Integrating HCI concerns into a UML based Software Engineering course

António Nestor Ribeiro
anr@di.uminho.pt

José Creissac Campos
jose.campos@di.uminho.pt

F. Mário Martins
fmm@di.uminho.pt

Departamento de Informática/CCTC
Universidade do Minho
Campus de Gualtar, 4710-057 Braga, Portugal

ABSTRACT

Software Engineering (SE) and HCI (Human Computer Interaction) are not the same age, do not have the same history, background or foundations, and did never share design principles and design models. The separation principle, by encouraging separate concerns and techniques to design the interactive and the computational layers of a software system - despite being absolutely correct from several SE crucial design principles, like modularity, separation of concerns, encapsulation, context independence and so on -, has sometimes been misjudged and mistakenly used. Therefore, instead of bridging the gap between the two separate designs, it helped widening that gap. However, the principle does not mention and does not impose any restrictions on how the integration should be done.

In the context of a software engineering course the authors have been involved with for some years, the need has arisen to provide students with HCI skills. Several attempts at integrating HCI into software engineering can be found in the literature. However, none seemed amenable to application in the context of the course, basically because none of them could be taught and learnt in such a way (methodology) that could easily be blent into the software engineering design process. We present a methodological process that we have been teaching that aims at shortening the gap that software engineering students face when trying to adapt SE techniques to the interactive layer.

Keywords

Model based design, UML, user centred design, UI modelling, software engineering

1. INTRODUCTION

This paper arises from experience by the authors in teaching a software engineering course with an emphasis in model based development. More specifically, a course on UML (Unified Modelling Language) [1] based modelling and development of software systems.

The course is taught every other semester to about 120 fourth year students of software engineering and computer science *licentiate* degrees (5 years). The students reach the course with good programming skills (including object oriented programming) but little knowledge of Human-Computer Interaction (HCI) related issues.

The course is organised into theoretical and practical lectures, complemented with independent team work and tutorial support. During the semester the students (organised in teams of 3 to 5 students) engage in a project were they must develop a software system using the entire set of knowledge needed to go from analysis, to specification and modelling, to development and later deployment.

Initially the course followed an approach close to IBM's RUP (Rational Unified Process) [11], going through the usual software development stages:

- requirements gathering
- architectural analysis and design
- behavioural analysis and design
- implementation
- deployment

Development of the software systems followed the traditional 3-tired architectural approach: User Interface layer, Business Logic layer and Database layer (see Figure 1 in section 5.2).

When this approach started being applied, it became evident that the students had great difficult in creating a complete understanding of the system to be built from the modelling process and its outcomes. One major difficulty was derived from the fact that the approach was mainly addressing the business logic layer of the application, but little was said about the data layer and user interface layers. Regarding the database layer the students already had notions of databases and were able to bridge the object relational mapping with minimal support.

The user interface layer, however, presented greater problems:

- On the one side, most of the 'logic' derived from the use case models could not be directly expressed in the business logic. This happened because we take the view that the use case model describes the system from an external perspective – i.e. from the perspective of its (user) interface to the outside –, while the business logic implements the services that support those interfaces, not the interfaces themselves;
- On the other side, without a model/understanding of the user interface, it was not always completely clear what the functionalities of the business logic should be.

It became clear and well assumed that the students were having problems bridging the (considerable) gap between use case model and the architectural model.

In order to solve this issue we needed to include user interface modelling and development into the course syllabus. The idea being that the user interface (UI) should act as a bridge between the outside (use case) and the inside (architectural) view of the system.

This had to be done in the context of the UML/RUP based approach already in place, keeping the focus on a model based approach to software systems development, and trying to create as little disruption and additional work load to the students as possible (the course was already a high load course as it was). The approach taken was to identify how best to use UML in the development of interactive systems. This paper presents the envisaged process – i.e. how to effectively use UML to design and model the UI layer and its integration to the rest of the system – and briefly discusses the results of that experience.

