implementation of the derived User Story and assure that the teams responsible for User Stories with dependencies with that particular one have all required documentation and provide updates on its implementation.

Multi-team management and coordination: the UH4SP case

Team settings

The UH4SP project was composed by five teams from four different entities for software development where each had specific expected contributes, from cloud architectures to industrial software services and mobile applications. The entities are geographically distributed, but each entity had a single located team. Figure 139 depicts the roles structuring between the involved teams.

The business need relates to managing, communicating and coordinating software delivery. Team #A is expertized in mobile and image recognition technologies, composed by three Developers, a Quality Assurance (QA) engineer and a Scrum Master (SM) - which acts with Org #1. The Chief Architect (CA) – that also takes the role of Product Owner (PO) – is responsible for the architectural decisions of team #A's Team Product Backlog (TPB) and part of the architecture team of the project. Teams #C and #E has the same role structure, but expertized in web and API applications. Team #B has a Scrum team expertized in web and microservices development, composed by three Developers, a QA engineer, the SM, a CA and a Business Analyst (BA) – also acting as a PO – responsible for managing team #B's TPB. Team #D is responsible for the cloud infrastructure, composed with a CA and DevOps engineers. Additionally, all teams also include strategic roles such as Project Manager (PrjMmg), and a Business Owner (BO) that is responsible for team's products portfolio.

Teams work internally within their delivery of increments, based in Scrum teams or not. Alongside with the delivery, there are important decisions that require integration among teams, relating to architecture, dependencies, and coordination. The management and communication between these teams arise the need to define a process. Strategic decisions also require communication between PrjMng and BO, however not addressed in this research.

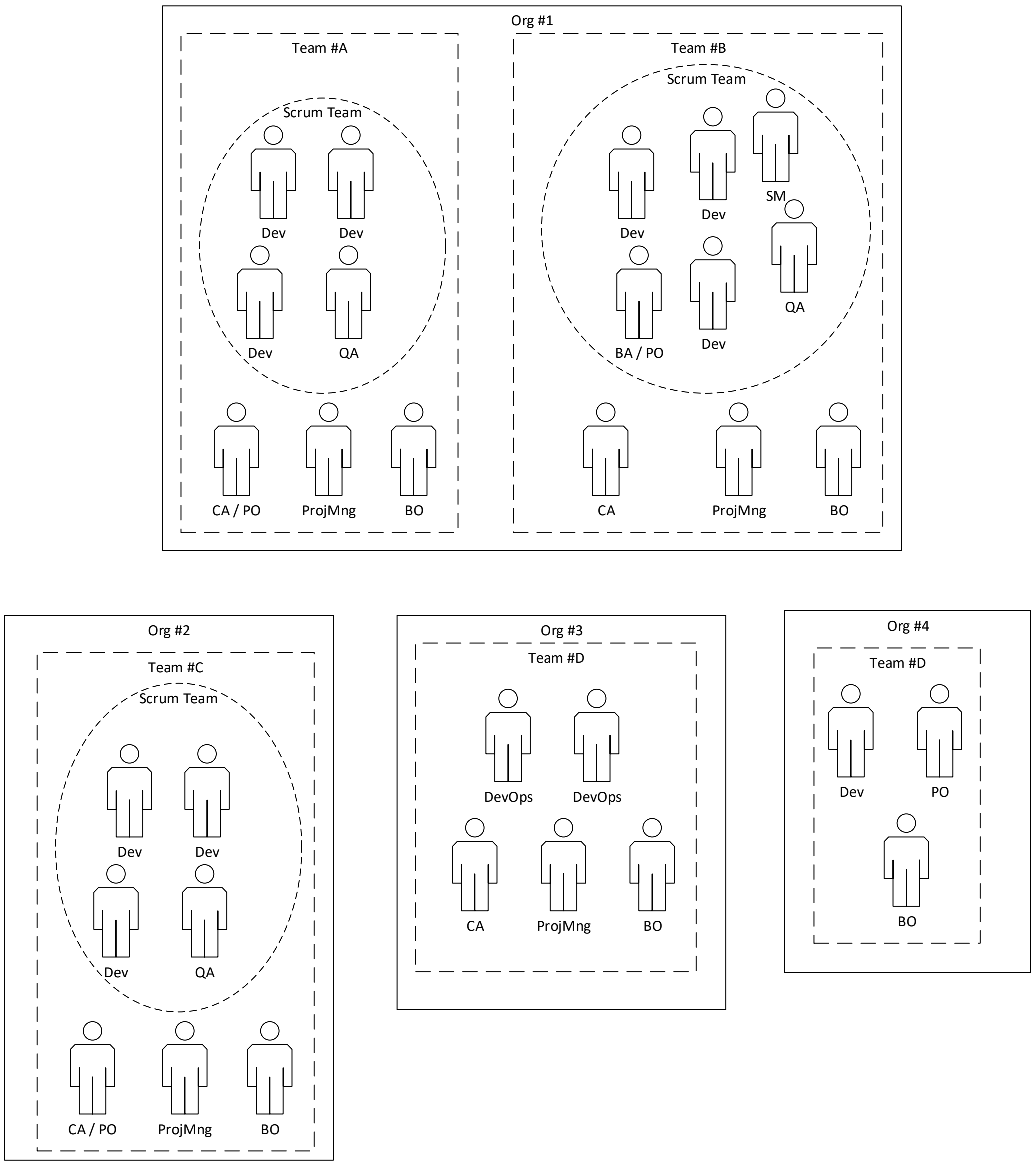


Figure 139. Structure of UH4SP teams

Architecture modularization

Section 5.5 described how a candidate version of the UH4SP project logical architecture was derived in an agile-oriented way using AMPLA. The UH4SP logical architecture had as input 37 use cases and, after executing 4SRS method (Annex C), was derived with 77 architectural components (Annex C) that compose it. This architecture was afterwards divided in a set of modules to be assigned to each of the project's teams (Figure 140).

The modularization depicted in Figure 140 originated five modules/subsystems, each assigned for 'Team #A', 'Team #B', 'Team #C', 'Team #D' and 'Team #E'. The bordering was based in the contributions that each team brings to the consortium, namely IoT, cloud infrastructure, cloud applications and sensors/embedded systems. Each domain was reflected in the use case model and, by consulting the 4SRS method, the use case coverage was mapped to the components, in order to the module be built. Figure 140 depicts the modules' borders and dependencies as a group of "spots".

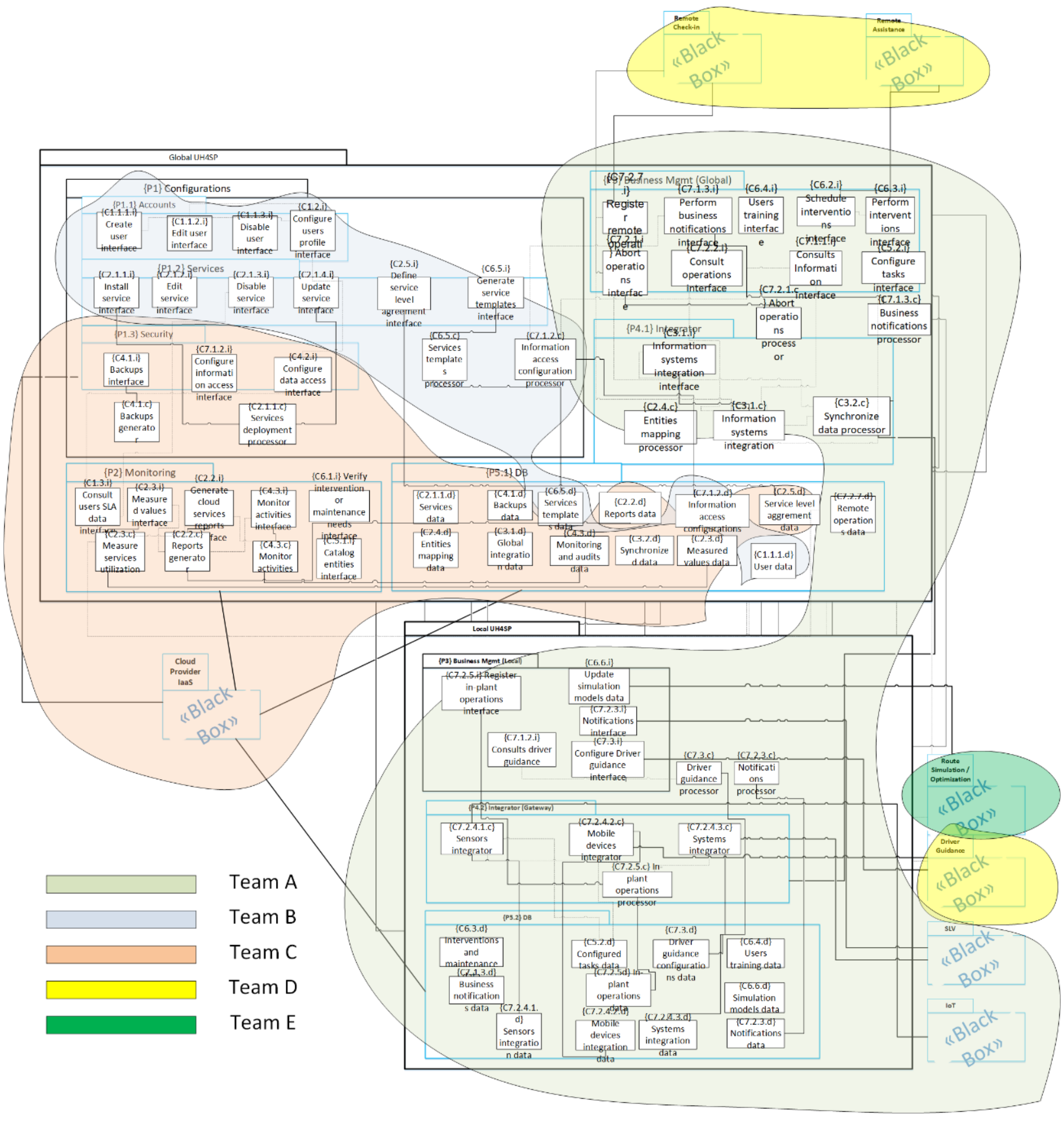


Figure 140. UH4SP logical architecture modularization

Backlog derivation

By the time of the refinement process, the total of Use cases after refinement was 96. Relating to the total of logical architecture components after the refinement, only two modules were measured, because the remaining teams decided to go for implementation without performing the 4SRS method. These measures are presented in Table 30. When analyzing Team B module, the subset more than doubled after the refinement, as presented in Table 31.

Table 30. Analysis on the product backlog

Candidate Arch	Before refinement	After refinement
Use cases	37	96
Components	77	94*

*only measured for two modules

Table 31. Analysis on the Team B backlog

Team B module	Before refinement	After refinement
Use cases	11	29
Components	15	39

The user stories were then derived and specified. A subset of the stories that compose Team #B's Team Backlog is presented in Table 32. The entire Product Backlog from Team #B may be consulted in Annex C. The stories details are defined by input of the components from the 4SRS method execution, depicted in **Table 32**.

Table 32. A subset of the team backlog

Epic: **Account Management**		
	Use Case: {UC1.1} Configure users account	
User Story	As a SysAdmin, I want to create a user account in order to configure user accounts.	Acceptance Criteria: SysAdmin is able to create user
User Story	As a SysAdmin, I want to change a user account in order to configure user accounts.	Acceptance Criteria: changed information is stored.

The application of the transformation rules resulted in a Team Backlog composed by 61 user stories, which were implemented in six Sprints. Among them, only two stories were considered incomplete, i.e., required additional knowledge acquisition from the developers besides the components and sequence diagrams. All stories were foreseen and clear after the transformation rules. Regarding dependencies, no user stories were identified out of order. It must be also pointed out that, among the 61 user stories, 2 of

Table 33. Traceability between use cases / user stories and the components from the 4SRS

Epic: Account Management		
	Use Case: {UC1.1} Configure users account	
User Story	{US1.1.I.} As a System Administrator, I want to create a user account in order to configure user accounts.	Acceptance Criteria: {AC1.1.1.I.} System Administrator is able to create user
User Story	{US1.1.II.} As a System Administrator, I want to change a user account in order to configure user accounts.	Acceptance Criteria: {AC1.1.1.II.} System Administrator changed user information is stored.
	Use Case: {UC.1.2} Configure users profile	
User Story	{US1.2.I.} As a System Administrator, I want to create a user account in order to configure user accounts.	Acceptance Criteria: {AC1.1.1.I.} System Administrator is able to create user
	…	…

Use Case	**Component**
{U1.1.1} Create user account	{C1.1.1.d} User data
	{C1.1.1.i} Create user interface
{U1.1.2} Edit user account	{C1.1.1.d} User data
	{C1.1.2.i} Edit user interface
…	…
…	…

Table 34. Analysis on Team B Sprints

Team B Sprints			4 Sprints
Nr. Success stories	59	Nr. New stories	2
Nr. Unforeseen stories	0	Nr. Incomplete stories	2
Nr. Unclear stories	0	Nr. Out of order stories	0
Nr. Interface stories	2		

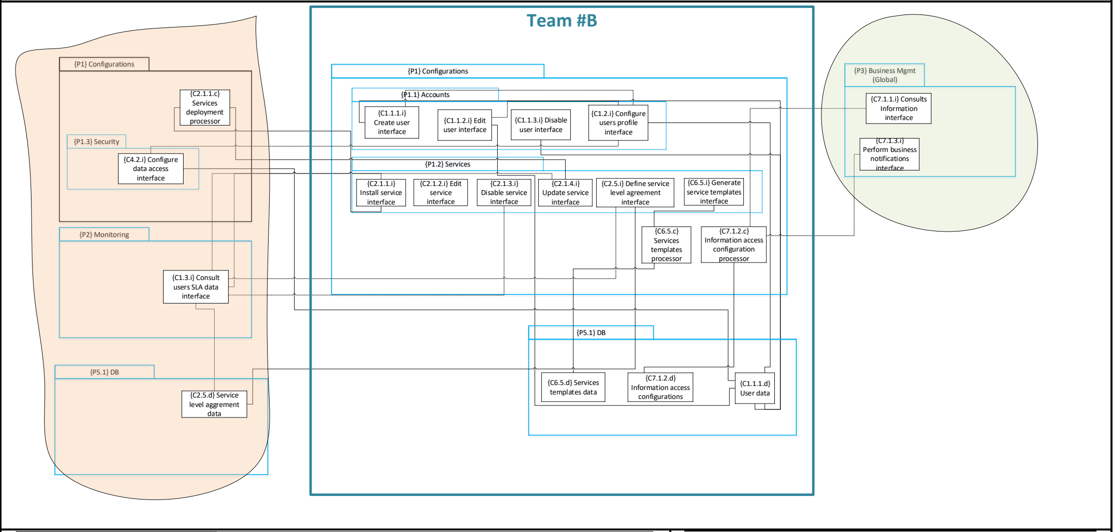
them relate to stories with interactions with other team's stories, and were immediately identified within the modularization.

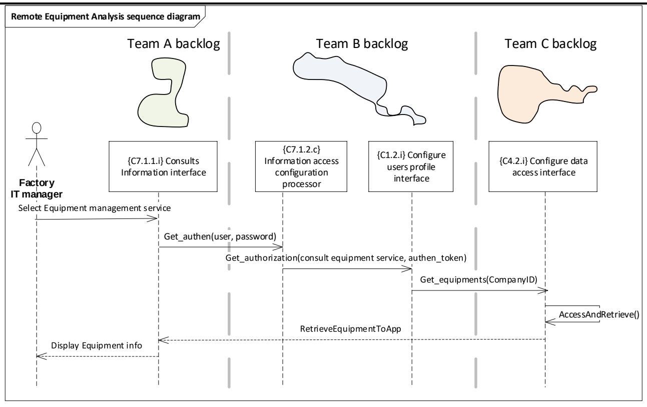
Requirements communication

Team #B’s module includes 15 components from the 77. They correspond to use cases “branches” *{UC1.1} Manage Accounts* (from *{UC.1} Manage business support*), *{UC2.1} Configure cloud services* (from *{UC.2} Configure cloud service*) and *{UC7.1} Integrate business information* (from *{UC.7} Performs business activities*), which were considered that together whey compose an MMF.

The remaining components after outbound collapsing are included in the W2ReqComm package (**Figure 141**) and delivered to Team #B, which is responsible for delivering the corresponding software. The associations between outbound components infer dependencies and orchestration needs with other teams, in this case Team #A, and #C. They are reflected in an *A-type* Sequence diagram, which is also part of the W2ReqComm package.

Access control to company data from Apps


Team #B
{P1} Configurations
{P1.1} Accounts
{P1.2} Services
{P1.3} Security
{P2} Monitoring
{P3} Business Mgmt (Global)
{P5.1} DB
{C2.1.1.c} Services deployment processor
{C4.2.i} Configure data access interface
{C1.3.i} Consult users SLA data interface
{C2.5.d} Service level agreement data
{C1.1.1.i} Create user interface
{C1.1.2.i} Edit user interface
{C1.1.3.i} Disable user interface
{C1.2.i} Configure users profile interface
{C2.1.1.i} Install service interface
{C2.1.2.i} Edit service interface
{C2.1.3.i} Disable service interface
{C2.1.4.i} Update service interface
{C2.5.i} Define service level agreement interface
{C6.5.i} Generate service templates interface
{C6.5.c} Services templates processor
{C7.1.2.c} Information access configuration processor
{C6.5.d} Services templates data
{C7.1.2.d} Information access configurations
{C1.1.1.d} User data
{C7.1.1.i} Consults Information interface
{C7.1.3.i} Perform business notifications interface


Remote Equipment Analysis sequence diagram
Team A backlog
Team B backlog
Team C backlog
Factory IT manager
{C7.1.1.i} Consults Information interface
{C7.1.2.c} Information access configuration processor
{C1.2.i} Configure users profile interface
{C4.2.i} Configure data access interface
Select Equipment management service
Get_authen(user, password)
Get_authorization(consult equipment service, authen_token)
Get_equipments(CompanyID)
AccessAndRetrieve()
RetrieveEquipmentToApp
Display Equipment info

A given stakeholder accesses to the platform, via the web or mobile app and consults business indicators. The visualized data depends on the user permissions. This web app invokes the authentication-related components, which confirms the permissions and retrieves the data related to a given unit’s instance representation in the cloud.

Figure 141. W2ReqComm package for "Access company data" scenario

Inter-team communication and management

Each team issuing a given set of components with connection points summons other team representatives to a Scrum of Scrums meeting to discuss integration efforts. The example illustrated in **Figure 142** relates to the development efforts between three teams. In this case, the scenario is the same as the one included in the W2ReqComm.

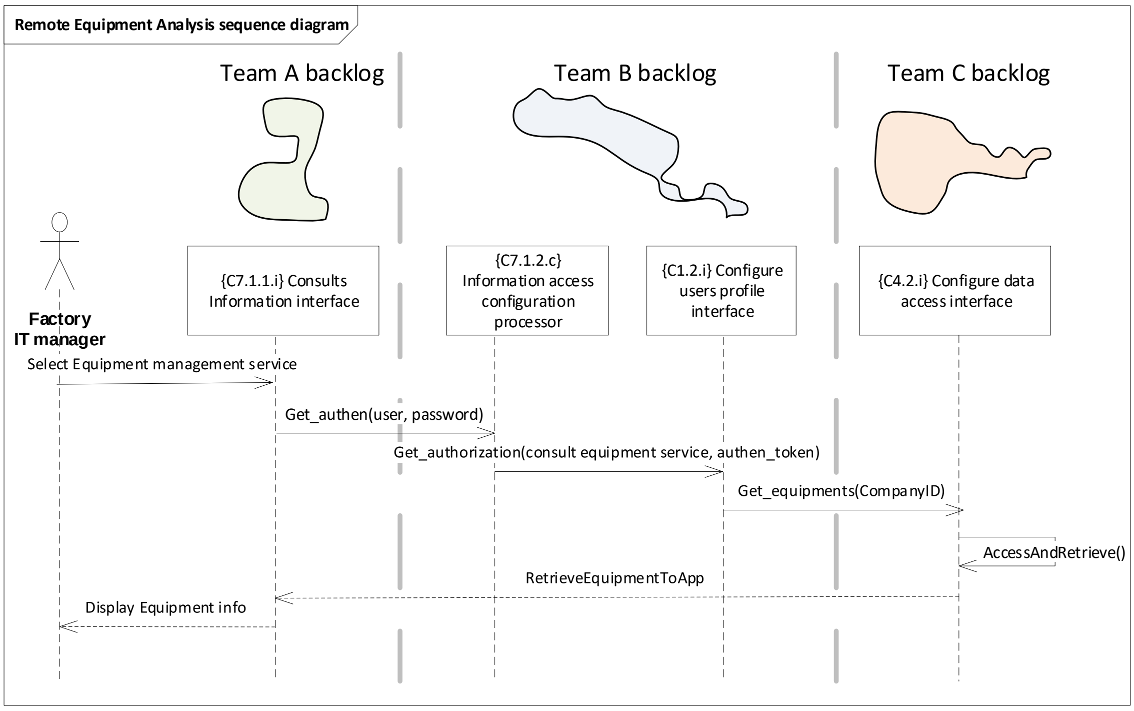

Remote Equipment Analysis sequence diagram
Team A backlog
Team B backlog
Team C backlog
Factory IT manager
{C7.1.1.i} Consults Information interface
{C7.1.2.c} Information access configuration processor
{C1.2.i} Configure users profile interface
{C4.2.i} Configure data access interface
Select Equipment management service
Get_authen(user, password)
Get_authorization(consult equipment service, authen_token)
Get_equipments(CompanyID)
AccessAndRetrieve()
RetrieveEquipmentToApp
Display Equipment info

Figure 142. An example scenario including inter-team management

In UH4SP, each team is developing a set of independent and loose-coupled microservices, exposed using a RESTful API. Team #B summons a Scrum of Scrums meeting before the end of their Sprint, with representatives from teams #A and #C, where the API is presented so other teams are informed on how to invoke the microservice. The API documentation is also available in a Swagger web page, where the remaining teams design and build services that consume the APIs. The coordination arenas regarding the coordination process are depicted in Table 35.

Table 35. Coordination arenas

Coordination arenas	Description
Scrum of scrums	Before the end of a Sprint, these meetings occurred whenever a work item with dependencies met a DoD condition, triggering a DoR to another team.
Wiki	Use of a Consortium shared Microsoft Sharepoint platform.
API documentation	Each team used a Swagger platform.
CoP meetings	Occasional and unplanned meetups between directly involved developers to discuss integration. Alternatively, the discussions were performed in chat groups using the Skype tool. Typically related to implementing a flow in the sequence diagram. The diagram was the basis for the discussion.

Team progress controlling

So far, it has been described how a logical architecture should be modularized in order to assign a team with a specific subsystem, how each team may derive their work items from the architecture, how they can communicate and coordinate with other teams.

This section describes controlling tasks performed by a Scrum team, namely Team #B in the UH4SP project, during the (Scrum) Sprints. In order to do so, a set of metrics were adopted, where they fit under these categories:

- *Earned Value Management* (EVM),
- Planning/Management,
- Development,
- Quality,
- *Stakeholders*

These categories allow controlling team progress encompassing metrics suitable for project managers, product owners, product managers, and the team members themselves. Additionally, some metrics may have dependencies with other ones from different categories.

EVM is a widely adopted metric within Project Managers for measuring the team performance based in their costs. EVM calculus precedes ASD approaches, which led to proposing some changes when in ASD projects, called AgileEVM (Sulaiman, Barton, & Blackburn, 2006). For planning/management, some metrics allow Product Owners having a clear understanding if the project is following a proper path. Development metrics refer to the software delivery process.

Quality metrics is a category where metrics are mainly measured by the team themselves. Finally, the Stakeholder metrics are more oriented for Product Managers as they are related to customers.

This control was based in gathered literature around this topic, presented in Table 36, and using such works in a setting where Team #B used the models from performing AMPLA.

The UH4SP project progress was monitored using the EVM system. Throughout Team #B's Sprints, the values for Actual Cost (AC), Earned Value (EV), Planned Value (PV), Cost Variance (CV), Schedule Variance (SV), Schedule Performance Index (SPI), Cost Performance Index (CPI) were monitored. The value for Budget Cost At Completion (BAC) was obviously previously defined before project kick-off. Only AC and PV measuring was completely independent from using AMPLA before Sprints.

Table 36. Agile metrics

EVM metrics (Sulaiman et al., 2006)	**Measurement**
Budget Cost At Completion (BAC)	Planned budget for the release
Actual Cost (AC)	Spent budget for the release
Earned Value (EV)	EV = APC × BAC
Planned Value (PV)	PV = PPC × BAC
Cost Variance (CV)	CV = EV – AC
Schedule Variance (SV)	SV = EV – PV
Schedule Performance Index (SPI)	SPI = EV / PV
Cost Performance Index (CPI)	CPI = EV / AC
Planning/Management metrics	**Measurement**
Business Value Delivered (BVD) (Hartmann & Dymond, 2006)	Business value provided by the delivered increment within the Sprint
Release Burndown (Hayes, Miller, Lapham, Wrubel, & Chick, 2014)	Which Product backlog items are 'done' and the remaining ones
Sprint Burndown (Hayes et al., 2014)	Which Sprint backlog items are 'done' and the remaining ones
Velocity (Hayes et al., 2014)	Story points delivered by Sprint
Planned Percentage Complete (PPC) (Sulaiman et al., 2006)	Nr. of Sprints performed / Nr. of Sprints planned
Actual Percentage Complete (APC)	Nr. of user stories 'done' / Nr. of user stories planned
Development metrics	**Measurement**

Lead Time (Mujtaba, Feldt, & Petersen, 2010)	Time between the item inclusion in backlog and to be 'done'
Queue Time (Mujtaba et al., 2010)	Time between the item inclusion in backlog and included in Sprint backlog
Processing Time (Mujtaba et al., 2010)	Time between the item inclusion in Sprint backlog and to be 'done'
Story Flow Percentage (Kupiainen, Mäntylä, & Itkonen, 2015)	% of completed story under development
Work in Progress (WIP) (Petersen & Wohlin, 2011)	Nr. of user stories under development
Quality metrics	**Measurement**
Defect Backlog (Staron, Meding, & Söderqvist, 2010)	Nr. of known defects still unresolved
Build Status (Janus, Schmietendorf, Dumke, & Jäger, 2012)	Nr. of builds performed in a Sprint
Test Coverage (Janus et al., 2012)	Code Covered by testing / Completed Code
Test Growth Ratio (Janus et al., 2012)	growth of the Test in relation to the growth of the Source Code
Deferred defects (Green, 2011)	Nr. of defects identified after going 'live'
Stakeholder metrics	**Measurement**
Customer Satisfaction	Survey filled by the customer about the experience
Feedback time	Time between customer and team for feedbacks

Since Actual Percentage Complete (APC) uses the number of user stories planned, which comes from the sum of derived user stories after performing the rules presented in section 6.4. Consequently, EV uses the APC value. Afterwards, CV, SV, SPI and CPI, which use the EV value, hence use the number of user stories derived within AMPLA.