The rest of the paper is organised as follows. Section 2 discusses model based analysis and the impact it has on development. Section 3 addresses the use of UML as a standard modelling language and its adequacy to HCI and interactive systems. Section 4 distinguishes between software engineering and interactive systems processes. Section 5 presents our methodological approach to teaching HCI concepts in a software engineering setting. In Section 6 the usage of UML diagrams to assist the interactive layer modelling is explained in detail. In Section 7 an example is introduced in which we briefly illustrate some results of project work carried out according to the approach described. Finally, in Section 8 the conclusions we have thus far reached from the application of the approach can be found.

2. MODEL BASED ANALYSIS AND DEVELOPMENT

The use of models has become a standard technique when dealing with complexity in software systems' development. The use of models has two main purposes:

- helps understanding a complex problem/solution — a good model represents a adequately simplified version of the problem/solution, making it easier to grasp what is essential about it;
- helps communicating complex problems/solutions — once the model is produced it can be used to communicate information to others (assuming they will be able to understand it).

The main modelling artifacts in UML are diagrams (the language identifies 12 different diagram types). UML has diagrams for many different purposes, which can be used at different levels of abstraction. The use of these diagrams can vary in:

- formality — they can range from very informal "*back of the envelope*" sketches to more formal models (for example, using OCL) of specific aspects of the system; typically, as the level of detail increases, so does decrease the range of features that can be expressed in the model;
- view — different diagrams will address different aspects of the system; a typical distinction is between structural and behavioural models;
- purpose — different needs will typically demand different types of models; a common distinction is that made between conceptual models (used for describing the problem domain), specification models (use for describing what the solution to the problem is), and implementation models (used for describing how the solution is implemented).

In any case a process of abstraction is used to focus the attention on the relevant issues that must be considered. One of the consequences of the abstraction process is that diagrams will reflect a partial view of the system. This view is determined by the combination of the factors just described.

Hence, a UML model is built from a collection of different diagrams, expressing different views of the system at the appropriate level abstraction.

3. UML AND INTERACTIVE SYSTEMS

The Unified modelling Language (UML) was an obvious choice of modelling language when initially preparing the course. The UML is nowadays the standard language for modelling object oriented software systems. To practitioners, novice or expert ones, the object oriented approach offers a high degree of affinity in what concerns the development of interactive systems, once it allows layer independency as stated in software design patterns such as MVC [13]. Nevertheless both UML, as the reference modelling language, and the Rational Unified Process [11] development process are notoriously insufficient when modelling interactive systems.

Several approaches to adapting the UML to best suite the modelling of interactive systems have been put forward over the last years. Despite their intrinsic value the adoption of these proposals has been slow and was not adequate for our purpose. They did not have the same usage scope and some of the proposals introduced new constructors into the language syntactic set.

We can divide these proposals into two groups: those that advocate the extention of the UML language to address HCI related models; those that make use of the UML's extention mechanisms (namely profiles) to taylor the UML to the needs of interactive systems modelling.

In the first group we can inlude Paternò's proposal of integrating CTT (ConcurTaskTrees [18] – a task modelling language) into the UML, or the proposal by Nobrega et al. [14] of an increment to the UML's abstract syntax in order to model tasks.

In the second group we have approaches such as WISDOM [16] and UMLi [19]. In this case, UML profiles are defined to capture the decisive aspects of interactive systems modelling. However these strategies pay more attention to the user interface modelling than to the process that defines what the user interface should be.

Practitioners have been adopting these proposals very slowly. This is due to the fact that they require the learning of a new language, which is definitely an obstacle to the modelling process. On the other hand the effort that software engineers must devote in order to master a set of different profiles and languages, is a natural obstacle to the popularity of these profiles. We specifically wanted to avoid introducing new constructs since one of the learning outcomes of the course is knowing UML, and introducing additional notations would create confusion.