Table 37 depicts the EVM controlling performed by Team #B's Project Manager (from the UH4SP team presented in section 6.2) throughout team's six (Scrum) Sprints. The Project Manager was able to compare EV evolution to AC and PV (Figure 143). Additionally, values of CPI and SPI from Table 37 allowed depicting the project status at the time of a given Sprint based in time and costs. In Figure 144, CPI and SPI values for each Sprint allow depicting whether in each Sprint the project was: (i) behind in time and costs ($0,5<SPI<1$ and $0,5<CPI<1$); (ii) bad times but good costs (0,5<SPI<1 and 1<CPI<1,5); (iii) good times but bad costs (1<SPI<1,5 and 0,5<CPI<1), and, finally, (iv) good times and costs (1<SPI<1,5 and 1<CPI<1,5).

Table 37. EVM controlling

EVM	**S#0**	**S#1**	**S#2**	**S#3**	**S#4**	**S#5**	**S#6**
Actual Cost (AC)	130	611,5	408	876	374	236	226
Earned Value (EV)	26,11	109,67	219,33	276,78	276,78	276,78	276,78
Planned Value (PV)	67,14286	134,2857	201,4286	268,5714	335,7143	402,8571	470
Cost Variance (CV)	-103,89	-501,83	-188,67	-599,22	-97,22	40,78	50,78
Schedule Variance (SV)	-41,03	-24,62	17,90	8,21	-58,94	-126,08	-193,22
Schedule performance index (SPI)	0,388889	0,816667	1,088889	1,030556	0,824444	0,687037	0,588889
Cost performance index (CPI)	0,200855	0,17934	0,537582	0,315956	0,740048	1,172787	1,22468

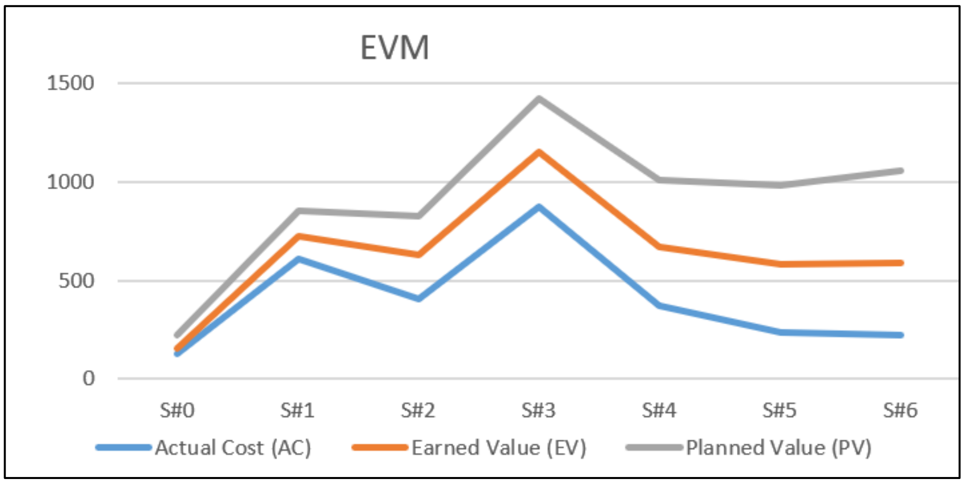


Figure 143. Evolution of AC, PV and EV

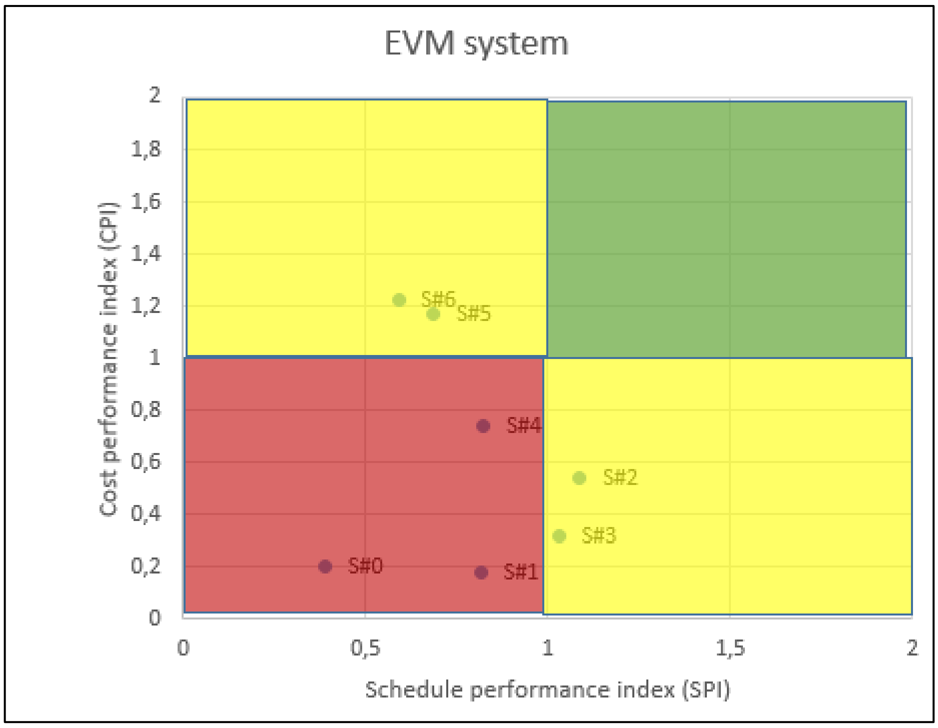


Figure 144. EVM monitoring

The progress of Sprints is monitored by the Product Owner (PO) using the Planning/Management metrics. Within these metrics, PO was able to measure Business Value Delivered (BVD), Release Burndown, Sprint Burndown, Velocity, Planned Percentage Complete (PPC), and the already mentioned Actual Percentage Complete (APC).

Release Burndown, Sprint Burndown, Velocity, and PPC are measured without any input from AMPLA. Figure 145 depicts these metrics measured by Team #B's PO.

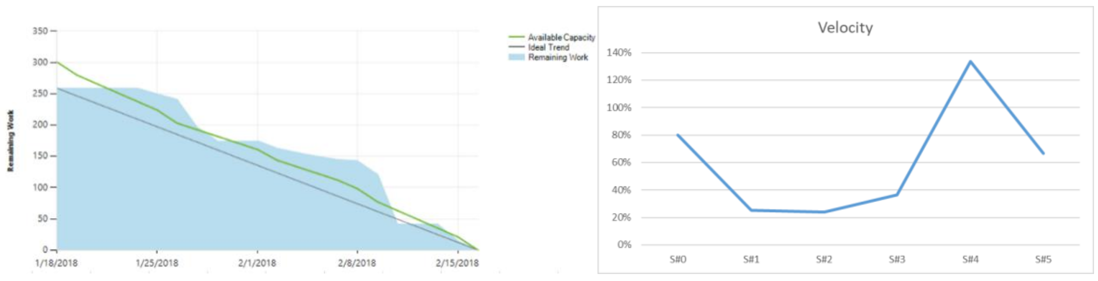


Figure 145. Sprint #3 burndown and team velocity measurement

As already used for EVM, APC uses the number of user stories planned, which are the sum of user stories derived after performing AMPLA.

The BVD value is a very important measure in ASD, since it focuses in the added value for the customers. As for its definition from the original publication from Hartmann and Dymond, BVD is measured using values of Net Present Value (NPV), Internal Rate of Return (IRR), and Return on Investment (ROI), by calculating Net cash flow per iteration (Hartmann & Dymond, 2006). In this research, the idea of measuring BVD is not by measuring the economic return of the delivered software, but rather the importance of delivered functionalities from each increment within the aimed product roadmap.

AMPLA provides the mechanism to measure BVD, by providing the linking between product objectives, functional requirements and models, and the backlog items. Therefore, whenever an increment was delivered at the end of a Sprint, as backlog items are marked as 'done', they are traced back to the objectives.

The control of the BVD throughout the Sprints is performed by the sum of the 'done' user stories. Such control is depicted in a Cumulative flow, like in Figure 146.

As previously described in section 4.4, the UH4SP project's objectives that were stated referred to:

(1) a unified view at the corporate (group of units) level;

(2) tools for third-party entities;

(3) in-plant optimization; and

(4) system reliability.

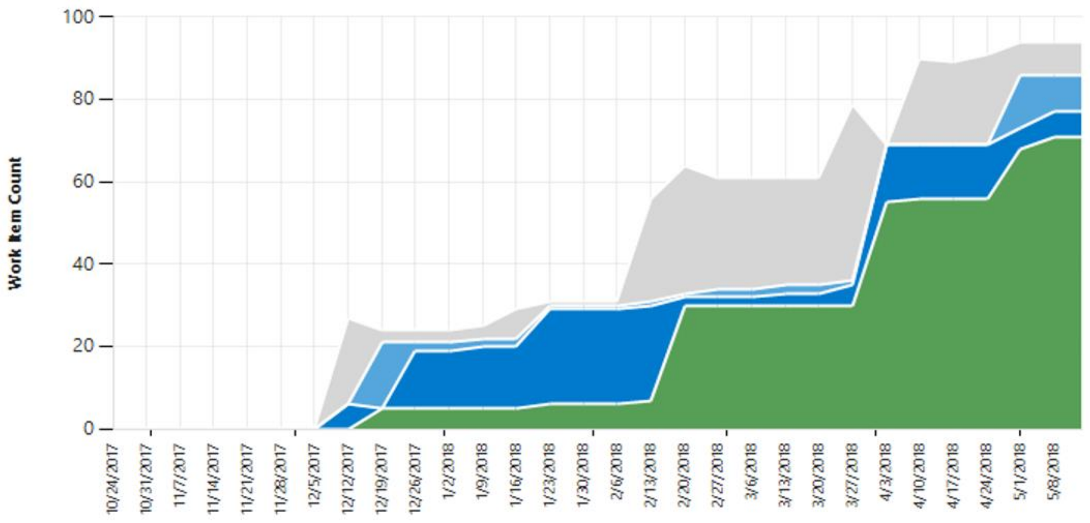


Figure 146. Cumulative flow

The previously derived backlog items from AMPLA, where the 61 user stories (available in Annex C), were grouped within the following features:

1. Configure User profile
2. Configure User Account
3. Perform Authentication
4. Manage Stakeholders
5. Manage Trucks
6. Manage Applications
7. Collaborative tool
8. Manage Work Tokens

These features were the input for aiming the project objectives. They contributed (in %) for the objectives, as depicted in Table 38. Each feature could contribute to more than one objective. Then, in addition to identifying features contributions to objectives, weights for the contribution were assigned (with an equal weight, e.g., if 3 features contributed to an objective, each one weighed 33%, if 4 features, each one weighed 25%, etc.).

Thus, using these weights, the user stories marked as 'done' in each Sprint were used for monitoring the evolution of the objective until it is achieved. The project's objectives, together with the gathered expectations elicited during requirements (section 4.4), are aligned with the "Objectives and Key Results" (OKR) (Doerr, 2018), where expectations describe the key results and how they are achieved. AMPLA provides the traceability mechanisms for linking the user stories and features to the expectations (and, consequently, the key results).

Table 38. Feature's contributions to project's objectives

	view at the corporate level	tools for third-party entities	in-plant optimization	system reliability
Configure User profile	20%	16,67%	25%	33,3%
Configure User Account	20%	16,67%	25%	33,3%
Perform Authentication	20%	16,67%	25	33,3%
Manage Stakeholders	20%	-	-	-
Manage Trucks	-	16,67%	-	-
Manage Applications	-	16,67%	-	-
Collaborative tool	20%	-	25%	-
Manage Work Tokens	-	16,67%	-	-

Table 39 depicts the evolution of BVD throughout the Sprints, namely by controlling the cumulative value (in %) of the project objective to be achieved. Namely, this way the PO monitored OKR's being met, as soon as the objective's BVD was 100% 'done'.

Table 39. Cumulative value of BVD

BVD (Cum)	view at the corporate level	tools for third-party entities	in-plant optimization	system reliability
stories Sprint 0	0%	0%	0%	0%
stories Sprint 1	0%	0%	0%	0%
stories Sprint 2	57%	54%	54%	72%
stories Sprint 3	77%	70%	71%	83%
stories Sprint 4	93%	78%	92%	99%

Regarding the Development metrics, Team #B measured Lead Time, Processing Time, Queue Time and the Work in Progress (WIP). The traceability provided by AMPLA allowed Team #B to hold the data of each feature, from its specification (before Sprint # 0 or already within the Sprints) until the respective user story is included in the Sprint Backlog and afterwards marked as 'done'. Table 40 depicts Lead Time, Processing Time, Queue Time. Table 41 depicts the WIP control, although in this case the traceability from AMPLA does not have impact on such measurement.

Table 40. Lead Time, Processing Time, Queue Time

Features	***Specified***	***Implemented Start***	***Implemented Finish***	***Lead Time***	***Queue Time***	***Processing Time***
Configure User profile	0	1	2	2	1	1
Configure User Account	0	1	2	2	1	1
Perform Authentication	0	1	2	2	1	1
Manage Stakeholders	0	1	3	3	1	2
Manage Trucks	0	1	2	2	1	1
Manage Applications	1	2	3	2	1	1
Collaborative tool	2	3	5	3	1	2
Manage Work Tokens	4	5	5	1	1	0

Table 41. Work in Progress

	Sprint #0	**Sprint #1**	**Sprint #2**	**Sprint #3**	**Sprint #4**	**Sprint #5**
WIP	5	16	21	11	4	9

Regarding stakeholder metrics, in UH4SP this measurement was performed based in customer satisfaction via a survey. Finally, regarding quality metrics, the measurement was basically based in registering bugs and number of performed tests. These two categories of metrics did not had any input from AMPLA.

Discussions

Use case-driven backlogs

Defining a hybrid approach (waterfall-based during initialization and Scrum-based during implementation), with the inclusion of artifacts modeling and documentation, strengthened the adoption of a Scrum process in a context as the one presented within the iFloW project. However, the entire adoption was a learning process, with advantages and disadvantages, which are detailed in this section.

This demonstration case showed the following advantages:

Requirements documentation waterfall-based – the fact that the Product Backlog was composed of 90 use cases led to a shared perception of the system complexity that originated the need to perform proper efforts in documenting the requirements. Thus, consuming efforts in almost exclusively for requirements engineering typically performed in waterfall approaches, in the initialization phase, allowed the project team to gain the required knowledge to properly implement a system of such complexity.

Implementation Scrum-based – within a customer perspective, Bosch was always aware of the system's current state of development. The iterative development, in form of Scrum Sprints, was crucial to manage Bosch's expectations, due to the periodical meetings and the incremental delivery of working software.

Use of a logical architecture – to enforce a proper organization on the set of components. The relationships among components suggest dependencies that may affect the implementation of functionalities and their inclusion in the Sprint Backlog.

On the other hand, it also showed the following disadvantages:

Effort estimation for use cases - the fact that it was a completely new development team (thus team velocity was unknown) and the need to frequently perform research spikes in order to overcome technological issues (for instance, related to GPS, EPCIS or SAP-OER) were the main obstacles for the estimation. In Scrum, estimation is performed using techniques such as planning poker, where user stories are estimated based in comparing efforts between other user stories. Due to the inexperience of the team, estimating the required effort for implementing use cases by comparing with other was itself a learning process. Such approach resulted in Sprint backlogs where use cases had not been implemented due to error in estimating and required conclusion in further Sprints, and where the effort estimating of the remaining use cases (as well as rework, whenever was required, and the spikes that were performed within almost every Sprints) required almost constant updates on every Sprint Closure and Planning meeting.

Dependence on negotiation for middleware use cases - collaborative coding among iFloW team members and service provider team members was required to implement middleware-related use cases. Most of the times the implementation required previous negotiation and agreements and the implementation did not progress at the desired velocity. The team's work reached a point where they had to pause and wait for those agreements, which resulted in the extension of use cases (and use cases with dependencies with them) through several Sprints.

Architecture modularization for inter-team management

Overall, in the ISOFIN project, there were clear advantages in using this approach:

(1) the teams experienced difficulties in interpreted the complete architecture, thus the modularization was required;

(2) since the project consortium was composed by Scrum teams, they easily understood the artifacts (*i.e.*, User Stories);

(3) User Stories were derived having an already designed logical architecture as input, allowing them to be properly aligned within reduced time.

Connection points between modules were identified and properly covered by User Stories but there was not enough time during this research work to assess that the team's efforts were in fact synched. Besides the identification of connection points, the authors believe that there is a vast area of progress in the topic of distributed Scrum teams.

Deriving work items

Although this case relates to the application of the method in Scrum teams, the authors believe that it is generic for being applied in other ASD methods, like XP or Kanban. In addition, it is perceived that such an approach is more helpful in LSA contexts, especially the sub-system partitioning and its further refinement. This approach is planned to be experimented in the future within LSA contexts.

By assigning a module of the architecture to a given team, we are basically defining a subset of the backlog, i.e., a Team Backlog. Hence, it is assumed that each team is responsible for developing a set of connected features. The predefinition of a subset (or a given feature) of the backlog is also present in approaches like Scaled Agile Framework (SAFe), Disciplined Agile Delivery (DAD), Enterprise Scrum, and Spotify model (or Squads/Tribes).

In the UH4SP project, the fact that it aimed to act within a complex ecosystem was taken in consideration for applying the 4SRS method. The 4SRS is a tool for tracing components and functional requirements models, moreover in large-scale contexts. We believe that the inclusion of modeling tasks in parallel with Sprints strengthened the Scrum process in the project. The entire research was a learning process, with advantages and disadvantages.

The advantages relate to the traceability between architecture and requirements, especially when changes occurred during Sprints. Previous experiences in using 4SRS and delivery of work items to Scrum teams were performed in BDUF contexts, which required around 9 months to derive a logical architecture composed with 107 components, which then were used to derive user stories. The emerging approach within this research allowed to first proposing a candidate architecture after 2 months. The architectural model was used as a shared understanding and provided a clear view of each entity's role within the project. When incrementally modeling the UML Use Cases in Sprints, the requirements package was also composed with wireframes, to enrich the discussion and benefited of user feedback. The components supported the project's pilot scenarios. However, the candidate architecture encompasses next releases in order to follow the product roadmap.

As a disadvantage, it is difficult to commit the entire consortium towards the approach. The candidate architecture was proposed and the sub-systems delivered to all entities, however it was not possible to compare the approach within all teams. Other teams used Scrum for their sub-systems but did not follow the approach for implementing their backlog, or did not used an ASD approach.

By applying the transformation rules, the team backlog was filled with themes, epics, use cases, user stories and acceptance criteria. The authors believe that there may be additional inputs to be included as transformation rules, like to foresee inclusion of backlog items that define the need for technical work tasks, knowledge acquisition tasks, prototyping, architectural spikes and development spikes.

6.5. Conclusions

Software development has been evolving towards an integrated MTS ecosystem, where a software team cooperates with other teams from other entities. LSA approaches address optimizing how scaled and distributed teams deliver software. However, it is still object of research, where inter-team coordination and boundaries are recognized challenges.

This chapter described a framework for distributed agile teams framework based in a logical architectural artefact. It includes inputs from a set of artefacts (logical architecture, UML Use Cases and *A-type* Sequence diagrams) to address architecture modularization and TPB assignment, dependencies derivation, and inter-team communication and coordination.

AMPLA defined a process for eliciting and analyzing requirements, and deriving logical architectures. The proposed framework (cf. Section 6.3) provided artefacts, roles and events for addressing architecture modularization, requirements communication and distributed team coordination.

This research's limitation is that validation was based only by application from the method's designers. Without proper training in AMPLA process, the 4SRS method, etc., it would be difficult for any project consortium to design the artefacts at this state of the theory's maturity. Thus, the research lacks a validation in observing "independent" teams.

Additionally, the distributed agile teams management framework is very dependent in prior execution of AMPLA's V-Model, since it uses the logical architecture artefact, but also, for instance, *A-type* Sequence diagrams for inter-team coordination.

The theory was used in a project where an analysis team was responsible for performing AMPLA, modularize the architecture and assign TPB. Then, each team was responsible to deliver the software and manage coordination efforts. When each team starts defining requirements from the beginning, it is required to propose new theories in inter-team cooperation, like architecture co-design, for instance.

This chapter essentially described the thesis' contribution to the process management at large-scale topic. As key results for this topic, the following were proposed in this chapter:

- A framework for addressing modularization, communication and coordination of MTS in a LSA setting;
- Concerns identification for modularization that traverses the logical architecture;
- Communication formats between teams using the component diagrams;
- Coordination dependencies depicting between teams from the interface components;
- Rules for writing User Stories statements;
- Rules for deriving required Product Backlog Items.

At this point, the models in an LSA setting started at the business-level and include the "just-enough" information for deriving a logical architecture model able to be used for specifying microservices and to define scope for a team in form of a product backlog. The model abstraction-level decreased until a service-level (in case of microservices) for the product specification, but also from a concrete work to be delivered (in form of a backlog item) for the process management. Thus, Chapter 7 describes the findings and outputs of this research, by discussing results regarding how AMPLA fully supports model derivation and abstraction-level decreasing throughout the SDLC.

Further reading

For architecture modularization and team assignment, there should be a proper acquaintance with concepts of Software-intensive systems of systems (SoS) (Maier, 1998), multiteams systems (MTS) (Mathieu et al., 2001) or feature teams (Larman & Vodde, 2008). Techniques examples for proposing modules are the Minimum Marketable Feature (MMF) (Denne & Cleland-Huang, 2003) and an elementary business process (EBP) (Larman, 2004) for a module minimum size.

For inter-team communication, Parnas' Principles (Parnas, 1972) suggests information format requirements, Conway's law suggests on the communication structure (Conway, 1968), and knowledge sharing is promoted with Communities of Practice (CoP) (Paasivaara & Lassenius, 2014). (Paasivaara & Lassenius, 2014).

For inter-team coordination, Coordination arenas (Dingsøyr, Moe, Fægri, & Seim, 2018; Dingsøyr, Rolland, Moe, & Seim, 2017) are tools for teams to expose their findings.

For understanding the different levels of approaches to compose a product backlog, Dean Leffingwell presents the types of requirements information in product backlogs in the book "Agile Software Requirements" (Leffingwell, 2010).

References

Ahmad, M. O., Lenarduzzi, V., Oivo, M., & Taibi, D. (2018). Lessons Learned on Communication Channels and Practices in Agile Software Development. In *2nd International Conference on Lean and Agile Software Development (LASD'18). Colocated with the Federated Conference on Computer Science and Information Systems*. Poznań, Poland.

Ambler, S. (2007). Agile Model Driven Development (AMDD). *XOOTIC MAGAZINE, February*.

Bjørnson, F. O., Wijnmaalen, J., Stettina, C. J., & Dingsøyr, T. (2018). Inter-team Coordination in Large-Scale Agile Development: A Case Study of Three Enabling Mechanisms. In *Agile Processes in Software Engineering and Extreme Programming. Proceedings of XP18* (pp. 216–231). Springer, Cham. https://doi.org/10.1007/978-3-319-91602-6_15

Cao, L., Mohan, K., Xu, P., & Ramesh, B. (2004). How extreme does extreme programming have to be? Adapting XP practices to large-scale projects. In *37th Annual Hawaii International Conference on System Sciences* (p. 10 pp.). IEEE. https://doi.org/10.1109/HICSS.2004.1265237

Cho, J. (2009). A hybrid software development method for large-scale projects: rational unified process with scrum. *Issues in Information Systems*, *10*(2).

Cockburn, A. (2001). *Writing effective use cases, The crystal collection for software professionals*. Addison-Wesley Professional Reading.

Cohn, M. (2004). Advantages of user stories for requirements. *InformIT Network*.

Conway, M. E. (1968). How Do Committees Invent? *Datamation*, 28–31.

Denne, M., & Cleland-Huang, J. (2003). *Software by Numbers: Low-Risk, High-Return Development, 1st Edition*. Sun Microsystems Press.

Díaz, J., Pérez, J., & Garbajosa, J. (2014). Agile product-line architecting in practice: A case study in smart grids. *Information and Software Technology*, *56*(7), 727–748. https://doi.org/10.1016/j.infsof.2014.01.014

Dingsøyr, T., Bjørnson, F. O., Moe, N. B., Rolland, K., & Seim, E. A. (2018). Rethinking coordination in large-scale software development. In *Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering - CHASE '18* (pp. 91–92). New York, New York, USA: ACM Press. https://doi.org/10.1145/3195836.3195850

Dingsøyr, T., & Moe, N. B. (2013). Research challenges in large-scale agile software development. *ACM SIGSOFT Software Engineering Notes*, *38*(5), 38–39.

Dingsøyr, T., & Moe, N. B. (2014). Towards Principles of Large-Scale Agile Development: A Summary of the workshop at XP2014 and a revised research agenda. In *Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation. XP 2014*. Springer Cham. https://doi.org/10.1007/978-3-319-14358-3_1

Dingsøyr, T., Moe, N. B., Fægri, T. E., & Seim, E. A. (2018). Exploring software development at the very large-scale: a revelatory case study and research agenda for agile method adaptation. *Empirical Software Engineering*, *23*(1), 490–520. https://doi.org/10.1007/s10664-017-9524-2

Dingsøyr, T., Rolland, K., Moe, N. B., & Seim, E. A. (2017). Coordination in multi-team programmes: An investigation of the group mode in large-scale agile software development. In *Procedia Computer Science* (Vol. 121, pp. 123–128). Elsevier. https://doi.org/10.1016/J.PROCS.2017.11.017

Doerr, J. (2018). *Measure What Matters: How Google, Bono, and the Gates Foundation Rock the World with OKRs*. Portfolio/Penguin.

Green, P. (2011). Measuring the Impact of Scrum on Product Development at Adobe Systems. In *2011 44th Hawaii International Conference on System Sciences* (pp. 1–10). IEEE. https://doi.org/10.1109/HICSS.2011.306

Hartmann, D., & Dymond, R. (2006). Appropriate Agile Measurement: Using Metrics and Diagnostics to Deliver Business Value. In *AGILE 2006* (pp. 126–134). IEEE. https://doi.org/10.1109/AGILE.2006.17

Hayes, W., Miller, S., Lapham, M. A., Wrubel, E., & Chick, T. (2014). *Agile Metrics: Progress Monitoring of Agile Contractors*.

IIBA. (2017). *Agile Extension to the BABOK Guide v2*. International Institute of Business Analysis.

Jacobson, I., Spence, I., & Bittner, K. (2011). *Use case 2.0: The Definite Guide*. Ivar Jacobson International.

Janus, A., Schmietendorf, A., Dumke, R., & Jäger, J. (2012). The 3C approach for agile quality assurance. In *Proceedings of the 3rd International Workshop on Emerging Trends in*

Software Metrics (WETSoM) (pp. 9–13). IEEE. https://doi.org/10.1109/WETSoM.2012.6226998

Jeon, S., Han, M., Lee, E., & Lee, K. (2011). Quality attribute driven agile development. In *9th International Conference on Software Engineering Research, Management and Applications (SERA)* (pp. 203–210). IEEE. https://doi.org/10.1109/SERA.2011.24

Kniberg, H., & Ivarsson, A. (2012). *Scaling agile@ spotify*.

Kruchten, P. (1995). The 4+1 View Model of Architecture. *IEEE Software*, *12*(6), 42–50. https://doi.org/10.1109/52.469759

Kupiainen, E., Mäntylä, M. V., & Itkonen, J. (2015). Using metrics in Agile and Lean Software Development – A systematic literature review of industrial studies. *Information and Software Technology*, *62*, 143–163. https://doi.org/10.1016/J.INFSOF.2015.02.005

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition*. Addison Wesley Professional.

Larman, C., & Vodde, B. (2008). *Scaling lean & agile development: thinking and organizational tools for large-scale Scrum*. Pearson Education.

Leffingwell, D. (2010). *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. Addison Wesley Longman.

Maier, M. W. (1998). Architecting principles for systems-of-systems. *Systems Engineering*, *1*(4), 267–284. https://doi.org/10.1002/(SICI)1520-6858(1998)1:4<267::AID-SYS3>3.0.CO;2-D

Mathieu, J. E., Marks, M. A., & Zaccaro, S. J. (2001). Multiteam Systems. In *Handbook of Industrial, Work & Organizational Psychology - Volume 2: Organizational Psychology* (pp. 289–313). 1 Oliver's Yard, 55 City Road, London EC1Y 1SP United Kingdom: SAGE Publications Ltd. https://doi.org/10.4135/9781848608368.n16

Moe, N. B., & Dingsøyr, T. (2017). Emerging Research Themes and updated Research Agenda for Large-Scale Agile Development: A Summary of the 5th International Workshop at XP2017. In *Proceedings of the XP '17 Workshops*. ACM.

https://doi.org/10.1145/3120459.3120474

Moe, N. B., Olsson, H. H., & Dingsøyr, T. (2016). Trends in Large-Scale Agile Development: : A Summary of the 4th Workshop at XP2016. In *Proceedings of the Scientific Workshop Proceedings of XP2016 on - XP '16 Workshops* (pp. 1–4). New York, New York, USA: ACM Press. https://doi.org/10.1145/2962695.2962696

Mujtaba, S., Feldt, R., & Petersen, K. (2010). Waste and Lead Time Reduction in a Software Product Customization Process with Value Stream Maps. In *2010 21st Australian Software Engineering Conference* (pp. 139–148). IEEE. https://doi.org/10.1109/ASWEC.2010.37

OMG. (2009). OMG Unified Modeling Language (OMG UML),Superstructure version 2.2. Object Management Group.

Owen, J. (2016). *Global Teams: How the best teams achieve high performance*. FT Publishing International.

Paasivaara, M., & Lassenius, C. (2014). Communities of practice in a large distributed agile software development organization – Case Ericsson. *Information and Software Technology*, *56*(12), 1556–1577. https://doi.org/10.1016/J.INFSOF.2014.06.008

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053–1058. https://doi.org/10.1145/361598.361623

Petersen, K., & Wohlin, C. (2011). Measuring the flow in lean software development. *Software: Practice and Experience*, *41*(9), 975–996. https://doi.org/10.1002/spe.975

Power, K. (2014). Definition of ready: An experience report from teams at cisco. In *Agile Processes in Software Engineering and Extreme Programming. XP 2014*. Springer Cham. https://doi.org/10.1007/978-3-319-06862-6_25

Rolland, K., Fitzgerald, B., Dingsoyr, T., & Stol, K.-J. (2016). Problematizing Agile in the Large: Alternative Assumptions for Large-Scale Agile Development. In *37th International Conference on Information Systems (ICIS)*. Dublin: AIS.

Schwaber, K., & Beedle, M. (2001). *Agile Software Development with Scrum*. Upper Saddle

River: Prentice Hall.

Staron, M., Meding, W., & Söderqvist, B. (2010). A method for forecasting defect backlog in large streamline software development projects and its industrial evaluation. *Information and Software Technology*, *52*(10), 1069–1079. https://doi.org/10.1016/J.INFSOF.2010.05.005

Sulaiman, T., Barton, B., & Blackburn, T. (2006). AgileEVM - Earned Value Management in Scrum Projects. In *AGILE 2006* (pp. 7–16). IEEE. https://doi.org/10.1109/AGILE.2006.15

PART IV – CONCLUSIONS

Chapter 7 – Conclusions

This chapter concludes this thesis. It describes the overall focus of the conducted work. Additionally, it synthesizes the research efforts as well as the scientific results of this thesis. Finally, this chapter ends with a set of proposed future work.

Cap 7 – Conclusions

7.1. Focus of the Work

The use of models throughout the software development lifecycle (SDLC) – typically, starting from identification of business needs or opportunity, then requirements, design, implementation, testing and deployment – reflects the knowledge that stakeholders possess, at a given phase or stage, about the solution under development. As the SDLC evolves, models typically include more detail on software solution behavior rather than the business setting where the solution will deploy. In model-driven development (MDD), this situation is stated as a decrease of model abstraction.

With the adoption of agile software development (ASD) approaches, by promoting iterative and incremental development and collecting feedback and learning for continuous adaptation, modeled artifacts decrease in abstraction but evolve in an incremental and continually updated way. Tracing such model evolution is needed so they are able to fulfill their purpose, which is to help teams develop software.

While ASD have been increasingly adopted by organizations[9], specifically at a team level, its usage in wider contexts – e.g., multiple and distributed teams or global development – has led to developing approaches for implementation at scale, which, as described in Chapter 2, are still object of research, with some emphasis in planning and inter-team coordination. Although acknowledging the importance of architecture in managing inter-team processes in an LSA context, these approaches lack of a structured approach for using such information to manage the software delivery process. Models are about presenting an abstraction of reality towards a shared understanding of the problem, but a proper analysis allows depicting their input in assigning work, derive dependencies, and manage inter-team communication and coordination.

Modeling a system includes multiple viewpoints regarding the architecture. A well-known example is the "4+1 framework" which addresses Logical, Process, Development, Physical and ("plus") Scenarios viewpoints. The architectural lifecycle throughout a project encompasses these viewpoints, thus all should be addressed during design. This research focuses in the *logical* one.

[9] VersionOne, "Annual State of Agile Report". https://www.stateofagile.com/

The logical viewpoint regards designing software components and organizing them so the software meets the business needs. Such organization allows depicting the scope of functionalities that a team needs to develop and, in case of multiple or distributed teams, allows depicting the scope of each team and respective interface needs. There is thus an opportunity for research is supporting the model evolution, namely the logical architecture one, in order to provide the mechanisms for ASD and LSA settings. Mechanisms include architecture emerging and refinement, evolution traceability, relationships with other viewpoints, continuous architecting promoted by "the power of small" (i.e., microservices) and baseline support for multiple teams management (namely in LSA contexts).

Previously in Chapter 1, this PhD's research question was presented, stating:

"How to adopt logical architectures in agile large-scale projects?"

This thesis presented the Agile Modeling Process for Logical Architectures (AMPLA), an Agile Modeling (AM) oriented process composed by UML diagrams (Sequence, Use Cases and Component). AMPLA uses agile practices in order to deliver small increments (of a requirements package) and to promote continuous customer feedback. The proposed AM process also includes a candidate architecture and further requirements refinement in parallel with a software increment delivery. The refinement ranges from component design to a microservices architecture. AMPLA supports that the model abstraction level decreases throughout the process, providing traceability for easing changes that may occur.

AMPLA uses the techniques as well as its outputs in modeling AMPLA artifacts, like (1) Lean Startup, Design Thinking, Domain-driven Design, BizDev and Kent Beck's 3X in requirements modeling; (2) Ambler's Agile Modeling, Lean Inception for the candidate logical architecture design; (3) Sprint zero for architecture modularization; (4) Use Cases 2.0, Scrum, XP and Kanban backlog structures; (5) DevOps for the microservices architecture and its deployment.

This research aimed answering the research question by addressing three research objectives. They are now revisited, as well as describing the results that resulted in their achievement.

O1: To develop an approach capable of deriving logical architectures in order to establish the initial requirements that are passed on to agile development teams.

In chapter 6, two possible types of approaches using a logical architecture were presented: (1) a use case-driven backlog, using UML Use Case models for defining the Product Backlog; and (2) a user story-driven backlog, using a set of UML diagrams (Use Case, Sequence and Components), described in Chapter 4, which can derive the "agile" backlog items. Related to the later, firstly it was presented a set of rules for deriving user story statements (in form of "As a...I want to...in order to..."). Then, the approach was revisited aiming deriving additional backlog items. After performing AMPLA, the resulting logical architecture was used in order to derive "agile" product backlog items, based in a widely adopted structure for backlogs (Leffingwell, 2010). Such derivation was enabled by a set of defined transformation rules, in short:

- A *theme* may be derived by the identified packages;
- an *epic* is directly derived from a Use Case;
- a *user story* creation should be based by each of the scenarios from a use case;
- *User Stories Details* are depicted by the software components responsible for executing the given functionality, identified during the 4SRS method execution. Another important aspect of the detail may be the context of use, which is depicted by scenarios, modeled in sequence diagrams ;
- *Acceptance Criteria* is described in the use case description, namely the post condition.

O2: To adopt flexibility and agility mechanisms in the refinement of logical architectures throughout the iterations of ASD teams.

In Chapter 5, AMPLA was described as a supporting mechanism for proposing a candidate logical architecture, using a set of "just-enough" requirements. Then, supported by the 4SRS method, the candidate logical architecture iteratively evolves as specific subsets of the architecture are analyzed within Scrum Sprints and its components are refined.

Agile architecting is about continuous design and evolution of the solution, by acknowledging the architecture's evolution but assuring such evolution is not endangered by business decisions. Included within the AMPLA method, Chapter 5 presented how continuous architecting (CA) is supported by supporting change-impact analysis (CIA) practices for eventual changes proposals. Due to the model traceability supported by AMPLA, CIA was able to depict impacts in concerns as architecturally significant requirements (ASR), Quality characteristic, Business and customer value of the requirement, Which components are affected, Compliance with standards, Requirements emerge, and Managing architectural debt.

O3: To develop an approach oriented for continuous architecting, aiming to specify microservices logical architectures (MSLA), identifying them and their interfaces.

As described in O2, design in ASD continuously evolves. In previous objectives the focus was to propose an architecture from scratch and incrementally refine it. However, continuous architecture includes assuring the architecture eases a proper maintenance. By making use of the "power of small", microservices architectures (MSA) style have been adopted in software development, on one hand, in developing cloud applications, promoted by the service's independent deployment and maintenance and, on the other hand, using such independence and "smallness" for promoting automation in a continuous integration, continuous delivery and DevOps processes.

In Chapter 5, by using an adapted version of the 4SRS, a model for microservices logical architectures (MSLA) was derived and presented using SoaML diagrams. The described approach allows deriving the microservices' internal behavior, their data models, and the existing communications. The approach was described in opposite settings: (1) in an existing monolith decomposition setting, with upfront information about legacy systems, demonstrated in IMP_4.0 and ISMPM projects; and (2) in greenfield settings, where requirements emerged (using AMPLA), demonstrated in UH4SP project. Transiting from the logical architecture to SoaML diagrams was systematized in modeling procedures, in short:

- Service Participants with boundary definition, communication needs (Requests/Services and Ports)
- Service Architecture with boundary definition and communication (service requests that the service performs)
- Service Capabilities with boundary definition and data model
- Service Interfaces with communication specifications

O4: To use logical architectures to manage assignment and orchestration process in LSA projects

Software development settings composed with multi-teams, distributed or co-located, have an additional concern in the SDLC to manage and coordinate work and the software delivered by them. This process of delivering software, at large-scale – i.e., using more than one development team, often geographically distributed - faces issues of dependencies, boundaries, coordination and/or synchronization. In ASD settings, often referred as '*large-scale agile development*' (LSA), such task had to be rethought, where team collaboration heavily relies in a proper addressing of communication, trust and alignment. A logical architecture model was able to provide important insights for such concerns, by the organization of the components assigned for teams and the component's relationships that may relate with required interfaces between teams.

In Chapter 6, a framework for distributed agile teams was presented, addressing the architecture modularization, team assignment, dependencies, requirements modelling towards coordination and communication within distributed teams.

Modularization phase is composed by the following principles: *Identify Modules* of the architecture; *Assign Modules to Teams*, which originates a subset of work items; *Set Module Boundaries*; and *Define Module Size*. *Derive Dependencies* and *Manage Communication* using predefined channels and periodicity, are present in Communication and Coordination phases. Communication phase also includes *Model W2ReqComm Package* for a multiview requirements package. Coordination phase also includes *Manage Inter-team Coordination*, composed by unplanned and planned events.

AMPLA is a modeling approach covering the evolution of agile architecting, from grooming to software delivery stages. Namely, AMPLA covers some modeling and design tasks, including initial inputs, candidate architecture design, incremental refinement, continuous architecting and change-impact analysis, microservice logical architecture design and deployment, and multi-teams and multi-backlogs management. This thesis focused in the contribution of AMPLA to a set of research topics necessary to support covering the SLDC, which are now described.

Agile modeling

AM, as the name states, is about modeling in ASD settings. While other settings, like plan-driven (e.g., Waterfall) address the entire modeling in a specific stage of the SDLC, AM advocates modeling throughout the SDLC, encompassing its evolution as further details of software emerge. It relates to the opposite of "Big Design Upfront" (BDUF), aiming to prevent modeling of "You Ain't Gonna Need It" (YAGNI) features.

AMPLA contributes to AM by providing a stepwise model evolution, where different software models (Sequence Diagrams, Use Cases, and Logical Architectures) are derived in succession and properly aligned with each other in a V-Model manner.

AMPLA proposes the design of a candidate version of the logical architecture during grooming stages of the ASD, afterwards providing mechanisms for the architecture refinement during iteration cycles (e.g. Scrum Sprints). Additionally, AMPLA provides the traceability between the models (Sequence Diagrams, Use Cases, Logical Architectures), as well as between the models and the Product Backlog Items (Epics, User Stories, Acceptance Criteria, etc.).

Agile requirements engineering

Just like AM, "agile requirements engineering"[10] differs from plan-driven by not being sticked in a specific stage of the SDLC, but rather throughout the SDLC. In ASD contexts, requirements engineering activities are still in a relatively early phase of development. However, there are change in their timings and how they are used. In ASD frameworks, like Scrum, XP, Kanban, SAFe, LeSS, Scrum@Scale, Nexus or Spotify Squad, the requirements are included in a product backlog, which then drives the development process, thus most of the RE activities are performed earlier.

For the scope of this thesis, requirements engineering outputs aim gathering the information for enabling AMPLA at providing a candidate logical architecture. Towards such aim, this research proposed an approach that addressed requirements elicitation, analysis and documentation as they emerge in a stepwise and traceable way, called "Decomposing User Agile Requirements ArTEfacts" (DUARTE).

DUARTE included inputs from ASD practices such as Lean Startup, Design Thinking, Domain-driven Design, BizDev and Kent Beck's 3X. By including inputs from these practices, DUARTE aims at assuring that the gathered and modeled requirements have only the "just-enough" detail for enabling AMPLA at deriving a candidate logical architecture.

AMPLA does not require that requirements engineering be based in DUARTE approach. In fact, it only requires that an output is a UML Use Case model, in order to perform the Four Step

[10] Requirements engineering performed in ASD settings is sometimes referred as "agile requirements engineering", but this term is not consensual since many authors state that "requirements engineering" techniques are the same whether in ASD or non-ASD settings. For the sake of this thesis, we use the term "agile requirements engineering" whenever techniques are performed in ASD settings.

Rule Set (4SRS) method. In chapter 4, AMPLA used both upfront and emerging requirements activities.

Agile logical architecting

Discussing agile architecting encompasses design evolution regarding Logical, Process, Development, Physical and Scenarios viewpoints, alongside with defining how the relationships between viewpoints also evolve. This research proposed an Agile Architecting Lifecycle (AAL) encompassing software design that evolves from architectural, mechanistic, and detailed design to development and deployment. The AAL pathway goes through three stages: Grooming, Backlog and Delivery. AAL pathway includes description of Context, Functionalities, a Candidate Architecture and, then, a Refined Architecture.

AMPLA focuses in the evolution of the logical architecture. The agile logical architecting approach is composed by firstly proposing a candidate version, which is later refined. The evolution of the architectural models between the candidate and the refined one is supported by the 4SRS method.

This thesis also included how AAL uses remaining viewpoints. In terms of the Context, architecture viewpoints relate to domain, enterprise, business and information systems where the project is scoped. The logical viewpoint is a central issue in Candidate Architecture and afterwards a Refined Architecture. Finally, the Delivery relates to the deployment viewpoint. The AAL pathway described in this thesis encompasses the inputs and outputs within each relationship between viewpoints throughout the pathway.

Change-impact analysis

As AAL addresses design throughout the SDLC, the logical architecture model evolves as the solution emerges. The architecture model evolution relating to Context, Candidate, Refined and Delivery stages. This makes design as a continuous process.

AMPLA supports the continuous process by using the 4SRS method for refining the architecture in an incremental way, allowing it to emerge as iterations occur. During these iterations, when stakeholders review current software development status, AMPLA uses its traceability capability, promoted by the 4SRS method, between requirements and architecture, towards managing changes and/or new requirements, to analyze its impact on the solution architecture, as well as manage any technical debts.

Microservices design

One of the principles for continuous architecting is to leverage the “power of small”. In that sense, microservices architectures (MSA) is one of the most common situations when companies adopt continuous architecting processes, characterized by small, interconnected and independent services.

AMPLA was used in order to derive a microservices logical architecture from functional requirements. In the refinement process, domains (DDD) were identified within the architecture and afterwards iteratively refined, enabling deriving microservices specifications, afterwards modeled in SoaML diagrams.

From models to (agile) backlog items

Within the Delivery stage of the SDLC, the software increment to be delivered in each iteration is defined by a Product Backlog. This backlog is composed by a set of items that define the work to be done during the iteration.

This research focused in defining backlogs, and its composing items, from requirements models (namely UML Use Cases, Components and Sequence Diagrams) using rules that assure the backlog items cover the gathered requirements. The derived backlog was filled with themes, epics, use cases, user stories and acceptance criteria.

Agile multi-teams management

The process of delivering software using more than one development team, often distributed, faces issues of dependencies, boundaries, coordination and/or synchronization. Architectures are an artefact capable of supporting a set of coordination decisions.

In this research, the derived logical architecture from AMPLA’s V-Model is the foundational artefact for a distributed agile team framework. The framework addresses the architecture modularization, team assignment, dependencies, requirements modelling towards coordination and communication within distributed teams.

7.2. Synthesis of the research efforts

It was at somewhat in the mid of 2013, when the ISOFIN project was entering its implementation stage, that we quickly identified that teams would have some difficulties in using the design specifications and modeled artefacts – based in a logical architecture composed with 105 components after being derived from the 4SRS method – in their development processes. Due to the complexity of the project, and hence the complexity of the logical architecture, the consortium decided to divide work between three teams from different entities, but even so they struggled to define each Product Backlog. At this point, the project lacked a systematic approach to define the backlog items as well as pointing out interfaces between teams.

These obstacles quickly arose as a research opportunity for this work. By defining a research method using the design science research (DSR), the objective was to base a research objective (O1) in the described need for backlog structuring (from models) and interfaces identification and address it using a DSR cycle from Kuechler and Vaishnavi's framework (presented in Chapter 1). Such results were achieved under transformation steps (in Chapter 6), however during the research evaluation we identified that the problem facing was broader than just deriving backlog items from logical architectures. Thus, this DSR cycles was not concluded for that time and additional research objectives were defined (O2, O3 and O4).

For addressing each research objectives, one different DSR cycle was conducted. However, since each DSR cycle was independent from another and thus a cycle did not rely in another cycle's results, they could be conducted in parallel.

As each DSR cycle was defined, their awareness was refined basing some literature review on combining architectures and agile frameworks, as well as technical reports on previous applications of the 4SRS method, among others, where the identified gaps were used to define the research goal for each DSR cycle and the criteria to complete them.

The development of these cycles used designed artefacts resulting from different demonstration cases, which relate to R&D projects. How demonstration case's contributions were organized towards the research objectives is synthetized in Table 42.

Table 42. Demonstration case's contributions towards the research objectives

Demonstration case	O1	O2	O3	O4	Observations
iFloW	X	X			Used use cases in backlogs (O1) and upfront modeling in ASD (O2), in opposition to ISOFIN and UH4SP
ISOFIN	X			X	Used user stories from models (O1) and modularization for interfaces identification (O4)
UH4SP	X	X	X	X	Derived a complete backlog structure (O1) in a complimentary way as performed in ISOFIN, emerging modeling using AMPLA (O2), modeling MSA in SoaML (O3) and defining inter-team coordination and communication (O4) in a complimentary way as performed in ISOFIN.
IMP_4.0			X		Defined an MSA (O3), but from an existing monolith rather than greenfield like the UH4SP
ISMPM			X		Defined an MSA (O3) from an existing monolith like IMP_4.0, but allowed to address additional patterns and deployment infrastructure, complimentary to UH4SP and IMP_4.0

As mentioned, the DSR cycles were performed in parallel, where a set of demonstration cases contributed for the development and evaluation of the artefacts. In general, a demonstration case was not specific to one DSR cycle (in exception from IMP_4.0 and ISMPM). The projects were used within different contributions for each research objective, but they overlapped in time between each other. The chronological order for the projects is the following:

- ISOFIN Cloud (2011-2014)
- iFloW (2014-2016)
- IMP_4.0 (2016-2017)
- UH4SP (2016-2018)
- ISMPM (2017-2018)

As depicted in the "Observations" column in Table 42, each project may contribute differently in the research objective. Their outcomes may contribute for a practice (cf. Section 7.3 for research contributes for details in the practices) but different perspectives in a given research objective. Namely, there were cases of similar contribution for different project settings (e.g., IMP_4.0 for breaking existing monoliths and UH4SP for greenfield projects), additional contributes as in a post iteration (e.g., ISOFIN defined inter-team management and backlog derivation approaches, and afterwards UH4SP used those approaches as basis and improved them, or ISMPM using approaches from UH4SP and IMP_4.0 as basis and additionally aligning deployment viewpoints), and opposite approaches (e.g., iFloW using use case-driven backlogs rather than user stories like ISOFIN and UH4SP).

These contributes – even the opposite ones – do not mean that one approach is best suited than other. Each described case can be seen as a possible approach to be included in adopted practices when teams are defining their development processes, as teams should define their processes by adopting practices best suited for them rather than blindly follow a given framework (Ambler & Lines, 2019; Jacobson, Ng, McMahon, Spence, & Lidman, 2013) – like Scrum, XP, Kanban, SAFe, LeSS, Spotify.

It is also worth referring that this research had intermediate progress control checkpoints. There is a "PhD Research Plan" report, dated 2015, that relates to the kick-off for the research. The "PhD Proposal" report, dated 2017, relates to initial findings and a deep understanding of the research process. Additionally, three doctoral consortiums allowed documenting the thesis evolution. First, the participation in the "2016 Interoperability For Enterprise Systems And Applications (i-ESA) Doctoral Symposium" documented some issues in adopting Scrum framework in distributed and interoperable settings, with focus in the results from iFloW. Afterwards, the participation in the "2016 Portuguese Software Engineering Doctoral Symposium (SEDES)", co-located with "International Conference on the Quality of Information and Communications Technology (QUATIC)", documented the research question as well as the results from iFloW and ISOFIN. Finally, a final participation in "2017 European Conference on Information Systems (ECIS) Doctoral Consortium)" updated the thesis progress, with equal focus in iFloW and ISOFIN. Other participations, namely related to projects such as IMP_4.0 and UH4SP, also allowed to discuss the research outcomes but are described in Section 7.3.