Some of the proposed profiles are mainly targeted at modelling the user interface and the graphical objects within. In that sense we can argue that nowadays IDE's master that task in a reasonable way allowing the direct graphical edition of the UI. This might be criticised as promoting permature commitment. However, students are encouraged to leave the definition of the concrete user interface to the coding stages, and to use paper prototyping techniques during the analysis stages. Because the course delays coding phase to the quarter of the semester, permature commitment to a concrete user interface is avoided.

We could have adopted a more structured approach to user interface prototyping (for example, using Canonical Abstract Prototypes [2]), however that would mean deviating from the standard UML notation. Something we did not want to do. In any case, it should be stressed that the question is not only how to specify and change the interface layout but on how to describe the dialogue between end-users and the application itself, as well as the structure of the code in order to support the intended dialogue.

The recent version of UML, UML 2.0, brings some new modelling constructors reinforcing the idea that it is important to exploit the best way to use the language in the context of an interactive layer modelling. As UML is a well-established modelling language being thoroughly used and supported by a large community of practitioners, and being taught to our students during their course, it is relevant to explore which is the best manner to use the language - in its standard definition - in order to take the best usage of it when modelling an interactive system. In that sense our primary goal is to allow that a software engineer with a common understanding of the language might not only be able to use it properly in order to specify, model and develop an interactive system, but also to integrate his or hers skills into a software engineering development team with no knowledge of HCI specific notations.

In the context of the software engineering course we teach, we developed a methodological framework to allow for an improved usage of UML in interactive systems modelling. We specifically avoided including new language constructors, instead we propose a new way to integrate actual standards and modelling best practices in order to improve the correctness of the overall model.

Hence we explore the ability to use UML in order to build coherent models in which the interactive layer is fully detailed. We propose a method in order to bring into the analysis and design phase aspects and facets related with the interactive dialogue that are often neglected by the modelling processes.

4. SOFTWARE ENGINEERING VS. INTERACTIVE SYSTEMS DEVELOPMENT

When trying to incorporate HCI related issues into a software engineering process we are faced with the fact that the theories and practices of software engineering and those of human-computer interaction have, to a great extent, evolved separately. Software engineering deals with the construction of software systems. Despite its infancy, from programming technology to software development process, a large body of tools and knowledge has been produced (cf. [9]). However, developing software is still a mostly difficult and complex process.

HCI is concerned with the process of communication between humans and computer systems. In the context of our course we are mainly interested in software interactive systems. This field is younger than software engineering, but also in this case a considerable body of knowledge has been developed (see, for example, [4, 15]).

Since the focus is on the interaction between system and users, the techniques developed within HCI for interactive systems development deal mainly with what can be called the interface layer of systems. Despite all progress, it is also true for interactive systems that developing them is a difficult and complex process. It is estimated that 60% to 90% of all system failures can be attributed to problems in the interaction between the systems and their users [7]. The problems with interactive systems development are not particularly surprising since interactive systems are a special case of software systems with the added complexity of having to cope with human activities, goals, capabilities and limitations.

Practitioners from both fields have developed distinct skills. The course already covered software engineering skills, we now felt the need to include enough of HCI related skills to enable students to develop better user interfaces for their systems, and help them create a clearer picture of the system being developed.

Decisions regarding the user interface design can have serious implications on the implementation of the whole system, not just the user interface implementation. Even in the case of non-interactive systems, input from the HCI body of knowledge can help in understanding the impact of the system in the overall context when this context involves humans. In practice, however, reconciling the two views on development becomes difficult. This can be attributed to a number of factors, such as:

- different views on where the development focus lies — software engineering is mainly interested in solving the technical difficulties faced when implementing a given functionality; HCI is mainly about solving the problem of which functionality should be provided, and how to optimise the way in which it is provided to users;
- communication difficulties — not always the same terms are used to describe the same concepts in the two communities; this hinders communication, at best can make it difficult and at worst can mislead the two parties into thinking that they are talking about the same thing when in fact they are not. This is particularly relevant in a teaching context since concepts need to be presented in a clear and non confusing way.