7.3. Scientific Outputs

In line with Hevner's design theory framework (cf. Chapter 1), this thesis aimed at contributing to both knowledge base space (i.e., added scientific theories and methods for "science" knowledge) and environment space (i.e., practices for adoption from organizations with software development teams). Due to the nature of the conducted research, namely by developing design science research based in demonstration cases, the designed artefacts, methods and processes were validated in both theoretical and practical way.

As for contributions to environment space, this thesis proposed the AMPLA process, composed with modeling practices that software development teams may adopt. The proposed design theory provided artefacts, roles and events for addressing architecture modularization, requirements communication and distributed team coordination. Thus, software teams may adopt AMPLA process for conducting software modeling throughout the SDLC – from a high-level, business-oriented, abstraction to a lower-level, at a microservice-level, model abstraction – in an aligned and traceable way. Additionally, it should be pointed out that, although this thesis proposed the AMPLA process, the same AMPLA process is composed with a set of practices that can be adopted in the SDLC independently, allowing flexibility in defining the best suitable software development process for a given environment.

As for contributions to knowledge base space, this thesis outputs a new theory on design artefacts for logical architecture's usage in LSA settings. As stated in Section 7.1, developing the theory regarding AMPLA included researching in topics such as modeling and design tasks, including initial inputs, candidate architecture design, incremental refinement, continuous architecting and change-impact analysis, microservice logical architecture design and deployment, and multi-teams and multi-backlogs management. The theory contributions are (1) an Agile Modeling Process for Logical Architectures, (2) a Process management at large-scale, and (3) an Incremental model refinement until service level.

Contribution 1: Agile Modeling Process for Logical Architectures

The evolution of the modeled artefacts that is required for ASD settings has as main concern to design what is "just enough" for addressing the problem, in opposition to BDUF. The proposed AMPLA process proposed a theory for using such "just enough" information in requirements, using DUARTE approach), in order to validate if all essential information for a candidate architecture was gathered.

The DUARTE approach defined a theory on defining how product development mindsets like Lean Startup, DDD, Design thinking, among others, affect requirements modeling. Before using such information in performing the 4SRS method, AMPLA encompassed a verification checkpoint to prevent any essential information was disregarded. This allowed preventing the candidate version of the logical architecture to have any YAGNI features.

Contribution 2: Incremental model refinement until service level

Microservices architectures (MSA) are an architectural style oriented towards modularization, where the idea is to split the application into smaller, interconnected services, running as a separate process that can be independently deployed, scaled and tested. Adopting microservices include specific concerns on design, development and deployment. Assuring MSA design aligned with business requirements needs to be supported by modeling methods that cover microservices principles.

This thesis proposed defining a method for deriving a microservices logical architecture (MSLA) from functional requirements. The method has as input an UML logical components diagram, enabling deriving microservices specifications, afterwards modeled in SoaML diagrams. Additionally, these diagrams were basis for discussing microservices principles.

This thesis included a method validation, both in greenfield settings (UH4SP project) and in breaking existing monoliths settings (IMP_4.0 and ISMPM projects).

Contribution 3: Process management at large-scale

Research in large-scale software development projects, or, in the case of this thesis, 'large-scale agile development' (LSA), relates to ASD practices for scaled settings in team's size, number of teams, number of lines of code, among others. These practices have the concern relating to business agility, role of architects, knowledge sharing and networks, inter-team coordination, etc. Although it was acknowledged the potential role of models – like the architecture – for promoting those concerns, there was a gap in a prescriptive approach for using architectural models as input (like components and their interfaces) for supporting dependencies, communication and coordination between teams.

The theory defines a logical architectural artefact as basis for managing the process of setting delivery boundaries, communicating the requirements, coordinating and synchronizing inter-teams work.

In an LSA setting, the candidate architecture is modularized and "presented" to multi-teams. This research demonstrated how teams coordinate, communicate and synchronize during their own model evolution, enabling the architecture model to refine incrementally in multi-modules in parallel throughout the project.

The inputs from these research contributions result from different demonstration cases. How demonstration case's inputs were organized towards the research contributions is synthetized in Table 43. As depicted in the "Observations" column in Table 43, each project may contribute differently in the research contribution. For instance, iFloW and UH4SP projects provide opposite inputs on modeling in ASD (or Agile Modeling), namely upfront modeling in ASD projects and emerging modeling preventing BDUF. Also, post iterations of a contribution, which aimed at revisiting, updating and improving the research contribution, like the case of ISOFIN in helping define rules for deriving user story statements from UML models and the case of UH4SP that improved these rules in order to derive additional product backlog items, such as themes, epics, user story details and acceptance criteria.

Table 43. Demonstration case's inputs towards the research contributions

Demonstration case	Contribution 1	Contribution 2	Contribution 3	Observations
iFloW	X			Use of models (upfront) in ASD projects
ISOFIN	X	X		Use of models (upfront) in ASD projects, Rules for deriving user story statements and inter-team dependencies
UH4SP	X	X	X	Agile (emerging) modeling using DUARTE and AMPLA, deriving agile backlog items (complimentary to ISOFIN), and uses 4SRS-MSLA in greenfield projects
IMP_4.0			X	Uses 4SRS-MSLA in an existing monolith project (with upfront requirements rather than emerging like the UH4SP)
ISMPM			X	Uses 4SRS-MSLA in an existing monolith project and discussed

In exception for ISMPM, each of the demonstration cases relate to a funded R&D project. Which is to say that this thesis work includes scientific outputs from ISOFIN, iFloW, UH4SP and IMP_4.0 projects.

In ISOFIN project, this research included developing and afterwards validating the applicability of using the 4SRS method for deriving the logical architecture and afterwards delivering it to distributed ASD teams in form of user stories and dependencies between them. Contributions such as **Contribution 1: Agile Modeling Process for Logical Architectures** and

Contribution 2: Process management at large-scale, are included work of the following project deliverables:

- M/D207 – ISOFIN Logical Architecture;
- M/D210 – Financial Domain Applications/Services Specifications

Namely, M/D207 – ISOFIN Logical Architecture documented the research work towards upfront modeling for **Contribution 1**, and M/D210 – Financial Domain Applications/Services Specifications regarding **Contribution 2** that documented the research work in modularizing the architecture, defining an approach to bound the modules scope, interfaces, and dependencies, and presenting transition rules for user story statements ("As a… I want to… In order to…") for usage in ASD projects.

In iFloW project, this research included developing and afterwards validating the applicability of using a requirements modeling in UML for use in an ASD project and backlog. Research work towards **Contribution 1: Agile Modeling Process for Logical Architectures** is included work of the following project deliverables:

- D4.4.2 - Specification of the model for experimental development;
- D5.3.8 – development of functionalities
- D6.7.9 – verification and validation of functionalities developed

Namely, report D4.4.2 - Specification of the model for experimental development documented the requirements process and the UML models. Reports D5.3.8 – development of functionalities, and D6.7.9 – verification and validation of functionalities developed documented the definition of the backlog and the results of the performed iterations (Scrum Sprints) which allowed perceiving how the iFloW project team used the derived backlog.

In UH4SP project, this research included developing the AMPLA, DUARTE and 4SRS-MSLA approaches and afterwards validating the applicability of using emerging (agile) modeling of requirements and candidate architecture, incremental refinement of the architecture following microservices principles and continuous architecting, and the delivery of product backlog items and multi-teams LSA process management. Contributions such as **Contribution 1: Agile Modeling Process for Logical Architectures**, **Contribution 2: Incremental model refinement until service level** and **Contribution 3: Process management at large-scale** are included work of the following project deliverables:

- D.3.1 – Functional and Technical Requirements Specification;
- D.3.2 – Technical and logical architecture;
- D3.3 – Service Specification For Material Reception And Shipment;
- D3.5 – Interoperability Between Platform And Services Requirements;
- D3.7 – Solution modelling;
- D4.1.1 – UH4SP Management Platform – Initial Version;
- D4.1.2 – UH4SP Management Platform – Final Version;
- D5.4 - Integration Services and Platform.

Namely, D.3.1 – Functional and Technical Requirements Specification report documented the results from applying the DUARTE approach, and D.3.2 – Technical and logical architecture report documented the results of applying AMPLA for candidate logical architecture design, both for **Contribution 1**. Reports D4.1.1 – UH4SP Management Platform – Initial Version, D4.1.2 – UH4SP Management Platform – Final Version and D5.4 - Integration Services and Platform documented, firstly, the architecture modularization, communication and coordination needs, and afterwards, the performed iterations (Scrum Sprints) which allowed perceiving how one of the UH4SP project team used the derived backlog, for **Contribution 3**. Finally, reports D3.3 – Service Specification For Material Reception And Shipment, D3.5 – Interoperability Between Platform And Services Requirements and D3.7 – Solution modelling regarding documented the results from applying the 4SRS-MSLA and the microservices modeling in SoaML, for **Contribution 2.**

In IMP_4.0 project, this research included developing 4SRS-MSLA and afterwards validating the applicability of the MSLA in an existing monolith setting. Research work towards **Contribution 2: Incremental model refinement until service level**, is included work of the following project deliverables:

- D.1.4 – Functional requirements specifications – initial version;
- D.1.5 – Functional requirements specifications – final version;
- D1.8 – Traceability mechanisms for production management;
- D1.9 – IMP_4.0 logical architecture – initial version;
- D1.10 – IMP_4.0 logical architecture – final version;
- D1.11 – IMP_4.0 platform services specification.

Namely, reports D.1.4 – Functional requirements specifications – initial version and D.1.5 – Functional requirements specifications – final version documented the requirements engineering task for the existing monoliths setting. The D1.8 – Traceability mechanisms for production management report documented the development of the 4SRS-MSLA method. Finally, reports D1.9 – IMP_4.0 logical architecture – initial version, D1.10 – IMP_4.0 logical architecture – final version and D1.11 – IMP_4.0 platform services specification documented the resulting MSLA, in form of different SoaML diagrams. All these reports documented research work towards **Contribution 2**.

As mentioned in Chapter 1, in the case of the IMSPM, because it is an internal project, the only available documentation is in form of a MSc thesis, that can be found in: Amaral, José Diogo Coelho, "The evolution of monolithic architectures to microservice-based architectures" (free translation of "A evolução das arquiteturas monolíticas para as arquiteturas baseadas em microserviços"), ISEP - DM – Engenharia Informática[11]. This work documented the application of 4SRS-MSLA for an internal project at i2S company, allowing depicting how the existing monolith was decomposed in an MSA, and additionally discussing deployment and infrastructure needs for that MSA, providing complimentary insights for **Contribution 2: Incremental model refinement until service level.**

A crucial part of the research work relates to communicating the results. For that aim, several research papers were published, which are now listed.

Conference papers:

- Ferreira, N., Santos, N. & Machado, R.J., 2014. Modularization of Logical Software Architectures for Implementation with Multiple Teams. In Proceedings of the 14th International Conference on Computational Science and Its Applications. IEEE, pp. 1–11. DOI: 10.1109/ICCSA.2014.14
- Costa, N., Santos, N., Ferreira, N., & Machado, R. J., 2014. Delivering user stories for implementing logical software architectures by multiple scrum teams. In Computational Science and Its Applications. Springer International Publishing, pp. 747-762. DOI: 10.1007/978-3-319-09150-1_55

[11] Available at: http://hdl.handle.net/10400.22/11920

- Santos, N., Barbosa, D., Maia, P., Fernandes, F., Rebelo, M., Silva, P. V., Fernandes, J. M., Machado, R. J. (2016). iFloW: an integrated logistics software system for inbound supply chain traceability. In E. Mendonça, J. P., Fensterbank, S.-A., Barthet (Ed.), "Enterprise Interoperability VII". Springer, Cham. DOI: 10.1007/978-3-319-30957-6_15
- Santos, N., Fernandes, J. M., Carvalho, S. M., Silva, P. V., Fernandes, F., Rebelo, M., Fernandes, J. M., Machado, R. J. (2016). Using Scrum together with UML models: A collaborative University-Industry R&D software project. In Gervasi, O., Murgante, B., Misra, S., Rocha, A.M.A.C., Torre, C.M., Taniar, D., Apduhan, B.O., Stankova, E., Wang, S. (Eds.) Computational Science and Its Applications – Part III. Lecture Notes in Computer Science. Springer. DOI: 10.1007/978-3-319-42089-9_34
- Santos, N., Pereira, J., Morais, F., Barros, J., Ferreira, N., & Machado, R. J. (2018). An agile modeling oriented process for logical architecture design. In Gulden, J., Reinhartz-Berger, I., Schmidt, R., Guerreiro, S., Guédria, W., Bera, P. (Eds.) Enterprise, Business-Process and Information Systems Modeling. Lecture Notes in Computer Science. Springer, Cham. DOI: 10.1007/978-3-319-91704-7_17
- Santos, N., Pereira, J., Morais, F., Barros, J., Ferreira, N., & Machado, R. J. (2018). An experience report on using architectural models within distributed Scrum teams contexts. In *XP'18 Scientific Workshops*. ACM. DOI: 10.1145/3234152.3234180
- Santos, N., Pereira, J., Morais, F., Barros, J., Ferreira, N., & Machado, R. J. (2018). Incremental architectural requirements for agile modeling: a case study within a Scrum project. In *XP'18 Scientific Workshops*. ACM. DOI: 10.1145/3234152.3234166
- Santos, N., Pereira, J., Morais, F., Barros, J., Ferreira, N., & Machado, R. J. (2018). Deriving user stories for distributed Scrum teams from iterative refinement of architectural models. In *XP'18 Scientific Workshops*. ACM. DOI: 10.1145/3234152.3234165
- Santos, N., Rodrigues, H., Pereira, J., Morais, F., Abreu, R., Fernandes, N., Martins, D., Machado, R. J. (2018). UH4SP: a software platform for integrated management of connected smart plants. In: 9th IEEE International Conference on Intelligent Systems (IS). IEEE. DOI: 10.1109/IS.2018.8710468
- Santos, N., Pereira, J., Ferreira, N., & Machado, R. J. (2018). Modeling in agile software development: decomposing use cases towards logical architecture design. In *Product-Focused Software Process Improvement*. Lecture Notes in Computer Science. Springer. DOI: 10.1007/978-3-030-03673-7_31

- Santos, N., Ferreira, N., & Machado, R. J. (2019): “Towards agile architecting: Proposing an architectural pathway within an Industry 4.0 project”, Information Systems: Research, Development, Applications, Education. Springer, Cham. DOI: 10.1007/978-3-030-29608-7_10
- Santos, N., Rodrigues, H., Ferreira, N., & Machado, R.J. (2019). “Inputs from a Model-based Approach towards the Specification of Microservices Logical Architectures: an Experience Report”. Product-Focused Software Process Improvement. Springer, Cham.
- Santos, N., Salgado, C. E., Rodrigues, H., Morais, F., Melo, M,, Silva, S., Martins, R., Pereira, M., Ferreira, N., Pereira, M. & Machado, R. J. (2019). A logical architecture design method for microservices architectures. Proceedings of 13th European Conference on Software Architecture (ECSA 2019) - Vol.2. ACM. DOI: 10.1145/3344948.3344991

Book chapters

- Santos, N., Rodrigues, H., Pereira, J., Morais, F., Martins, R., Ferreira, N., Abreu, R., Machado, R.J. (2018): Specifying Software Services for Fog Computing Architectures Using Recursive Model Transformations. In: Mahmood, Z. (ed.) Fog Computing: Concepts, Frameworks and Technologies. pp. 153–181. Springer, Cham. DOI: 10.1007/978-3-319-94890-4
- Santos, N., Morais, F., Rodrigues, H. & Machado, R.J. (2019). Systems Development for the Industrial IoT: Challenges from Industry R&D Projects. In: Mahmood, Z. (Ed.), The Internet of Things in the Industrial Sector, 1st ed. Springer Cham. DOI: 10.1007/978-3-030-24892-5

Doctoral Consortium papers:

- Santos, N., Fernandes, J. M., Carvalho, S. M., Silva, P. V., Fernandes, F., Rebelo, M., Fernandes, J. M., Machado, R. J. (2016). Industrial interoperability issues when adopting Scrum in research projects: the case of the iFloW system. In “Enterprise Interoperability”, International Conference on Interoperability for Enterprise Systems and Applications (I-ESA) Doctoral Symposium.
- Santos, N., Machado, R. J., Ferreira, N. (2016). Adopting Logical Architectures within Agile Projects. In 6th Portuguese Software Engineering Doctoral Symposium (SEDES) in conjunction with the 10th International Conference on the Quality of Information and Communications Technology (QUATIC’16). DOI: 10.1109/QUATIC.2016.059

- Santos, N., Machado, R. J., Ferreira, N. (2017). Adopting Logical Architectures within Agile Projects. In *Doctoral Consortium of the European Conference in Information Systems* (ECIS'17 DC).

Involved Master thesis:

- MSc #1: Martins, Raquel "Designing Architectures for Industrial Cloud Solutions: The UH4SP Demonstration Case" (free translation of "Conceção de Arquiteturas de Soluções Cloud para a Indústria: Caso de Demonstração UH4SP"), Integrated Masters in Engineering and Management of Information Systems (MIEGSI), Department of Information Systems, University of Minho, Portugal, 2018 (http://repositorium.sdum.uminho.pt/handle/1822/59343)
- MSc #2: Correia, Luis "Development of an Agile metrics Framework" (free translation of "Desenvolvimento de um framework de métricas para projetos ágeis"), Integrated Masters in Engineering and Management of Information Systems (MIEGSI), Department of Information Systems, University of Minho, Portugal, 2018 (https://repositorium.sdum.uminho.pt/handle/1822/59138)
- MSc #3: Amaral, José Diogo Coelho, "The evolution of monohlithic architectures to microservice-based architectures" (free translation of "A evolução das arquiteturas monolíticas para as arquiteturas baseadas em microserviços"), ISEP - DM – Engenharia Informática, 2018. http://hdl.handle.net/10400.22/11920

Presentations at scientific conferences:

- "Modularization of Logical Software Architectures for Implementation with Multiple Teams", "Tools and Techniques in Software Development Processes" session at the 14th International Conference on Computational Science and Its Applications. Guimarães, Portugal, July 1st 2014
- "Delivering user stories for implementing logical software architectures by multiple scrum teams", "Workshop of Agile Software Development Techniques" session at the 14th International Conference on Computational Science and Its Applications. Guimarães, Portugal, July 2nd 2014.

- “Industrial interoperability issues when adopting Scrum in research projects: the case of the iFloW system”, at the International Conference on Interoperability for Enterprise Systems and Applications (I-ESA) Doctoral Symposium, Guimarães, Portugal a March 28th 2016
- “iFloW: an integrated logistics software system for inbound supply chain traceability”, 8th International Conference on Interoperability for Enterprise Systems and Applications (I-ESA), Guimarães, Portugal, March 31st 2016.
- “Adopting Logical Architectures within Agile Projects”, Doctoral Consortium of the European Conference in Information Systems (ECIS DC), Guimarães, Portugal, June 5th 2017
- “An agile modeling oriented process for logical architecture design.”, *23rd International Conference on Exploring Modeling Methods for Systems Analysis and development (EMMSAD) in conjunction with 30th International Conference on Advanced Information Systems (CAiSE)*, Tallin, Estonia, June 11th 2018.
- “An experience report on using architectural models within distributed Scrum teams contexts”, *6th International Workshop on Large-Scale Agile Development (LargeScaleAgile’18) in conjunction with the 19th International Conference on Agile Software Development (XP’18),* Porto, Portugal, May 28th 2018.
- “Incremental architectural requirements for agile modeling: a case study within a Scrum project” and “Deriving user stories for distributed Scrum teams from iterative refinement of architectural models”, at Poster Session and at “Poster Madness” session of International Conference on Agile Software Development (XP’18), Porto, Portugal, May 22nd to 24th 2018.

- “A logical architecture design method for microservices architectures”. In 3rd Workshop on Formal Approaches for Advanced Computing Systems (FAACS’19) in conjunction with the 13th European Conference on Software Architecture (ECSA 2019), Paris, France, September 10th 2019.
- “Towards agile architecting: Proposing an architectural pathway within an Industry 4.0 project”, at 12th EuroSymposium on Systems Analysis and Design, Gdansk, Poland, September 19th 2019.
- “Inputs from a Model-based Approach towards the Specification of Microservices Logical Architectures: an Experience Report”. International Conference on Product-Focused Software Process Improvement (PROFES) 2019. Barcelona, Spain, November 27th to 29th 2019.

Presentations at other events:

- Poster session, presenting preliminar research results, at "Doctoral Programme in Information Systems and Technology (PDTSI) Autumn Symposium 2016", Guimarães, Portugal, November 16th 2016
- Presentation of AMPLA process and the UH4SP architecture, included in the project's results, at the event "Solutions for Smart Factories in the Era of Industry 4.0 – OpenDay by Cachapuz, Braga's Municipality Economic Week", Braga, Portugal, May 24th 2018
- Presentation of the IMP_4.0 architecture, included in the project's results, at the event "Industry 4.0 – Presentation of the IMP_4.0 Platform", Braga, Portugal, November 27th 2017
- Presentation of the ASD approach adopted by the iFloW project, included in the project's results, at "HMIExcel Program Closing Conference", Braga, Portugal, June 30th 2015

Table 44 summarizes the contribution and content of the published scientific outputs in relation with the project/demonstration case the work was validated as well as the research contribution.

Table 44. Published paper's relation with demonstration case and research contribution

	ISOFIN	iFloW	UH4SP	IMP_4.0	ISMPM
TTSDP'14	RC2				
WAGILE'14	RC2				
I-ESA DS		RC1			
I-ESA'16		RC1			
SEPA'16		RC1			
SEDES'16		RC1	RC1		
ECIS DC		RC1	RC1		
XP'18 WS			RC2		
EMMSAD'18			RC1/3		
Fog book			RC3		
MSc #1			RC1/3		
QuASD'18			RC1		
IS'18			RC3		
MSc #2			RC2		
MSc #3					RC3
IIoT book			RC3	RC3	
FAACS'19				RC3	
Eurosymposium'19			RC1		
PROFES'19			RC3		

7.4. Future work

This thesis addressed topics (cf. Section 7.1) such as agile modeling, agile requirements engineering, agile logical architecting, Change-impact analysis, microservices design, (agile) backlog items, and Agile multi-teams management. In addition, in all of them, there are opportunities for improvement in future work.

Agile modeling

AMPLA proposes additional activities and artifacts in agile requirements engineering methods. AMPLA has three established phases: (i) Requirements Elicitation; (ii) Requirements Analysis & Modeling; (iii) Architecture Design; and (iv) Delivery Cycles. It is also composed by milestones and artifacts - like the several derived models (sequence, use cases, components). Because the process prescribes models derivation sequencing and dependencies, it may be perceived as some threats for the process agility[12]. We acknowledge that these dependencies are mandatory as it is, but it is planned to use AMPLA in future R&D projects at the CCG\ZGDV Institute and depict how AMPLA process' flexibility may be increased.

The initial three phases (Requirements Elicitation, Requirements Analysis & Modeling and Architecture Design) required having dedicated teams for conducting these phases, with more focus in modeling. Only when entering the fourth and final phase, the Delivery Cycles, project teams were included in the process. In projects with more than one team, the dedicated team for conducting the first three phases was composed by representatives from the involved teams, like the cases of ISOFIN and UH4SP. However, in settings of more than one team, each team may start defining requirements from the beginning, rather than start by a "shared" model, like our proposed candidate architecture. There is a need for future research in a proper discussion on the suitability of parallel requirements emerging rather than starting with a single architecture proposal.

Another issue is that AMPLA was only applied in settings where the method's designers were included in the project's teams. For that reason, there is a lack of information regarding the project teams' ability to adopt AMPLA. It is also planned for future work to apply AMPLA in other projects at the CCG\ZGDV Institute without including the method's designers in the project team. This work intends to analyze if team members are able to interpret the process rules, measure the learning curve, the willingness of teams to model all artefacts, among others.

[12] Such claim is based on collected feedback, for instance, in presentations at conferences or in paper reviews.

The process lacks still of a formalization in order to ease its adoption without including the method's designers. Hence, as future work it is planned a formalization in a process-oriented notation, like Software Process Engineering Metamodel (SPEM) (OMG, 2008), but mainly Essence (Jacobson et al., 2013). Essence is a modular, method-agnostic progress control tool for software engineering endeavors. Essence, or the SEMAT's Essence Theory of Software Engineering, consists of a kernel and a language. The kernel contains all the elements present in every software engineering endeavor, while the language can be used to extend the kernel to be tailored for specific contexts. The alphas of the kernel serve as a way of tracking project health. Alpha states offer a way of tracking progress. In the future, AMPLA's model derivation and its composing practices will be formalized using Essence's Kernel alphas, activity spaces, and competencies. The formalization includes describing AMPLA's modeling practices, how performing in parallel with other agile framework or practice, and how progressing may be tracked.

Agile requirements engineering (RE)

The MVP requirements elicitation may be performed by gathering stakeholder's individual needs, also referred as "bespoke", "custom" or "tailor-made" RE (Fernandes & Machado, 2016). Alternatively, it may be oriented to a combination of a number of known customers, or to a mass-market where customers cannot be clearly pinpointed, also referred as "market-driven RE" (MDRE) (Regnell & Brinkkemper, 2005).

DUARTE is oriented for requirements elicitation together with a set of key stakeholders (e.g., business owner, product lead, etc.) that have clearly defined their business needs, based on a previously defined market strategy. Hence, this phase is clearly based in Bespoke RE. Alternatively, elicitation could be based in involving customers or other market entities (MDRE). This context will be object of future research.

Agile logical architecting

Agile architecting is characterized for performing design activities in a way that architecture and its requirements emerge throughout software development, where BDUF is avoided because needs change and many features specified in BDUF are afterwards classified as YAGNI.

The AAL described in Section 5.2 showed that architecture evolves throughout the SDLC and supports the viewpoints during such transition. However, there is space for approaches that automate such transitions. This research presented logical architectures for the design phase and deployment architectures for the delivery phase, like UH4SP and ISMPM. Although one may depict relationships with both models and its inherent viewpoint transition, such transition was not performed in a systematic and automatized way. For that reason, we acknowledge that transition from design to technical and deployment (in Delivery stage) needs addressing in future research for of a proper transition support.

The research works presented in Chapter 2 included architecture-centric approaches present in ASD (Bellomo, Kruchten, Nord, & Ozkaya, 2014; Farhan, Tauseef, & Fahiem, 2009; Jeon, Han, Lee, & Lee, 2011; Madison, 2010; R. L. Nord & Tomayko, 2006; R. Nord, Ozkaya, & Kruchten, 2014), where architecture-centric methods such as Quality Attribute Workshop (QAW), Attribute-Driven Design (ADD), Architecture Trade-off Analysis Method (ATAM) / Cost-Benefit Analysis Method (CBAM) and Active Review for Intermediate Designs (ARID) are performed in parallel with the development iterations. These approaches are based in quality requirements. AMPLA, in the other hand, is based in functional requirements. Although not explored in this thesis, AMPLA (more concretely, the 4SRS) is able to co-exist with these approaches – in fact, co-existing the 4SRS with any of these approaches would certainly strengthen AMPLA.

Change-impact analysis (CIA)

Included within the AMPLA method, this thesis presented how to support CIA practices for adjustments proposals to the product backlog, enabled by the model traceability. This thesis proposed traceability paths for performing CIA practices that focuses in architecturally significant requirements (ASR), quality characteristics, business and customer value of the requirement, affected components, standards compliance, requirements emerge, and architectural debt.

The method was demonstrated by the applicability of the CIA practices within a software team. This research focused in the use of models to support decision-making for business and for architects, pointing where software is affected by changes and projecting the new behavior after addressing those changes. However, the scope was not to provide guidelines or to promote some metrics for an effective decision-making. In future research, we plan in properly analyzing model information for providing inputs in such decision-making.

Microservices design

The traceability associated to 4SRS-MSLA method assures an alignment between the initial Use Case model and the derived architecture proposed solution. Additionally, the method's outputs stand as inputs for several SoaML diagrams, which are complimentary for a proper specification of microservices behavior and associations, such as Service Participants and Contracts. The 4SRS-MSLA steps were adapted to meet widely known microservices characteristics. Remaining diagrams such as Service Interface, Capabilities, Service Data, Service Architecture, and Service Contracts, among others, are to be discussed in future works.

Moreover, the increasing adoption of microservices in industry led to defining other patterns adopted when MSAs begin to scale, as the ones related to communications, database architectures, data consistency, security, deployment, among many others. The discussion from this research is an initial effort in designing the microservices architecture. It allowed defining the bounded contexts, separation of data models, needs for API calls. However, many issues around these concerns need to be addressed in microservices development but will be focused in future research, like data consistency, security (tokens) needs, or messaging, brokerage or API management. Although these have direct implications in the logical architecture derivation, and hence relating to the present work, future work will address detailing the database-per-service definition from the 4SRS-MSLA.

From models to (agile) backlog items

In this thesis is presented an approach for modeling in UML a set of "just-enough" requirements and a candidate architecture, that afterwards originate ASD-oriented backlog items. The complexity is addressed by scaling the development, namely by distributed Scrum teams. The candidate architecture is designed by using the 4SRS method for deriving a logical architecture, which is then modularized, refined, and used as input for a set of transformation rules for backlog structure.

By applying the transformation rules, the team backlog was composed by themes, epics, use cases, user stories. In future research we plan in defining additional inputs to be included as transformation rules, like to foresee inclusion of backlog items that define the need for technical work tasks, knowledge acquisition tasks, prototyping, architectural spikes and development spikes.

Agile multi-teams management

AMPLA was used in a project where a dedicated team was responsible for performing requirements modeling (by applying DUARTE) and the candidate architecture design (by performing the 4SRS), modularize the architecture and assign those modules. Then, each team was responsible to deliver the software and manage coordination efforts.

When each team starts defining requirements from the beginning (as already referred in future work for "agile modeling"), it is required to propose new theories in inter-team cooperation, like architecture co-design, for instance. As part of the future work for "agile modeling", it is also planned to include the development of inter-teams communication and coordination in such settings.

For project management, AMPLA provides the traceability mechanisms for linking the user stories and features to the expectations, which supported the control of the project progress when at a delivery stage. At this stage, whenever an iteration occurs (e.g., Sprint), one of the main concerns in controlling progress is by measuring the value delivered to customers. One approach that has been increasingly adopted is the "Objectives and Key Results" (OKR) (Doerr, 2018), where expectations describe the key results and how they are achieved. AMPLA allowed controlling the evolution of value delivered throughout the Sprints, namely by controlling the cumulative value (in %) of the project objective to be achieved. Namely, OKR's were being met, as soon as the objective's value delivered was 100% 'done'. OKR is still an immature approach and, for that reason, we plan to continue to use it in future projects but it is predictable that its use within AMPLA will change.

References

Ambler, S., & Lines. (2019). *Choose Your WoW!: A Disciplined Agile Delivery Handbook for Optimizing Your Way of Working (WoW)*. Disciplined Agile.

Bellomo, S., Kruchten, P., Nord, R., & Ozkaya, I. (2014). How to Agilely Architect an Agile Architecture. *Cutter IT Journal*.

Doerr, J. (2018). *Measure What Matters: How Google, Bono, and the Gates Foundation Rock the World with OKRs*. Portfolio/Penguin.

Farhan, S., Tauseef, H., & Fahiem, M. A. (2009). Adding agility to architecture tradeoff analysis method for mapping on crystal. In *WRI World Congress on Software Engineering (WCSE'09) - Volume 04* (Vol. 4, pp. 121–125). IEEE. https://doi.org/10.1109/WCSE.2009.405

Fernandes, J. M., & Machado, R. J. (2016). *Requirements in Engineering Projects*. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-18597-2

Jacobson, I., Ng, P., McMahon, P., Spence, I., & Lidman, S. (2013). *The essence of software Engineering: applying the SEMAT kernel*. Addison-Wesley.

Jeon, S., Han, M., Lee, E., & Lee, K. (2011). Quality attribute driven agile development. In *9th International Conference on Software Engineering Research, Management and Applications (SERA)* (pp. 203–210). IEEE. https://doi.org/10.1109/SERA.2011.24

Leffingwell, D. (2010). *Agile software requirements: lean requirements practices for teams, programs, and the enterprise*. Addison Wesley Longman.

Madison, J. (2010). Agile architecture interactions. *IEEE Software*, *27*(2), 41–48. https://doi.org/10.1109/MS.2010.35

Nord, R. L., & Tomayko, J. E. (2006). Software architecture-centric methods and agile development. *IEEE Software*, *23*(2), 47–53. https://doi.org/10.1109/MS.2006.54

Nord, R., Ozkaya, I., & Kruchten, P. (2014). Agile in distress: architecture to the rescue. In T. Dingsøyr & N. B. Moe (Eds.), *International Conference on Agile Software Development (XP'14)* (pp. 43–57). Springer Verlag. https://doi.org/10.1007/978-3-319-14358-3_5

OMG. (2008). Software and Systems Process Engineering Meta-Model (SPEM).

Regnell, B., & Brinkkemper, S. (2005). Market-Driven Requirements Engineering for Software Products. In *Engineering and Managing Software Requirements* (pp. 287–308). Berlin/Heidelberg: Springer-Verlag. https://doi.org/10.1007/3-540-28244-0_13

APPENDIXES

Appendix A – iFloW models

Use Case model

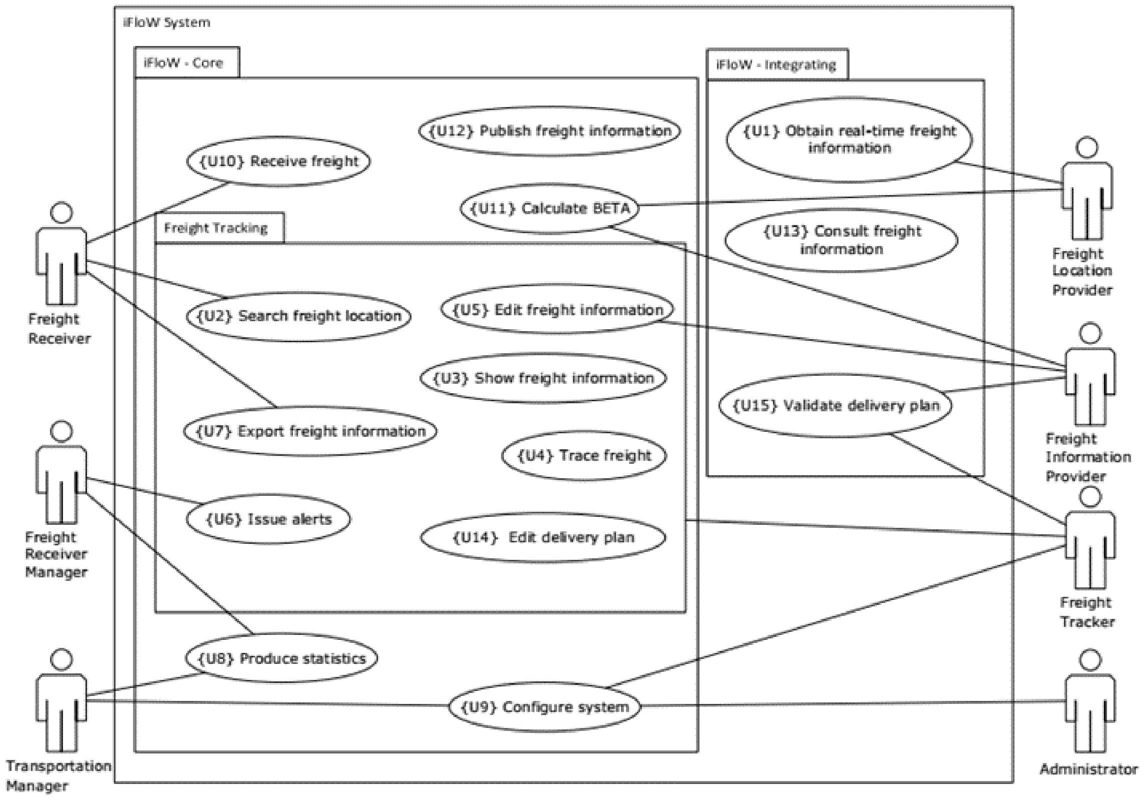

iFloW System
iFloW - Core
iFloW - Integrating
Freight Tracking
{U1} Obtain real-time freight information
{U2} Search freight location
{U3} Show freight information
{U4} Trace freight
{U5} Edit freight information
{U6} Issue alerts
{U7} Export freight information
{U8} Produce statistics
{U9} Configure system
{U10} Receive freight
{U11} Calculate BETA
{U12} Publish freight information
{U13} Consult freight information
{U14} Edit delivery plan
{U15} Validate delivery plan
Freight Receiver
Freight Receiver Manager
Transportation Manager
Freight Location Provider
Freight Information Provider
Freight Tracker
Administrator

Figure 147. iFloW use case model

Logical Architecture

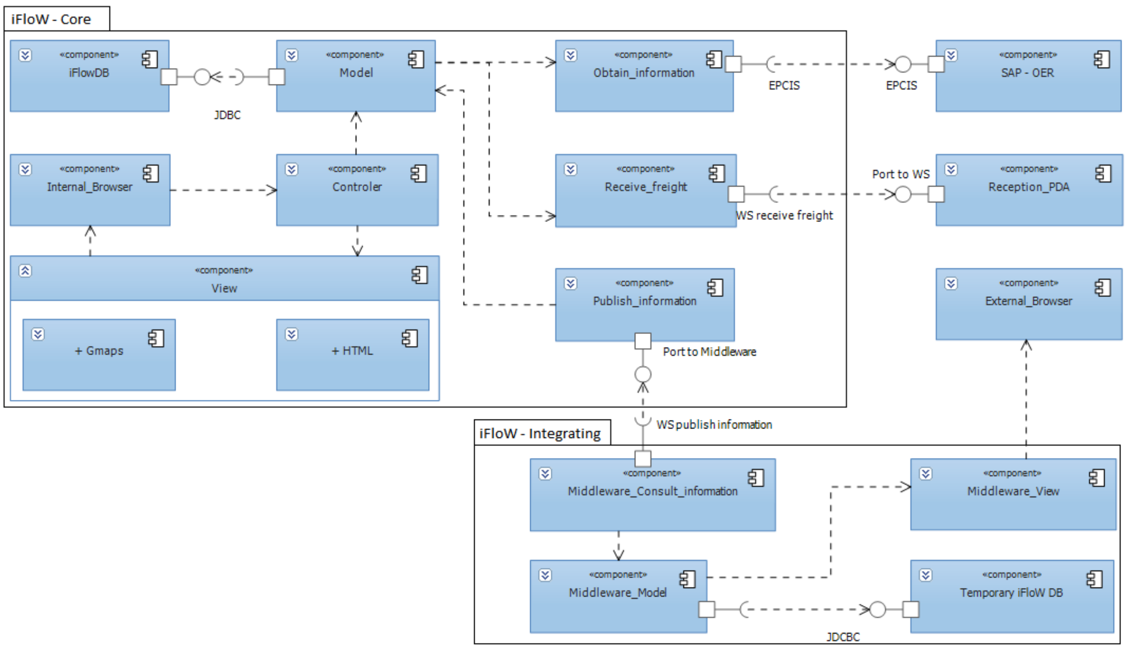

iFloW - Core
«component» iFlowDB
«component» Model
«component» Obtain_information
«component» SAP - OER
JDBC
EPCIS
EPCIS
«component» Internal_Browser
«component» Controler
«component» Receive_freight
Port to WS
«component» Reception_PDA
WS receive freight
«component» View
+ Gmaps
+ HTML
«component» Publish_information
«component» External_Browser
Port to Middleware
WS publish information
iFloW - Integrating
«component» Middleware_Consult_information
«component» Middleware_View
«component» Middleware_Model
«component» Temporary iFloW DB
JDCBC

Figure 148. iFloW architecture

Product Backlog

Table 45 - iFloW Backlog list

Task ID	Task Description	Bosch Priority
{U1.1.1}	Obtain freight information from Kuehne Nagel related to transit from Algeciras Consolidation Center	1
{U1.1.2}	Obtain freight information from Kuehne Nagel related to transit from Hong-Kong Consolidation Center	2
{U3.1}	Show freight general information	3
{U10}	Receive Freight	4
{U5.5}	Edit freight ETA	5
{U6.1}	Issue alerts for ETA changes	6
{U2}	Search freight location	9
{U7.1}	Export freight information to SAP	10
{U6.2}	Issue alerts for updates in freight quantities	11
{U4}	Trace freight	12
{U7.2}	Export freight information to EWL	13
{U5.7}	Cancel freight tracking	14
{U12}	Publish freight information	15
{U13}	Consult freight information	16
{U9.5}	Configure delivery plan	17
{U14}	Edit delivery plan	18
{U15}	Validate delivery plan	19
{U1.1.3}	Obtain freight information from Kuehne Nagel related to transit from Singapore Consolidation Center	20
{U1.1.4.2}	Obtain real-time freight location from Kuehne Nagel related to transit from Algeciras Consolidation Center	21

{U1.1.4}	Obtain real-time freight location from Kuehne Nagel related to transit from Singapore Consolidation Center	22
{U1.1.5}	Obtain freight information from Kuehne Nagel related to transit from Penang Consolidation Center	23
{U1.1.6}	Obtain real-time freight location from Kuehne Nagel related to transit from Penang Consolidation Center	24
{U1.1.7}	Obtain freight information from Kuehne Nagel related to transit from Shangai Consolidation Center	25
{U1.1.8}	Obtain real-time freight location from Kuehne Nagel related to transit from Shangai Consolidation Center	26
{U1.1.9}	Obtain freight information from Kuehne Nagel related to transit from Taiwan Consolidation Center	27
{U1.1.10}	Obtain real-time freight location from Kuehne Nagel related to transit from Taiwan Consolidation Center	28
{U1.1.11}	Obtain freight information from Kuehne Nagel related to transit from Bangkok Consolidation Center	29
{U1.1.12}	Obtain real-time freight location from Kuehne Nagel related to transit from Bangkok Consolidation Center	30
{U1.1.13}	Obtain freight information from NNR Global Logistics related to transit before Hong-Kong Consolidation Center	31
{U1.1.14}	Obtain real-time freight location from NNR Global Logistics related to transit before Hong-Kong Consolidation Center	32
{U9.3}	Configure users	33
{U9.1}	Configure routes	34
{U9.2}	Configure raw materials	35
{U9.4.2}	Configure interface layout	36
{U9.4.1}	Configure alerts	37
{U5.1}	Tag freight	38
{U8.2.6}	Produce statistics of special freights requests	39
{U8.2.1}	Produce OTD statistics per forwarder	A
{U5.3}	Indicate alternative special freight	A
{U8.4}	Export statistics	A

{U11}	Calculate BETA (ETA obtained by alghoritm calculation)	A
{U5.6}	Add comments	A
{U6.3}	Issue alerts for changes in transportation	B
{U5.4}	Indicate freight volume changes	B
{U7.3}	Export freight information to *"Ficheiro de descargas"*	B
{U1.2.2}	Obtain real-time freight location from UPS related to transit before Consolidation Centre	C
{U1.2.3}	Obtain real-time freight location from TNT related to transit before Consolidation Centre	C
{U1.2.4}	Obtain real-time freight location from Schenker related to transit before Consolidation Centre	C
{U1.2.5}	Obtain real-time freight location from Schenker related to transit after Consolidation Centre	C
{U1.2.6}	Obtain real-time freight location from LUSOCARGO related to transit after Consolidation Centre	C
{U1.2.7}	Obtain real-time freight location from FEDEX related to transit before Consolidation Centre	C
{U1.2.8}	Obtain real-time freight location from DHL related to transit before Consolidation Centre	C
{U1.1.15}	Obtain freight information from UPS related to transit before Schweinfurt Consolidation Centre	C
{U1.1.16}	Obtain freight information from TNT related to transit before Schweinfurt Consolidation Centre	C
{U1.1.17}	Obtain freight information from Schenker related to transit before Schweinfurt Consolidation Centre	C
{U1.1.18}	Obtain freight information from Schenker related to transit after Schweinfurt Consolidation Centre	C
{U1.1.19}	Obtain freight information from LUSOCARGO related to transit after Schweinfurt Consolidation Centre	C
{U1.1.20}	Obtain freight information from FEDEX related to transit before Schweinfurt Consolidation Centre	C
{U1.1.21}	Obtain freight information from DHL related to transit before Schweinfurt Consolidation Centre	C
{U1.2.9}	Obtain real-time freight location from Vanquish China Ltd related to transit before Hong-Kong Consolidation Center	D
{U1.2.10}	Obtain real-time freight location from TLP related to transit before Hong-Kong Consolidation Center	D
{U1.2.11}	Obtain real-time freight location from Supplier's own transport related to transit before Hong-Kong Consolidation Center	D
{U1.2.12}	Obtain real-time freight location from NNR Global Logistics related to transit before Singapure Consolidation Center	D

{U1.2.13}	Obtain real-time freight location from Man Tak related to transit before Hong-Kong Consolidation Center	D
{U1.2.14}	Obtain real-time freight location from Kamfu Logistics related to transit before Hong-Kong Consolidation Center	D
{U1.2.15}	Obtain real-time freight location from Hotline International Transport (H.K.) LTD related to transit before Hong-Kong Consolidation Center	D
{U1.2.16}	Obtain real-time freight location from Hankyu Hanshin Express related to transit before Penang Consolidation Center	D
{U1.2.17}	Obtain real-time freight location from Good Start Transportation Company related to transit before Hong-Kong Consolidation Center	D
{U1.2.18}	Obtain real-time freight location from CWM related to transit before Penang Consolidation Center	D
{U1.2.19}	Obtain real-time freight location from CWB related to transit before Shangai Consolidation Center	D
{U1.2.20}	Obtain real-time freight location from AEO Logistics related to transit before Penang Consolidation Center	D
{U1.1.22}	Obtain freight information from Vanquish China Ltd related to transit before Hong-Kong Consolidation Center	D
{U1.1.23}	Obtain freight information from TLP related to transit before Hong-Kong Consolidation Center	D
{U1.1.24}	Obtain freight information from Supplier’s own transport related to transit before Hong-Kong Consolidation Center	D
{U1.1.25}	Obtain freight information from NNR Global Logistics related to transit before Singapure Consolidation Center	D
{U1.1.26}	Obtain freight information from Man Tak related to transit before Hong-Kong Consolidation Center	D
{U1.1.27}	Obtain freight information from Kamfu Logistics related to transit before Hong-Kong Consolidation Center	D
{U1.1.28}	Obtain freight information from Hotline International Transport (H.K.) LTD related to transit before Hong-Kong Consolidation Center	D
{U1.1.29}	Obtain freight information from Hankyu Hanshin Express related to transit before Penang Consolidation Center	D
{U1.1.30}	Obtain freight information from Good Start Transportation Company related to transit before Hong-Kong Consolidation Center	D
{U1.1.31}	Obtain freight information from CWM related to transit before Penang Consolidation Center	D
{U1.1.32}	Obtain freight information from CWB related to transit before Shangai Consolidation Center	D
{U1.1.33}	Obtain freight information from AEO Logistics related to transit before Penang Consolidation Center	D

{U3.2}	Show Invoice Document	E
{U8.3.2}	Produce transportation costs statistics	E
{U8.2.5}	Produce statistics of ETA variation by forwarder	E
{U8.1.1}	Produce quantity delivered statistics per supplier	E
{U8.2.2}	Produce OTD statistics related to the freights	E
{U8.2.3}	Produce OTD forwarders’ ranking	E
{U8.3.1}	Produce occupancy rate statistics by container	E
{U8.1.2}	Produce frequency of delivery statistics per supplier	E
{U8.2.4}	Produce forwarders’ deliveries frequency ranking	E
{U5.2}	Issue internal urgent note	E

Appendix B - ISOFIN models

Logical Architecture

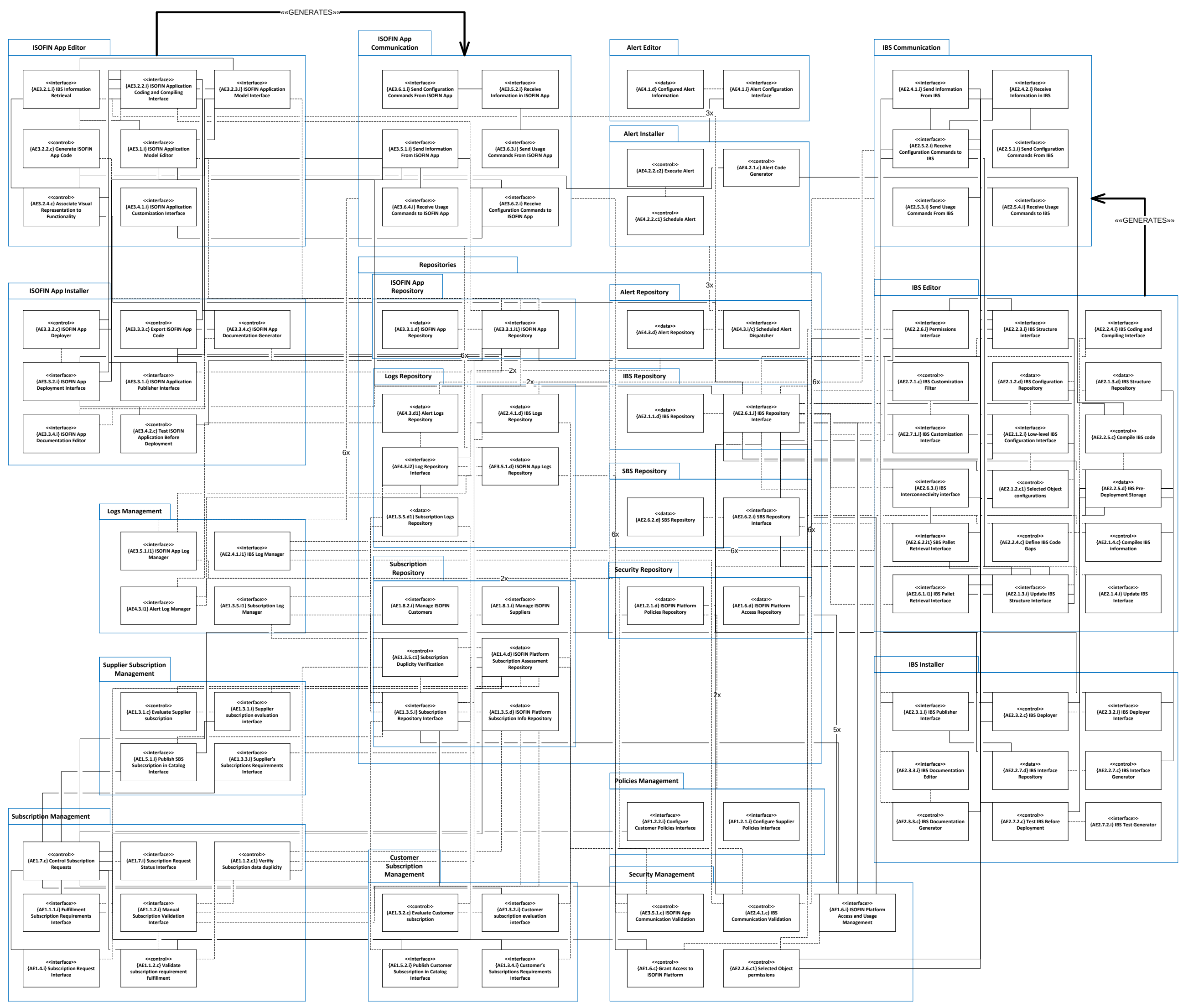

««GENERATES»»
ISOFIN App Editor
<<interface>> {AE3.2.1.i} IBS Information Retrieval
<<interface>> {AE3.2.2.i} ISOFIN Application Coding and Compiling Interface
<<interface>> {AE3.2.3.i} ISOFIN Application Model Interface
<<control>> {AE3.2.2.c} Generate ISOFIN App Code
<<interface>> {AE3.1.i} ISOFIN Application Model Editor
<<control>> {AE3.2.4.c} Associate Visual Representation to Functionality
<<interface>> {AE3.4.1.i} ISOFIN Application Customization Interface
ISOFIN App Communication
<<interface>> {AE3.6.1.i} Send Configuration Commands From ISOFIN App
<<interface>> {AE3.5.2.i} Receive Information in ISOFIN App
<<interface>> {AE3.5.1.i} Send Information From ISOFIN App
<<interface>> {AE3.6.3.i} Send Usage Commands From ISOFIN App
<<interface>> {AE3.6.4.i} Receive Usage Commands to ISOFIN App
<<interface>> {AE3.6.2.i} Receive Configuration Commands to ISOFIN App
Alert Editor
<<data>> {AE4.1.d} Configured Alert Information
<<interface>> {AE4.1.i} Alert Configuration Interface
3x
Alert Installer
<<control>> {AE4.2.2.c2} Execute Alert
<<control>> {AE4.2.1.c} Alert Code Generator
<<control>> {AE4.2.2.c1} Schedule Alert
IBS Communication
<<interface>> {AE2.4.1.i} Send Information From IBS
<<interface>> {AE2.4.2.i} Receive Information in IBS
<<interface>> {AE2.5.2.i} Receive Configuration Commands to IBS
<<interface>> {AE2.5.1.i} Send Configuration Commands From IBS
<<interface>> {AE2.5.3.i} Send Usage Commands From IBS
<<interface>> {AE2.5.4.i} Receive Usage Commands to IBS
««GENERATES»»
ISOFIN App Installer
<<control>> {AE3.3.2.c} ISOFIN App Deployer
<<control>> {AE3.3.3.c} Export ISOFIN App Code
<<control>> {AE3.3.4.c} ISOFIN App Documentation Generator
<<interface>> {AE3.3.2.i} ISOFIN App Deployment Interface
<<interface>> {AE3.3.1.i} ISOFIN Application Publisher Interface
<<interface>> {AE3.3.4.i} ISOFIN App Documentation Editor
<<control>> {AE3.4.2.c} Test ISOFIN Application Before Deployment
Repositories
ISOFIN App Repository
<<data>> {AE3.3.1.d} ISOFIN App Repository
<<interface>> {AE3.3.1.i1} ISOFIN App Repository
6x
2x
Logs Repository
<<data>> {AE4.3.d1} Alert Logs Repository
<<data>> {AE2.4.1.d} IBS Logs Repository
<<interface>> {AE4.3.i2} Log Repository Interface
<<data>> {AE3.5.1.d} ISOFIN App Logs Repository
<<data>> {AE1.3.5.d1} Subscription Logs Repository
Alert Repository
3x
<<data>> {AE4.3.d} Alert Repository
<<interface>> {AE4.3.i/c} Scheduled Alert Dispatcher
IBS Repository
<<data>> {AE2.1.1.d} IBS Repository
<<interface>> {AE2.6.1.i} IBS Repository Interface
SBS Repository
<<data>> {AE2.6.2.d} SBS Repository
<<interface>> {AE2.6.2.i} SBS Repository Interface
6x
Subscription Repository
<<interface>> {AE1.8.2.i} Manage ISOFIN Customers
<<interface>> {AE1.8.1.i} Manage ISOFIN Suppliers
<<control>> {AE1.3.5.c1} Subscription Duplicity Verification
<<data>> {AE1.4.d} ISOFIN Platform Subscription Assessment Repository
<<interface>> {AE1.3.5.i} Subscription Repository Interface
<<data>> {AE1.3.5.d} ISOFIN Platform Subscription Info Repository
Security Repository
<<data>> {AE1.2.1.d} ISOFIN Platform Policies Repository
<<data>> {AE1.6.d} ISOFIN Platform Access Repository
2x
5x
IBS Editor
<<interface>> {AE2.2.6.i} Permissions Interface
<<interface>> {AE2.2.3.i} IBS Structure interface
<<interface>> {AE2.2.4.i} IBS Coding and Compiling Interface
<<control>> {AE2.7.1.c} IBS Customization Filter
<<data>> {AE2.1.2.d} IBS Configuration Repository
<<data>> {AE2.1.3.d} IBS Structure Repository
<<interface>> {AE2.7.1.i} IBS Customization Interface
<<interface>> {AE2.1.2.i} Low-level IBS Configuration Interface
<<control>> {AE2.2.5.c} Compile IBS code
<<interface>> {AE2.6.3.i} IBS Interconnectivity interface
<<control>> {AE2.1.2.c1} Selected Object configurations
<<data>> {AE2.2.5.d} IBS Pre-Deployment Storage
<<interface>> {AE2.6.2.i1} SBS Pallet Retrieval Interface
<<control>> {AE2.2.4.c} Define IBS Code Gaps
<<control>> {AE2.1.4.c} Compiles IBS information
<<interface>> {AE2.6.1.i1} IBS Pallet Retrieval Interface
<<interface>> {AE2.1.3.i} Update IBS Structure Interface
<<interface>> {AE2.1.4.i} Update IBS Interface
IBS Installer
<<interface>> {AE2.3.1.i} IBS Publisher Interface
<<control>> {AE2.3.2.c} IBS Deployer
<<interface>> {AE2.3.2.i} IBS Deployer Interface
<<interface>> {AE2.3.3.i} IBS Documentation Editor
<<data>> {AE2.2.7.d} IBS Interface Repository
<<control>> {AE2.2.7.c} IBS Interface Generator
<<control>> {AE2.3.3.c} IBS Documentation Generator
<<control>> {AE2.7.2.c} Test IBS Before Deployment
<<interface>> {AE2.7.2.i} IBS Test Generator
Logs Management
<<interface>> {AE3.5.1.i1} ISOFIN App Log Manager
<<interface>> {AE2.4.1.i1} IBS Log Manager
<<interface>> {AE4.3.i1} Alert Log Manager
<<interface>> {AE1.3.5.i1} Subscription Log Manager
Supplier Subscription Management
<<control>> {AE1.3.1.c} Evaluate Supplier subscription
<<interface>> {AE1.3.1.i} Supplier subscription evaluation interface
<<interface>> {AE1.5.1.i} Publish SBS Subscsription in Catalog Interface
<<interface>> {AE1.3.3.i} Supplier's Subscriptions Requirements Interface
Subscription Management
<<control>> {AE1.7.c} Control Subscription Requests
<<interface>> {AE1.7.i} Suscription Request Status Interface
<<control>> {AE1.1.2.c1} Verifiy Subscription data duplicity
<<interface>> {AE1.1.1.i} Fulfillment Subscription Requirements Interface
<<interface>> {AE1.1.2.i} Manual Subscription Validation Interface
<<interface>> {AE1.4.i} Subscription Request Interface
<<control>> {AE1.1.2.c} Validate subscription requirement fulfillment
Customer Subscription Management
<<control>> {AE1.3.2.c} Evaluate Customer subscription
<<interface>> {AE1.3.2.i} Customer subscription evaluation interface
<<interface>> {AE1.5.2.i} Publish Customer Subscsription in Catalog Interface
<<interface>> {AE1.3.4.i} Customer's Subscriptions Requirements Interface
Policies Management
<<interface>> {AE1.2.2.i} Configure Customer Policies Interface
<<interface>> {AE1.2.1.i} Configure Supplier Policies Interface
Security Management
<<control>> {AE3.5.1.c} ISOFIN App Communication Validation
<<control>> {AE2.4.1.c} IBS Communication Validation
<<interface>> {AE1.6.i} ISOFIN Platform Access and Usage Management
<<control>> {AE1.6.c} Grant Access to ISOFIN Platform
<<control>> {AE2.2.6.c1} Selected Object permissions

Figure 149: ISOFIN Product-level Logical Architecture

Product Spots Overview

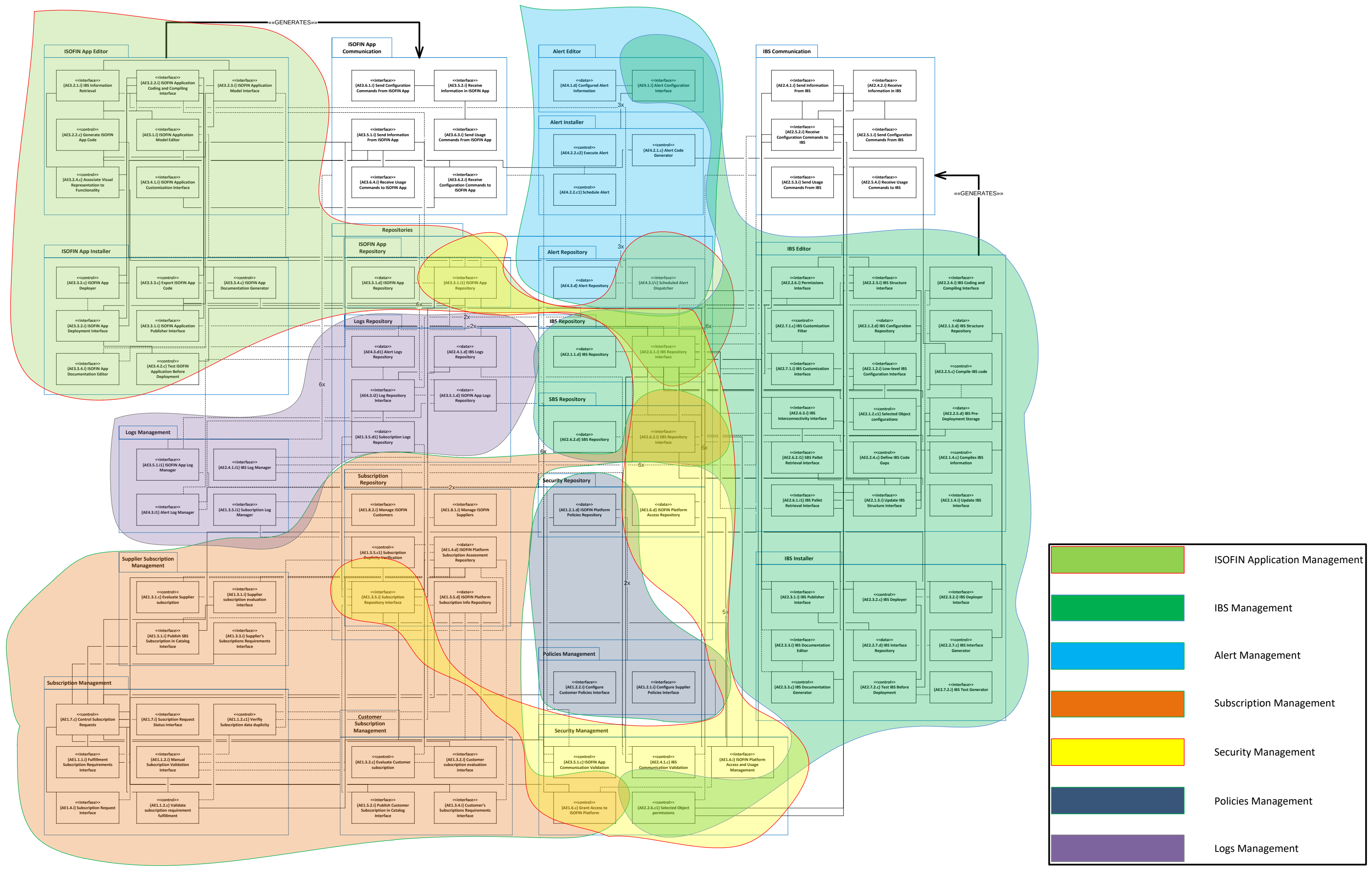


Figure 150: Logical Architecture Applications Spots

ISOFIN App module's User Stories

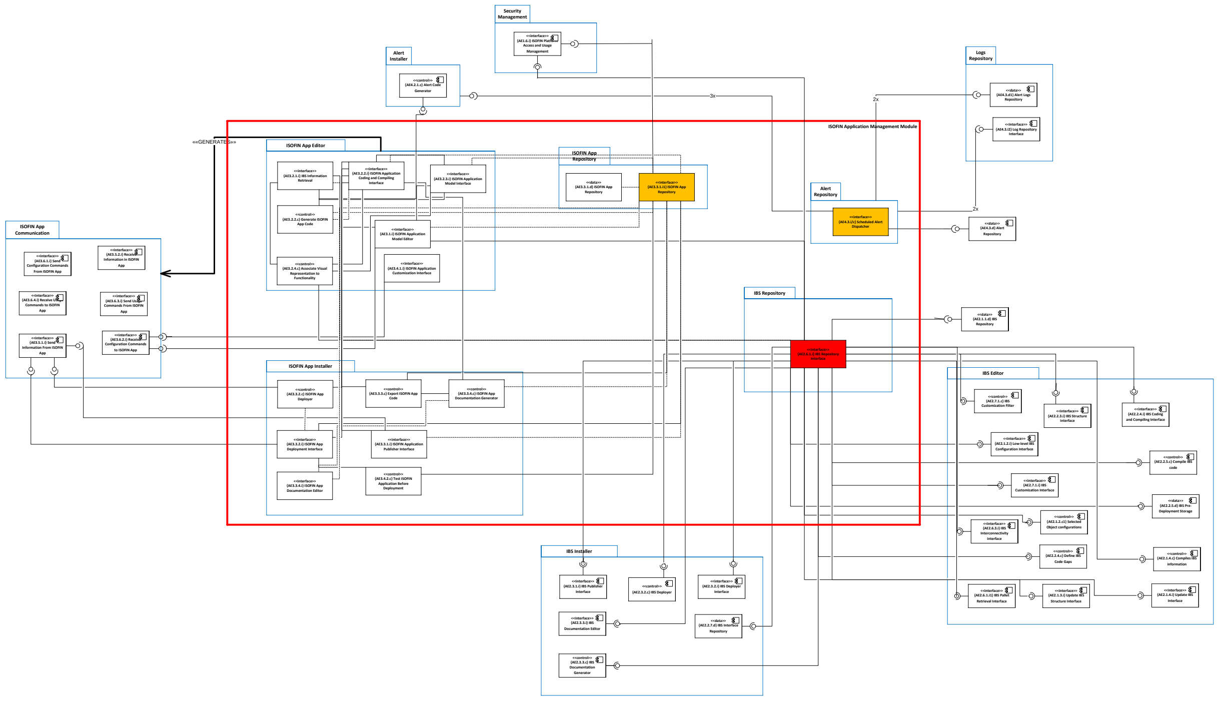


Figure 151. ISOFIN App module

Table 46. User Stories for c-types from ISOFIN App Management module

Component	**Name**	**As a(n) <actor>**	**I want/need (to/be able to) <description>**	**In order to <outcome>**
3.2.2.c	Generate ISOFIN App Code	IBS Developer	generate ISOFIN App code	generate ISOFIN App
3.2.4.c	Associate Visual Representation to Functionality	IBS Developer	associate visual representations to functionalities	define interface fields
3.3.2.c	ISOFIN App Deployer	IBS Developer	ISOFIN App deployer	execute ISOFIN App deployment
3.3.3.c	Export ISOFIN App Code	IBS Developer	export ISOFIN App code	deploy ISOFIN Application
3.3.4.c	ISOFIN App Documentation Generator	IBS Developer	ISOFIN App Documentation Generator	automatically generate ISOFIN app documentation

3.4.2.c	Test ISOFIN App Before Deployment	IBS Developer / Business User	test ISOFIN App Before Deployment	render and test ISOFIN App in PreRuntime

Table 47. User Stories for d-types from ISOFIN App Management module

Component	Name	As a(n) <actor>	I want/need (to/be able to) In order to	<description> <outcome>
3.3.1.d	ISOFIN App Repository	IBS Developer	ISOFIN App repository	publish ISOFIN application in catalog

Table 48. User Stories for i-types from ISOFIN App Management module

Components	Name	As a(n) <actor>	I want/need (to/be able to) <description>	In order to <outcome>
3.1.i	ISOFIN App Model Editor	IBS Business Analyst	ISOFIN App model editor interface	model the composition of an ISOFIN application
3.2.1.i	IBS Information Retrieval	IBS Developer	IBS information retrieval interface	provide access to IBS catalogs
3.2.2.i	ISOFIN App Coding and Compiling Interface	IBS Developer	ISOFIN App coding and compiling interface	create ISOFIN app code
3.2.3.i	ISOFIN App Model Interface	IBS Developer	ISOFIN App model interfac	create ISOFIN app model and generate ISOFIN app code
3.3.1.i	ISOFIN App Publisher Interface	IBS Developer	ISOFIN App publisher interface	publish ISOFIN application in catalog
3.3.1.i1	ISOFIN App Repository	IBS Developer	ISOFIN App repository interface	access and publish ISOFIN applications in catalogs
3.3.2.i	ISOFIN App Deployment Interface	IBS Developer	ISOFIN App deployment interface	execute ISOFIN App deploymen

3.3.4.i	ISOFIN App Documentatio Editor	IBS Developer	ISOFIN App documentatior interface	provides an interface to allow tc get the automatically generated documentation

Appendix C - UH4SP models

Use Cases decomposition

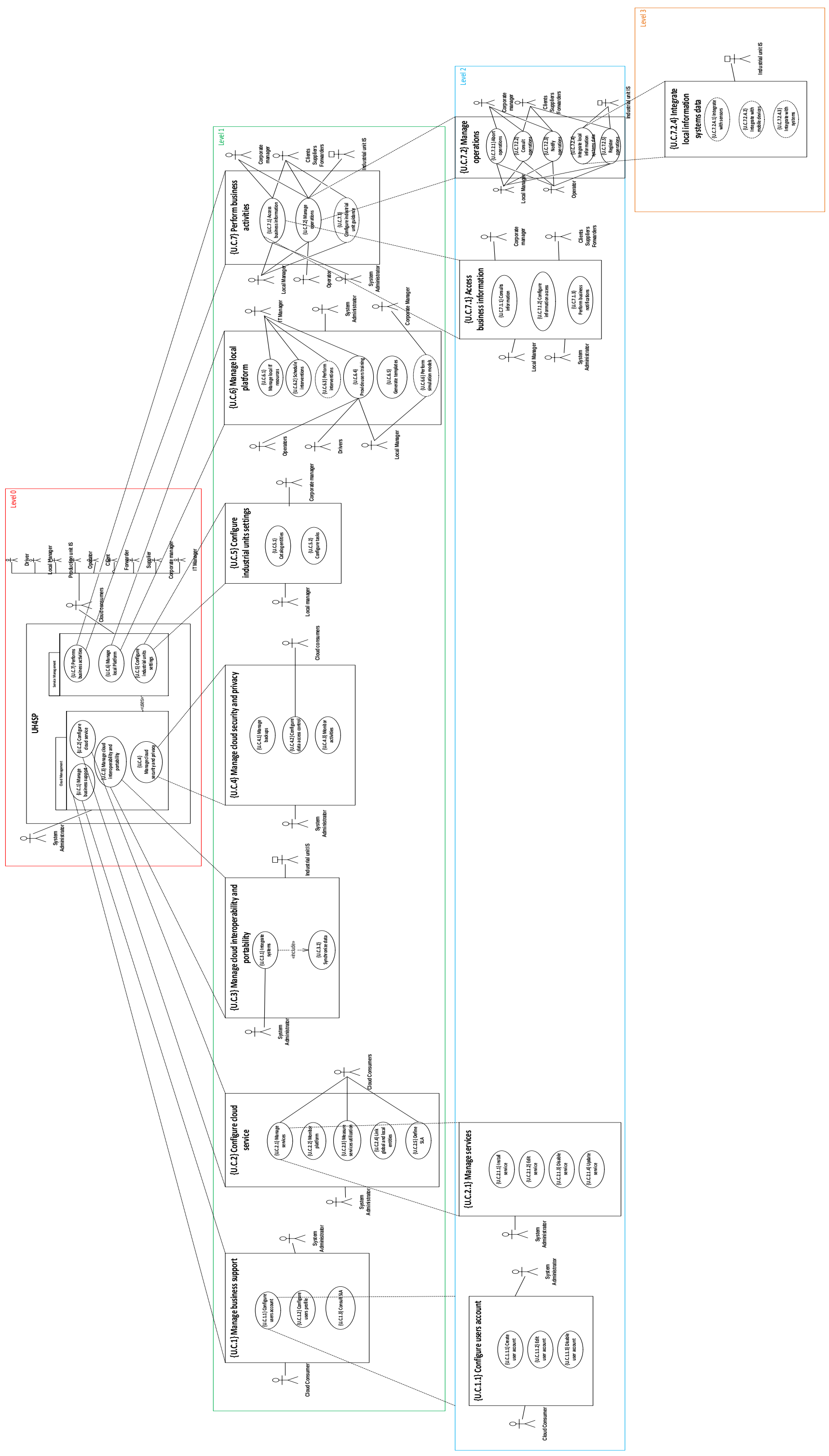

Level 0
Level 1
Level 2
Level 3
UH4SP
[U.C.1] Manage business support
[U.C.2] Configure cloud service
[U.C.3] Manage cloud interoperability and portability
[U.C.4] Manage cloud security and privacy
[U.C.5] Configure industrial units settings
[U.C.6] Manage local platform
[U.C.7] Perform business activities
[U.C.1.1] Configure users account
[U.C.2.1] Manage services
[U.C.7.1] Access business information
[U.C.7.2] Manage operations
[U.C.7.2.4] Integrate local information systems data

Figure 152. Use case model refinement

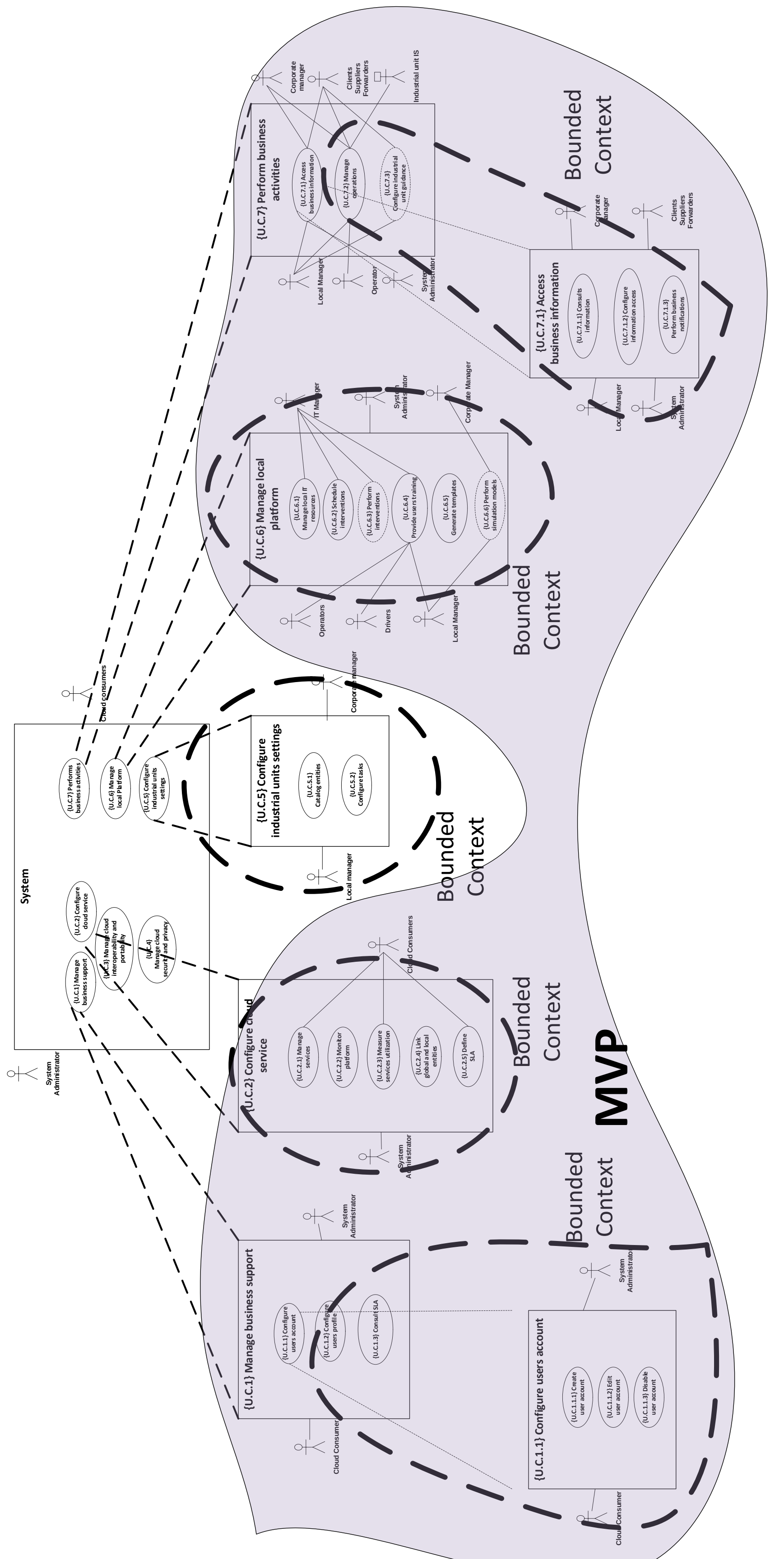
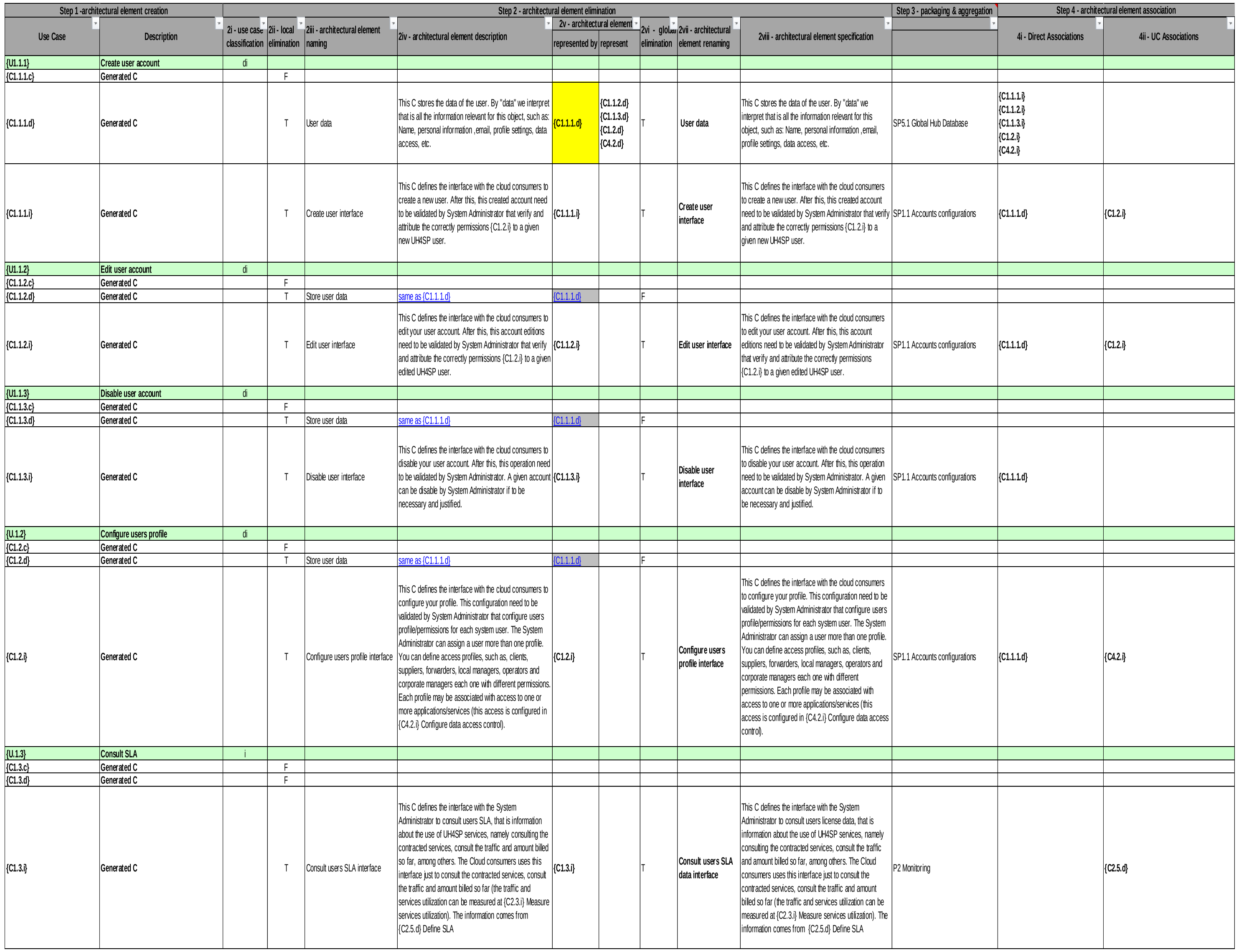

MVP
Bounded Context
System
{U.C.1} Manage business support
{U.C.2} Configure cloud service
{U.C.3} Manage cloud interoperability and portability
{U.C.4} Manage cloud security and privacy
{U.C.5} Configure industrial units settings
{U.C.6} Manage local platform
{U.C.7} Perform business activities
{U.C.1.1} Configure users account
{U.C.1.2} Configure users profile
{U.C.1.3} Consult SLA
{U.C.1.1.1} Create user account
{U.C.1.1.2} Edit user account
{U.C.1.1.3} Disable user account
{U.C.2.1} Manage services
{U.C.2.2} Monitor platform
{U.C.2.3} Measure services utilization
{U.C.2.4} Link global and local entities
{U.C.2.5} Define SLA
{U.C.5.1} Catalog entities
{U.C.5.2} Configure tasks
{U.C.6.1} Manage local IT resources
{U.C.6.2} Schedule interventions
{U.C.6.3} Perform interventions
{U.C.6.4} Provide users training
{U.C.6.5} Generate templates
{U.C.6.6} Perform simulation models
{U.C.7.1} Access business information
{U.C.7.2} Manage operations
{U.C.7.3} Configure industrial unit guidance
{U.C.7.1.1} Consults information
{U.C.7.1.2} Configure information access
{U.C.7.1.3} Perform business notifications
System Administrator
Cloud Consumer
Cloud Consumers
Corporate manager
Local manager
IT Manager
Operators
Drivers
Operator
Clients Suppliers Forwarders
Industrial unit IS

Figure. 153. Use case decomposition tree of UH4SP, the Use cases from MVP features and for further releases (Lean Startup), and the domain's and sub-domain's bounded contexts (DDD)

4SRS



Step 1 -architectural element creation		Step 2 - architectural element elimination									Step 3 - packaging & aggregation	Step 4 - architectural element association	
Use Case	Description	2i - use case classification	2ii - local elimination	2iii - architectural element naming	2iv - architectural element description	2v - architectural element represented by	represent	2vi - global elimination	2vii - architectural element renaming	2viii - architectural element specification		4i - Direct Associations	4ii - UC Associations
{U1.1.1}	Create user account	di											
{C1.1.1.c}	Generated C		F										
{C1.1.1.d}	Generated C		T	User data	This C stores the data of the user. By "data" we interpret that is all the information relevant for this object, such as: Name, personal information ,email, profile settings, data access, etc.	{C1.1.1.d}	{C1.1.2.d} {C1.1.3.d} {C1.2.d} {C4.2.d}	T	User data	This C stores the data of the user. By "data" we interpret that is all the information relevant for this object, such as: Name, personal information ,email, profile settings, data access, etc.	SP5.1 Global Hub Database	{C1.1.1.i} {C1.1.2.i} {C1.1.3.i} {C1.2.i} {C4.2.i}	
{C1.1.1.i}	Generated C		T	Create user interface	This C defines the interface with the cloud consumers to create a new user. After this, this created account need to be validated by System Administrator that verify and attribute the correctly permissions {C1.2.i} to a given new UH4SP user.	{C1.1.1.i}		T	Create user interface	This C defines the interface with the cloud consumers to create a new user. After this, this created account need to be validated by System Administrator that verify and attribute the correctly permissions {C1.2.i} to a given new UH4SP user.	SP1.1 Accounts configurations	{C1.1.1.d}	{C1.2.i}
{U1.1.2}	Edit user account	di											
{C1.1.2.c}	Generated C		F										
{C1.1.2.d}	Generated C		T	Store user data	same as {C1.1.1.d}	{C1.1.1.d}		F					
{C1.1.2.i}	Generated C		T	Edit user interface	This C defines the interface with the cloud consumers to edit your user account. After this, this account editions need to be validated by System Administrator that verify and attribute the correctly permissions {C1.2.i} to a given edited UH4SP user.	{C1.1.2.i}		T	Edit user interface	This C defines the interface with the cloud consumers to edit your user account. After this, this account editions need to be validated by System Administrator that verify and attribute the correctly permissions {C1.2.i} to a given edited UH4SP user.	SP1.1 Accounts configurations	{C1.1.1.d}	{C1.2.i}
{U1.1.3}	Disable user account	di											
{C1.1.3.c}	Generated C		F										
{C1.1.3.d}	Generated C		T	Store user data	same as {C1.1.1.d}	{C1.1.1.d}		F					
{C1.1.3.i}	Generated C		T	Disable user interface	This C defines the interface with the cloud consumers to disable your user account. After this, this operation need to be validated by System Administrator. A given account can be disable by System Administrator if to be necessary and justified.	{C1.1.3.i}		T	Disable user interface	This C defines the interface with the cloud consumers to disable your user account. After this, this operation need to be validated by System Administrator. A given account can be disable by System Administrator if to be necessary and justified.	SP1.1 Accounts configurations	{C1.1.1.d}	
{U.1.2}	Configure users profile	di											
{C1.2.c}	Generated C		F										
{C1.2.d}	Generated C		T	Store user data	same as {C1.1.1.d}	{C1.1.1.d}		F					
{C1.2.i}	Generated C		T	Configure users profile interface	This C defines the interface with the cloud consumers to configure your profile. This configuration need to be validated by System Administrator that configure users profile/permissions for each system user. The System Administrator can assign a user more than one profile. You can define access profiles, such as, clients, suppliers, forwarders, local managers, operators and corporate managers each one with different permissions. Each profile may be associated with access to one or more applications/services (this access is configured in {C4.2.i} Configure data access control).	{C1.2.i}		T	Configure users profile interface	This C defines the interface with the cloud consumers to configure your profile. This configuration need to be validated by System Administrator that configure users profile/permissions for each system user. The System Administrator can assign a user more than one profile. You can define access profiles, such as, clients, suppliers, forwarders, local managers, operators and corporate managers each one with different permissions. Each profile may be associated with access to one or more applications/services (this access is configured in {C4.2.i} Configure data access control).	SP1.1 Accounts configurations	{C1.1.1.d}	{C4.2.i}
{U.1.3}	Consult SLA	i											
{C1.3.c}	Generated C		F										
{C1.3.d}	Generated C		F										
{C1.3.i}	Generated C		T	Consult users SLA interface	This C defines the interface with the System Administrator to consult users SLA, that is information about the use of UH4SP services, namely consulting the contracted services, consult the traffic and amount billed so far, among others. The Cloud consumers uses this interface just to consult the contracted services, consult the traffic and amount billed so far (the traffic and services utilization can be measured at {C2.3.i} Measure services utilization). The information comes from {C2.5.d} Define SLA	{C1.3.i}		T	Consult users SLA data interface	This C defines the interface with the System Administrator to consult users license data, that is information about the use of UH4SP services, namely consulting the contracted services, consult the traffic and amount billed so far, among others. The Cloud consumers uses this interface just to consult the contracted services, consult the traffic and amount billed so far (the traffic and services utilization can be measured at {C2.3.i} Measure services utilization). The information comes from {C2.5.d} Define SLA	P2 Monitoring		{C2.5.d}

Figure 154. Subset of the 4SRS execution

Logical Architecture

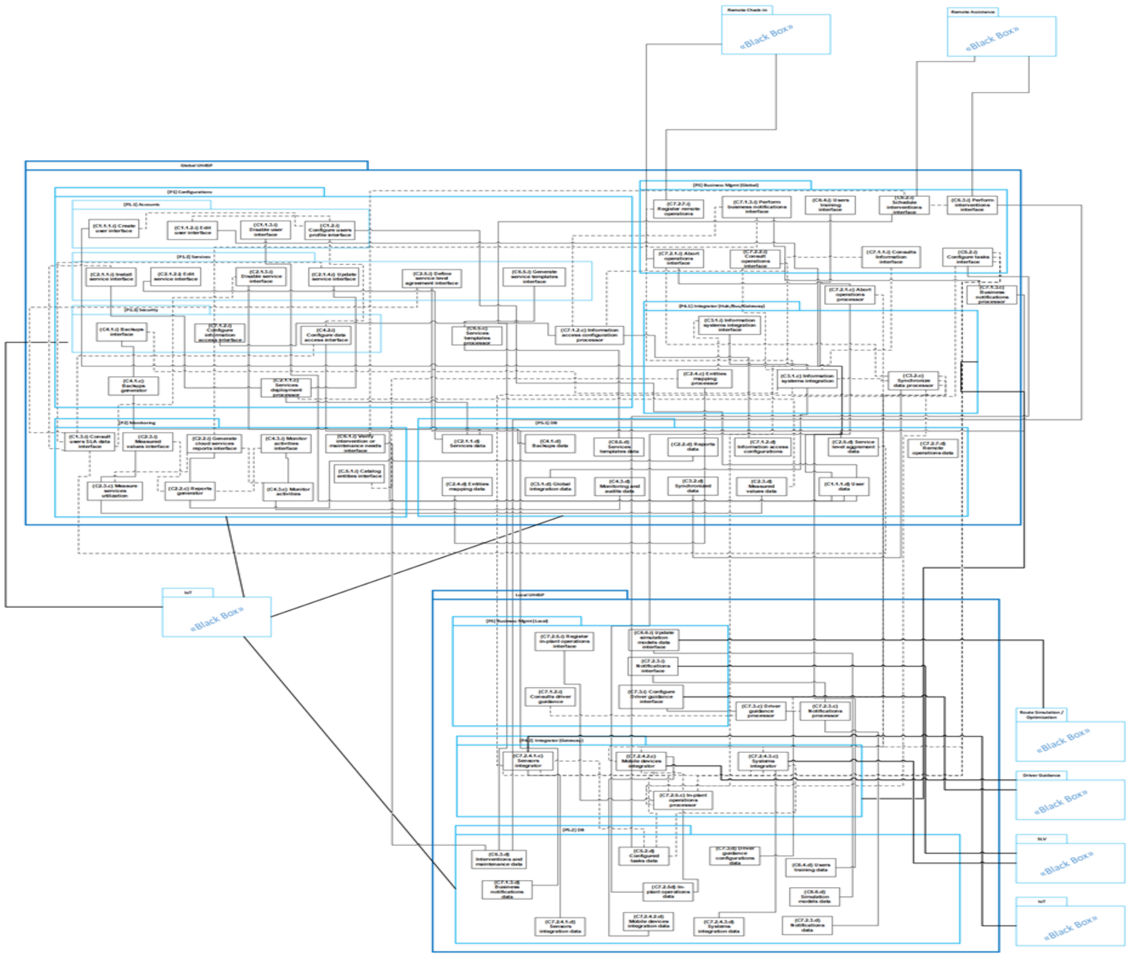
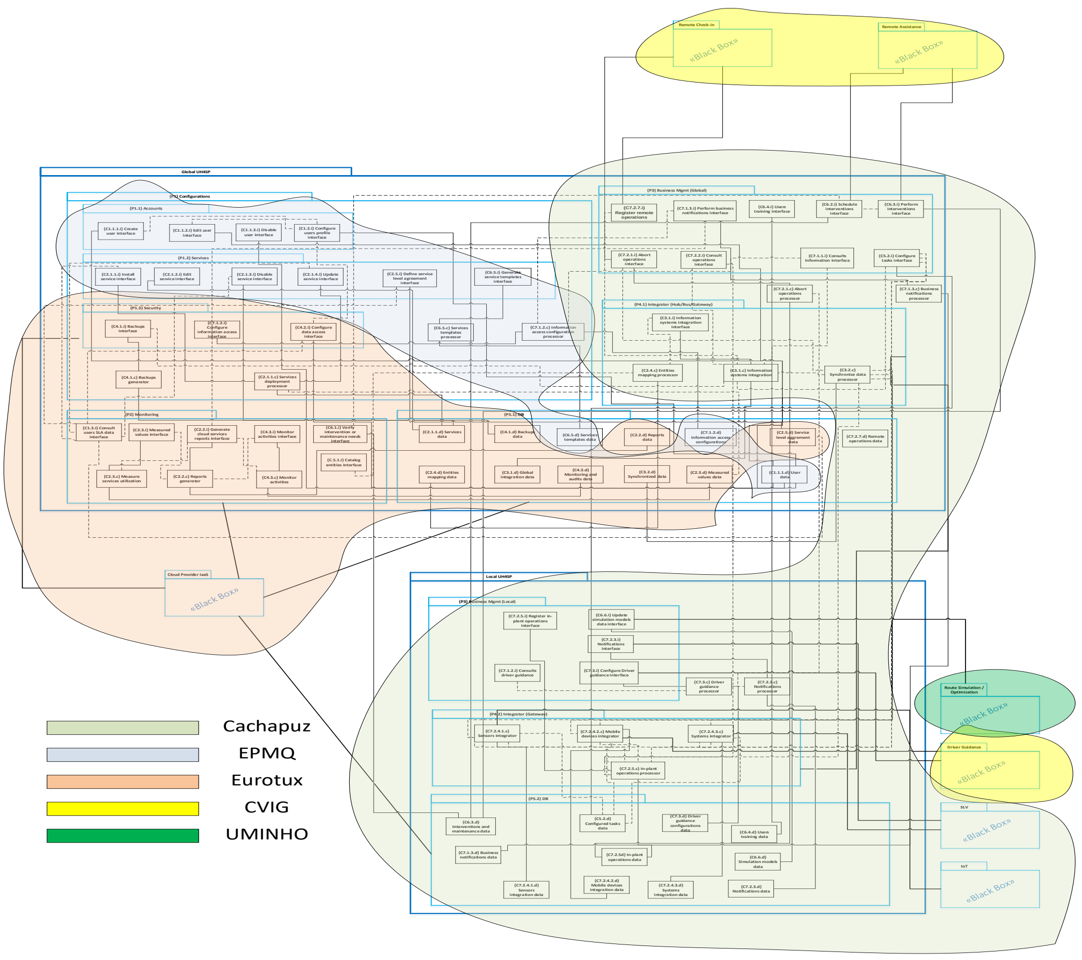


Figure 155. UH4SP logical architecture



Product Backlog

Features list:

- *Configure User profile*
- *Configure User Account*
- *Perform Authentication*
- *Manage Stakeholders*
- *Manage Trucks*
- *Manage Applications*
- *Collaborative tool*
- *Manage Work Tokens*

Table 49. UH4SP Product Backlog

Feature	**Configure users account**		
Backlog Item	**Name**	**Acceptance Criteria**	**Use Case**
User Story	As a System Administrator, Corporate manager, company manager, factory manager, forwarders, client and supplier, I want to CRUD a user account, to configure user account.	User is able to CRUD a user account	{UC1.1.1}, {UC1.1.2}, {UC1.1.3}, {UC1.1.4}
Feature	**Configure users profile**		
Backlog Item	**Name**	**Acceptance Criteria**	**Use Case**
User Story	As a System Administrator, Corporate manager, Company manager, Factory manager, Forwarders, Client and Supplier, I want to CRUD a user profile to configure user profile.	User is able to CRUD a profile	{UC.1.2.1}
User Story	As a System Administrator, Corporate manager, Company manager, Factory manager, Forwarders, Client and Supplier, I want to assign permissions to a user profile.	User profile has permissions associated	{UC.1.2.2}
Epic	Consult SLA	-	-
User Story	As a Corporate manager, Company manager, Factory manager, Client, Supplier and Forwarders, I want to consult a contract information in order to consult SLAs.	SLA contract information is consulted.	{U.1.3}
Feature	**Perform Authentication**		
Backlog Item	**Name**	**Acceptance Criteria**	**Use Case**
User Story	As a System administrator and user, I want to Insert username and password to perform authentication	1. login successful 2. account creation successful 3. login successfully	{U1.4.1}

		after password recovering 4. login successfully after username recovering 5. login unsuccessful message after wrong inputs	
User Story	As a System administrator and user, I want to recover account username or password to perform authentication	User gets password or username	{U1.4.2}
Feature	**Manage Skateholders**		
Backlog Item	**Name**	**Acceptance Criteria**	**Use Case**
User Story	As a System Administrator, I want to CRUD an industrial group in order to manage business groups.	User is able to CRUD a group	{U1.5.1}
User Story	As a System Administrator or a corporate manager, I want to CRUD a group company in order to manage group companies.	User is able to CRUD a group company	{U1.5.2.1}
User Story	As a System Administrator or a forwarder, I want to CRUD a forwarder company in order to manage forwarder companies.	User is able to CRUD a forwarder company	{U1.5.2.2}
User Story	As a System Administrator or a client admin, I want to CRUD a client companies in order to manage client companies, and last manage companies.	User is able to CRUD a client company	{U1.5.2.3}
User Story	As a System Administrator or a supplier, I want to CRUD a supplier company in order to manage supplier companies.	User is able to CRUD a supplier company	{U1.5.2.4}
User Story	As a System administrator, Corporate manager, Company manager or Factory manager, I want to CRUD factories in order to manage factories	User is able to CRUD a factory	{U1.5.3}
Feature	**Manage Trucks**		
Backlog	**Name**	**Acceptance Criteria**	**Use Case**

Item			
User Story	As a Forwarder admin or systems admin, I want to CRUD trucks in order to manage trucks.	1. User is able to CRUD a truck 2. Truck successfully created	{UC.1.7}
User Story	As a Forwarder admin or systems admin, I want to associate trailers to a given trucks.	Truck successfully created	{UC.1.7}
Feature	**Manage Trailers**		
Backlog Item	**Name**	**Acceptance Criteria**	**Use Case**
User Story	As a Forwarder admin or systems admin, I want to CRUD trailers in order to manage trailers.	1. User is able to CRUD a trailer 2. Trailer successfully created	{UC.1.8}
Feature	**Manage Applications**		
Backlog Item	**Name**	**Acceptance Criteria**	**Use Case**
User Story	As a System Administrator, I want to CRUD an application account, to configure application account.	1. User is able to CRUD an application 2. Application successfully created	{UC.1.9.1}
User Story	As an application, I want to send an app_gid and a GPS location to perform authentication, in order to access the UH4SP WebAPIs.	Authenticated application	{UC.1.9.2}
User Story	As a System Administrator, I want to assign or refresh a token to an application.	1. Token assigned 2. Token refreshed	{UC.1.9.3}
Feature	**Collaborative Tool**		
Backlog Item	**Name**	**Acceptance Criteria**	**Use Case**
User Story	As a System administrator, Corporate manager,	User is able to consult	{U.C.5.3.1}

	Company manager, and Factory manager, Forwarder, Client or Supplier, I want to visualize graphical indicators about my group, companies, factories or to resources or data that I am associated.	dashboards	
User Story	As a System administrator, Corporate manager, Company manager, and Factory manager, Forwarder, Client or Supplier, I want to visualize system operations about my groups, companies, factories or to resources or data that I am associated.	User is able to consult dashboards	{U.C.5.3.2}
User Story	As a web application, I want to receive a JSON file in order to develop dashboards.	Dashboard developed	{U.C.5.3.4}
User Story	As a System administrator, Corporate manager, Company manager, and Factory manager, Forwarder, Client or Supplier, I want to configure dashboards settings in order to configure dashboards.	Dashboard successful configured	{U.C.5.3.7}
User Story	As a service, I want to get users permissions to access to a particular data source.	The user has access to the requested data source	{U.C.5.3.1} {U.C.5.3.2}
User Story	As a user I want to perform login, in order to access collaborative web app.	The user has access to the requested application	{U.C.5.3.5}
User Story	As a user I want to recover login credentials, in order to access collaborative web app.	The user has access to the requested application	{U.C.5.3.6}
Feature	**Manage work tokens**		
Backlog Item	**Name**	**Acceptance Criteria**	**Use Case**
User Story	As a System Administrator I want to CRUD work tokens in order to manage work tokens.	Work token successful created	{U.C.1.6.4}
User Story	As a System Administrator I want to validate work tokens that was requested by managers in order to manage work tokens	Work token successful validated	{U.C.1.6.1}
User Story	As a Corporate manager I want to request, read, update and disable work tokens in order to manage work tokens to my group companies and	Work token successful requested	{U.C.1.6.4} {U.C.1.6.3} {U.C.1.6.2}

	factories		
User Story	As a Company manager I want to request, read, update and disable work tokens in order to manage work tokens to my Company factories.	Work token successful requested	{U.C.1.6.4} {U.C.1.6.3} {U.C.1.6.2}
User Story	As a Factory manager I want to request, read, update and disable work tokens in order to manage work tokens to my factory.	Work token successful requested	{U.C.1.6.4} {U.C.1.6.3} {U.C.1.6.2}
User Story	As an Entity (forwarder, client or supplier) manager I want to request, read, update and disable work tokens in order to manage work tokens to my entity.	Work token successful requested	{U.C.1.6.4} {U.C.1.6.3} {U.C.1.6.5}
User Story	As a Stakeholder/Entity manager I want to receive a notification when a given work tokens were associated to my entity in order to manage work tokens.	Notification was delivered	{U.C.1.6.2}
User Story	As an Entity (forwarder, client or supplier) manager I want to assign drivers and trucks to work tokens that were associated to my entity in order to manage work tokens.	Work token successful assigned	{U.C.1.6.5}
User Story	As a System admin I want to receive a notification when a stakeholder/entity managers request work tokens in order to validate work tokens.	Notification was delivered	{U.C.1.6.4}

Appendix D – IMP_4.0 models

Use Case model

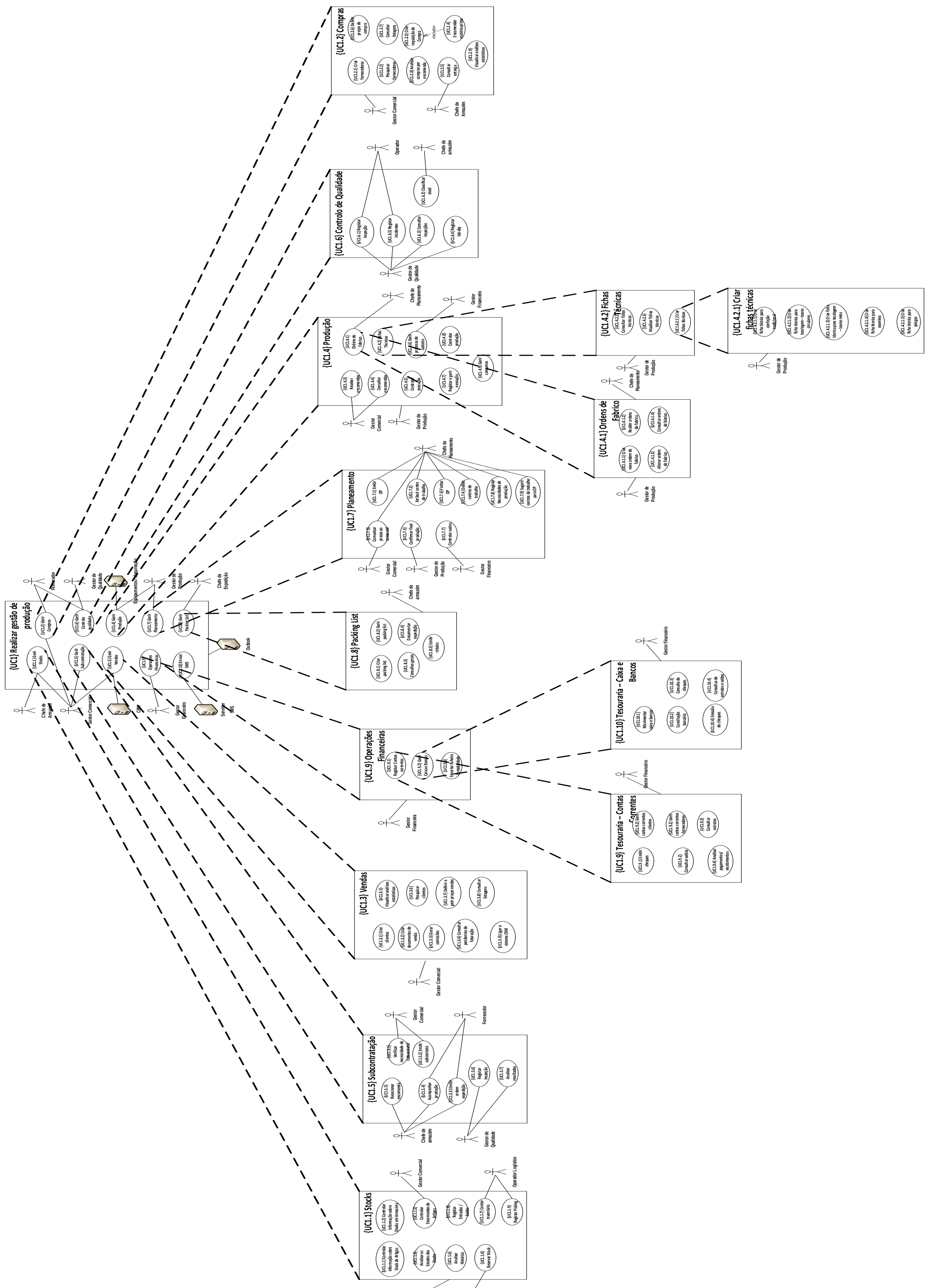


Figure 156. The decomposition of IMP_4.0 use cases

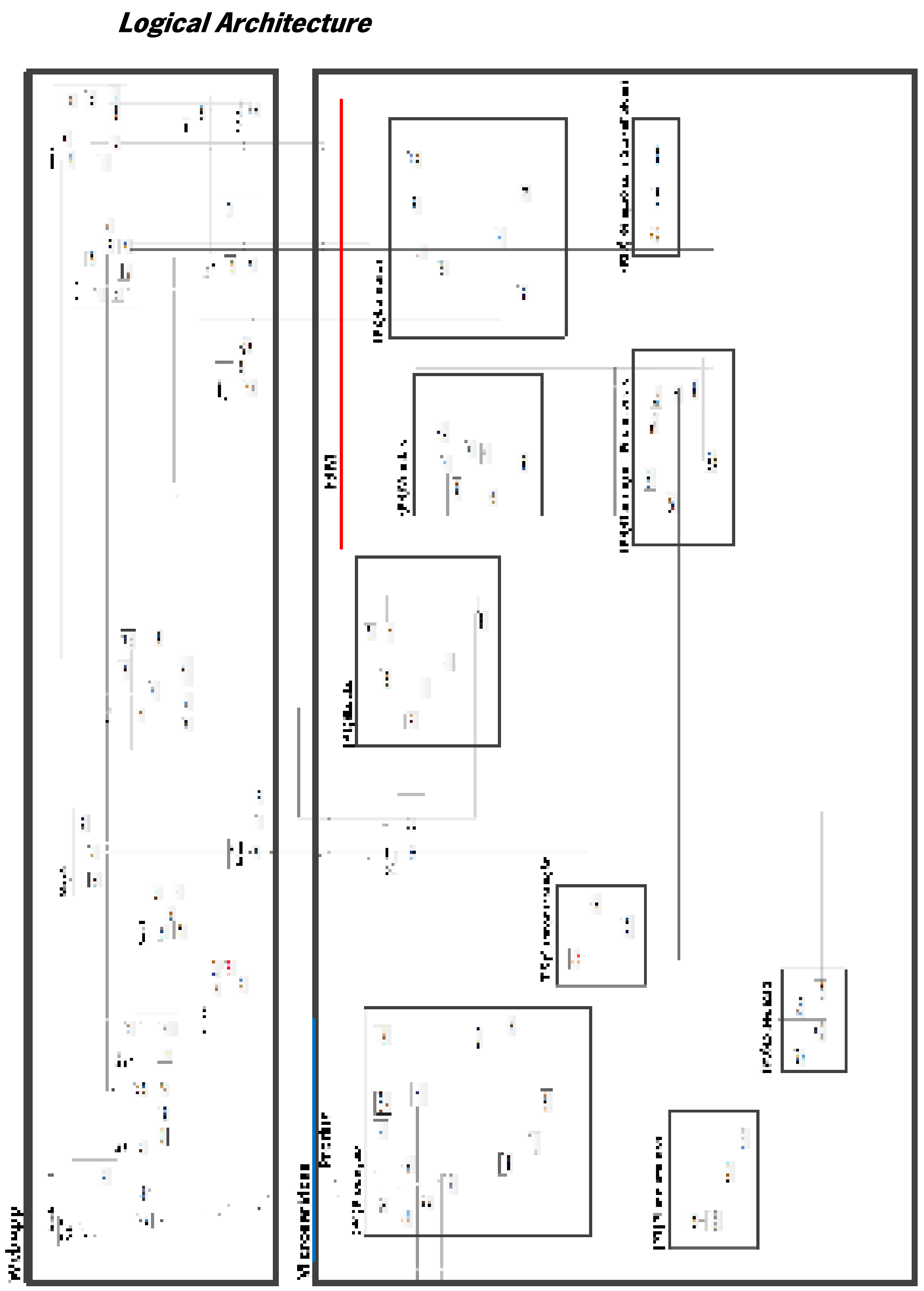


Figure 157. IMP_4.0 Component Architecture

Microservices Architecture

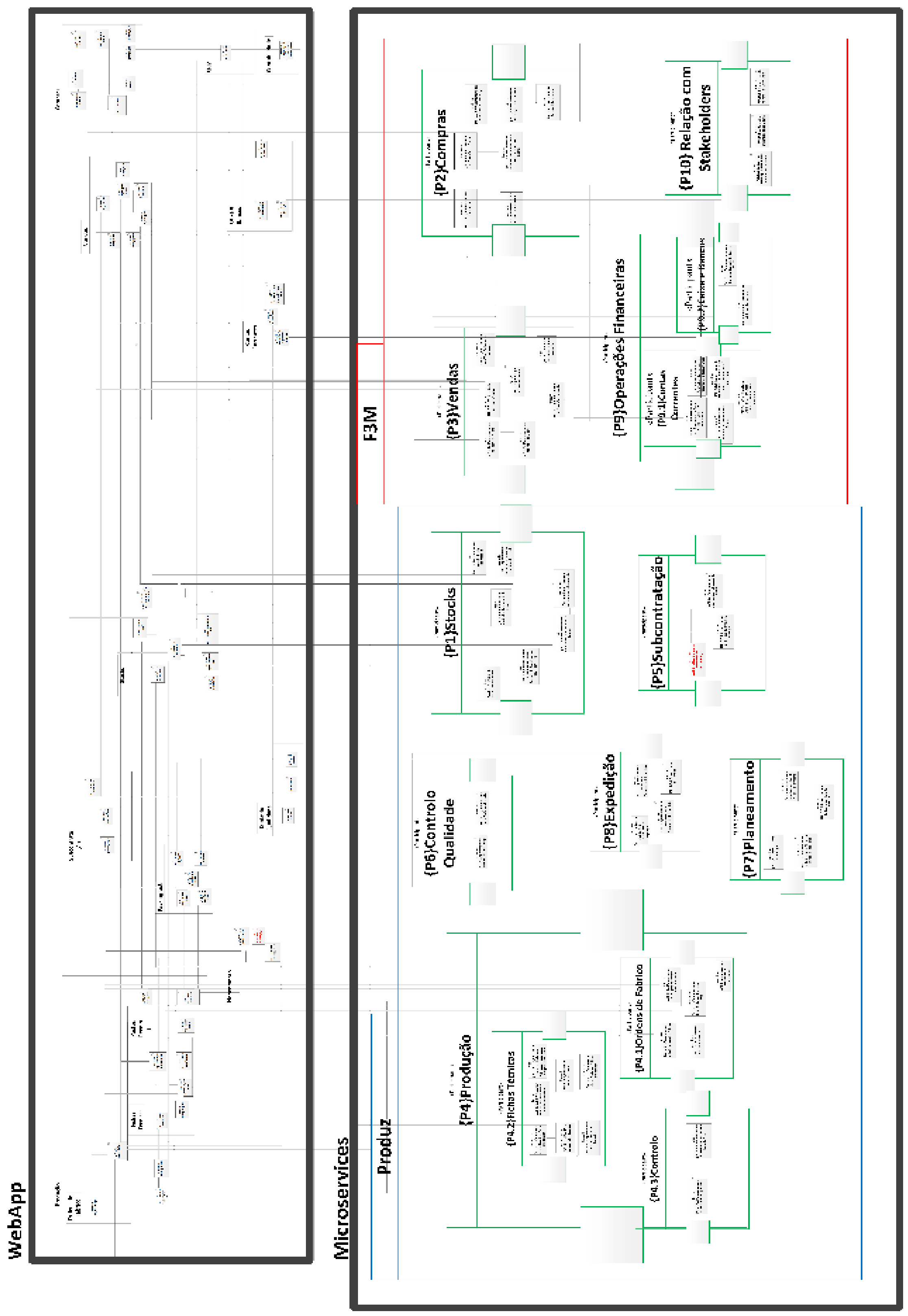


Figure 158. IMP_4.0 Microservices Architecture

Appendix E - IMSPM models

Use cases model

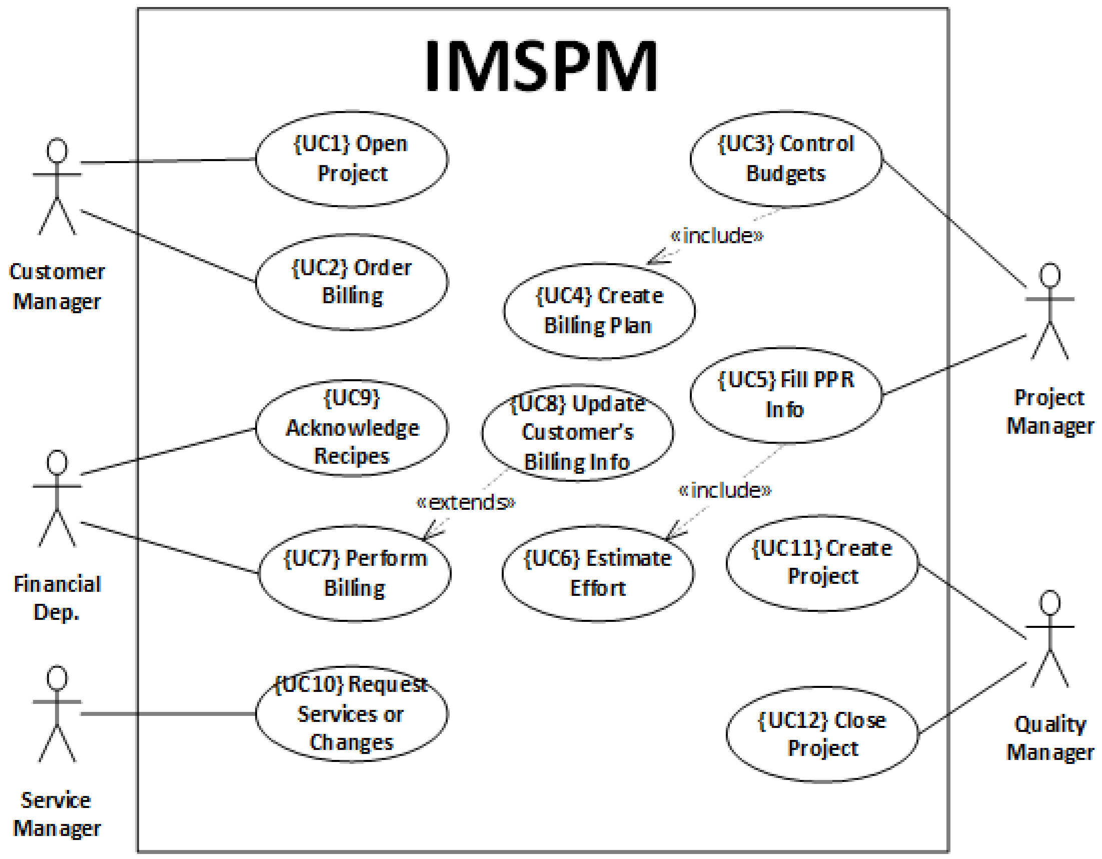


Figure 159. IMSPM Use Case model

Sequence diagram

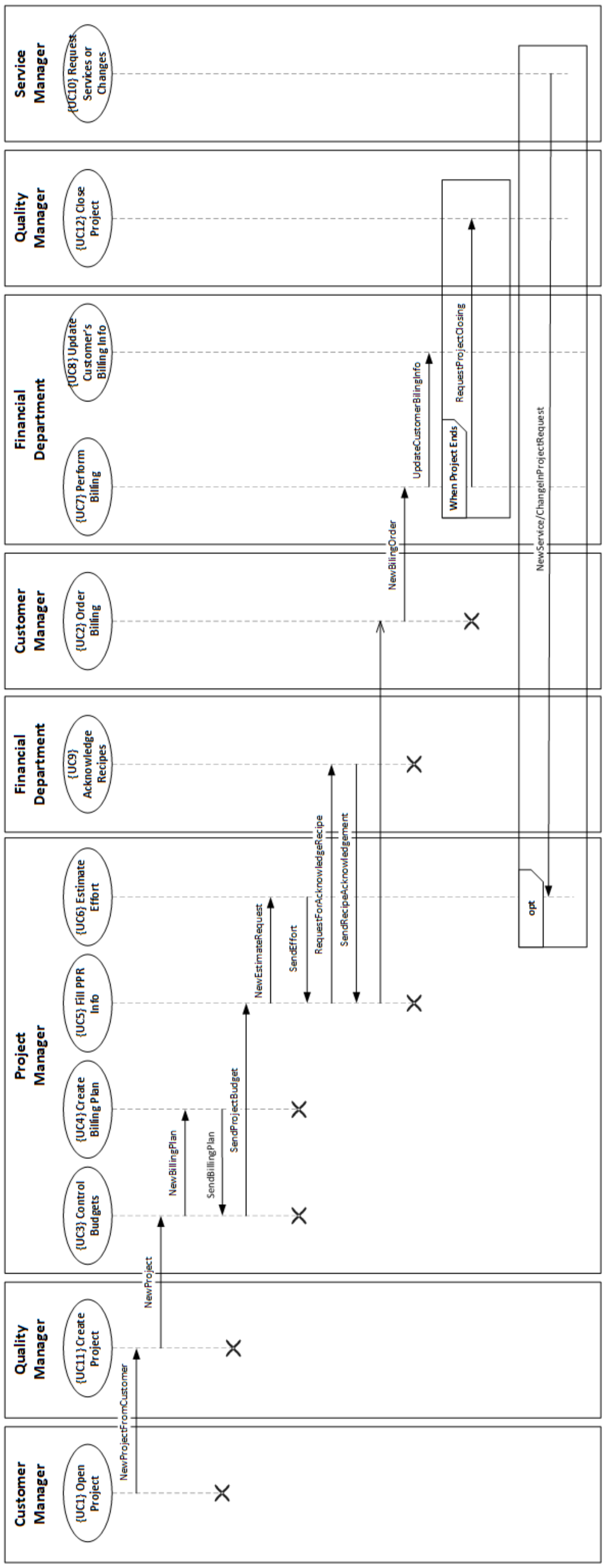
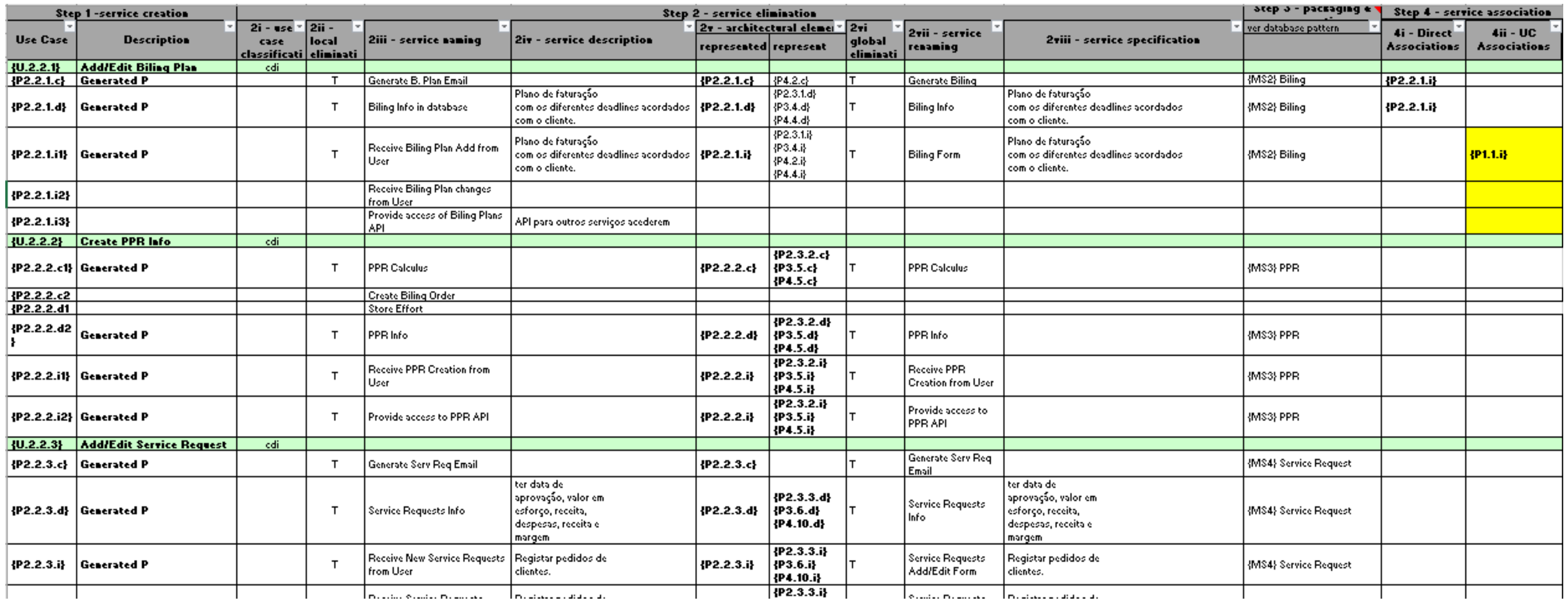


Figure 160. IMSPM Sequence diagram

4SRS



Step 1 -service creation		Step 2 - service elimination									Step 3 - packaging &	Step 4 - service association	
Use Case	Description	2i - use case classificati	2ii - local eliminati	2iii - service naming	2iv - service description	2v - architectural elemen		2vi global eliminati	2vii - service renaming	2viii - service specification	ver database pattern	4i - Direct Associations	4ii - UC Associations
						represented	represent						
{U.2.2.1}	**Add/Edit Biling Plan**	cdi											
{P2.2.1.c}	**Generated P**		T	Generate B. Plan Email		**{P2.2.1.c}**	{P4.2.c}	T	Generate Biling		{MS2} Biling	**{P2.2.1.i}**	
{P2.2.1.d}	**Generated P**		T	Biling Info in database	Plano de faturação com os diferentes deadlines acordados com o cliente.	**{P2.2.1.d}**	{P2.3.1.d} {P3.4.d} {P4.4.d}	T	Biling Info	Plano de faturação com os diferentes deadlines acordados com o cliente.	{MS2} Biling	**{P2.2.1.i}**	
{P2.2.1.i1}	**Generated P**		T	Receive Biling Plan Add from User	Plano de faturação com os diferentes deadlines acordados com o cliente.	**{P2.2.1.i}**	{P2.3.1.i} {P3.4.i} {P4.2.i} {P4.4.i}	T	Biling Form	Plano de faturação com os diferentes deadlines acordados com o cliente.	{MS2} Biling		**{P1.1.i}**
{P2.2.1.i2}				Receive Biling Plan changes from User									
{P2.2.1.i3}				Provide access of Biling Plans API	API para outros serviços acederem								
{U.2.2.2}	**Create PPR Info**	cdi											
{P2.2.2.c1}	**Generated P**		T	PPR Calculus		**{P2.2.2.c}**	**{P2.3.2.c} {P3.5.c} {P4.5.c}**	T	PPR Calculus		{MS3} PPR		
{P2.2.2.c2				Create Biling Order									
{P2.2.2.d1				Store Effort									
{P2.2.2.d2 }	**Generated P**		T	PPR Info		**{P2.2.2.d}**	**{P2.3.2.d} {P3.5.d} {P4.5.d}**	T	PPR Info		{MS3} PPR		
{P2.2.2.i1}	**Generated P**		T	Receive PPR Creation from User		**{P2.2.2.i}**	**{P2.3.2.i} {P3.5.i} {P4.5.i}**	T	Receive PPR Creation from User		{MS3} PPR		
{P2.2.2.i2}	**Generated P**		T	Provide access to PPR API		**{P2.2.2.i}**	**{P2.3.2.i} {P3.5.i} {P4.5.i}**	T	Provide access to PPR API		{MS3} PPR		
{U.2.2.3}	**Add/Edit Service Request**	cdi											
{P2.2.3.c}	**Generated P**		T	Generate Serv Req Email		**{P2.2.3.c}**		T	Generate Serv Req Email		{MS4} Service Request		
{P2.2.3.d}	**Generated P**		T	Service Requests Info	ter data de aprovação, valor em esforço, receita, despesas, receita e margem	**{P2.2.3.d}**	**{P2.3.3.d} {P3.6.d} {P4.10.d}**	T	Service Requests Info	ter data de aprovação, valor em esforço, receita, despesas, receita e margem	{MS4} Service Request		
{P2.2.3.i}	**Generated P**		T	Receive New Service Requests from User	Registar pedidos de clientes.	**{P2.2.3.i}**	**{P2.3.3.i} {P3.6.i} {P4.10.i}**	T	Service Requests Add/Edit Form	Registar pedidos de clientes.	{MS4} Service Request		
				[illegible]	[illegible]		**{P2.3.3.i}**		[illegible]	[illegible]			

Figure 161. IMSPM 4SRS-MSLA execution

Logical Architecture

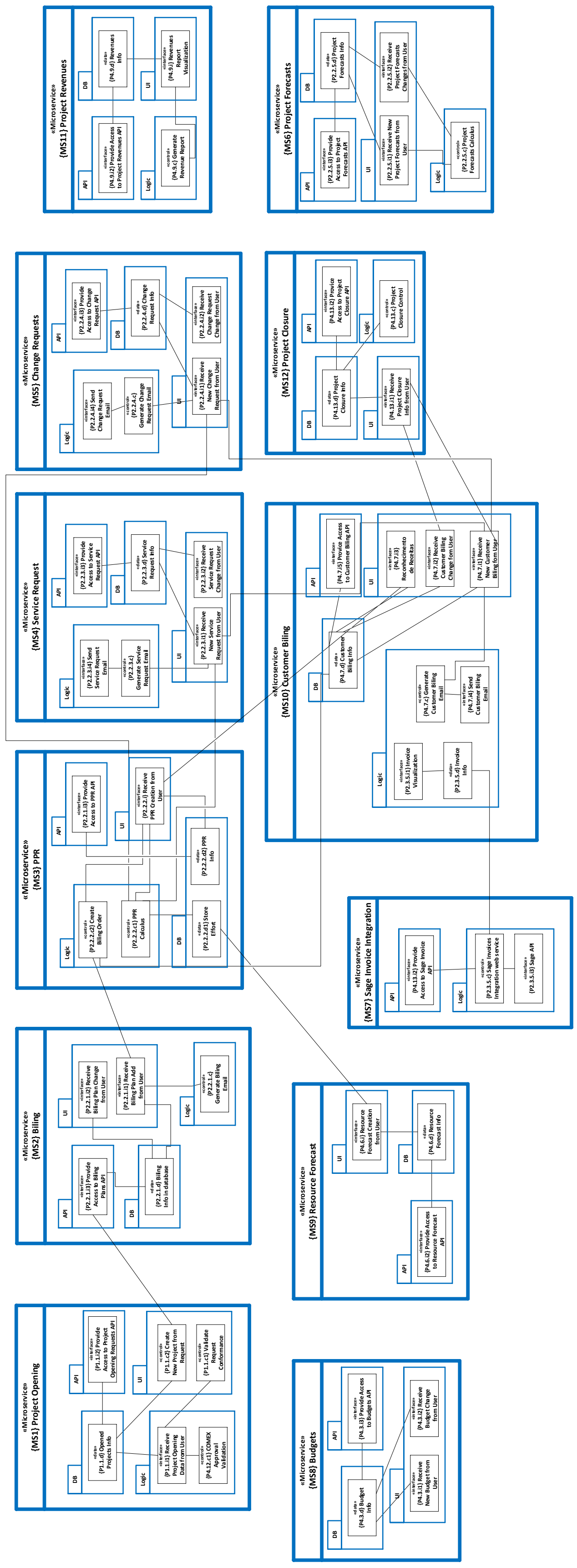

«Microservice» {MS11} Project Revenues
«Microservice» {MS6} Project Forecasts
«Microservice» {MS5} Change Requests
«Microservice» {MS12} Project Closure
«Microservice» {MS4} Service Request
«Microservice» {MS10} Customer Biling
«Microservice» {MS3} PPR
«Microservice» {MS7} Sage Invoice Integration
«Microservice» {MS2} Biling
«Microservice» {MS9} Resource Forecast
«Microservice» {MS1} Project Opening
«Microservice» {MS8} Budgets
API
DB
UI
Logic

Figure 162. IMSPM Microservices Architecture