5. THE PROPOSED APPROACH

In order to understand why the differences between SE and HCI exist we can look at typical development processes used by each community. This enables us to identify how the previously identified differences manifest themselves.

5.1 Human-Centred Design vs. RUP

Software engineering methods (for example, the Rational Unified Process [11]) are mostly concerned with building the system. Typically the requirements gathering phase attempts to determine the functionality the system should provide, and the focus quickly shifts to the issue of how to better implement that functionality. The main concerns are the quality and maintainability of the code produced. Usability issues are seldomly mentioned, if at all.

Interactive systems development methods (for example, human-centred design [10]) on the contrary, are more concerned with the design of the interaction between the system and its users. The focus of attention is on how best to support the users in performing specific activities with the aid of the system in concrete contexts of usage.

In [12] a brief comparison between the ISO 13407 standard for human-centred design (HCD) [10] and RUP is made. There we can conclude that the human-centred design (HCD) process is primarily concerned with the design of the system, while the Rational Unified Process (RUP) primarily concerned with the implementation of the system. Both methods are based on prototyping and iteration. However, these terms do not necessarily mean exactly the same in both contexts. In RUP, prototypes are mainly executable code and mostly seen as intermediate steps in the development of the final system (a partially implemented system). In HCD prototypes are simulations or models of the user interface of the system. They are developed for usability testing purposes, and need not to be executable.

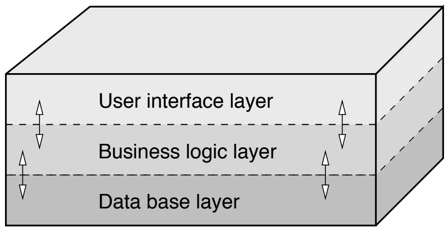

Figure 1: 3-tier architecture

5.2 Different perspectives on development

From the above we can conclude that the two disciplines have different perspectives on development. HCI is primarily interested in developing the "*outside*" of the system. That is, the interaction of the system with its users. Software engineering is primarily interested in the "*inside*" of the system. That is, how the system is actually implemented. We will say that HCI has a black box view of the system, while software engineering has a white box view of the same system.

Software engineers think about the system in terms of its internal architecture, encompassing how components are organised and communicate. It is common for this architectural view to be organised as a set of overlapped layers. An example of this organisation is the 3-tier architecture presented in Figure 1.

Note that in this architectural model there is no mention to the user. In fact, despite mentioning the user interface, most of the initial development effort usually goes into the business and data layers, and concerns about the user interface, when present, are geared towards its implementation.

A usability practitioner will typically talk of users' goals, tasks, and user interface designs, without deeper consideration of the architectural issues and tradeoffs "*behind the scene*". The focus is on the interaction between user and artifact, not on the artifact by itself. It is usual to see references to the "interactive system" as the composition of human + system (system, in a software engineering sense).

These different views of the system (development) lead to differences on how the available *tools* are applied. Nevertheless these two different processes are not incompatible and how to best establish a communication link between them is a problem that deserves study. Namely, regarding synchronization points between the teams responsible for modelling different parts of the system being developed within a project.

The main differences between the two processes lie in the audience they are targeted at and in the different perceptions of what the result should be. HCD is intended to define the requirements gathering process related to the interactive layer and the subsequent prototyping and UI validation. On the other hand RUP is targeted at defining the activities that must be carried out in order to successfully deliver a software system according to requirements. RUP does not recognize the UI layer as a particular sub-system with its own specificities and treats it as one more of the components that compose the overall system.

5.3 Main stages of the process

We have seen that UML/RUP does not offer native support to the process of user interface design and modelling. Hence, it is mandatory that we introduce some changes to how UML diagrams are used in order to meet our objective. The changes we propose to the standard process imply that some diagrams must be used earlier than what is recommended in RUP. In that way we are able to gather information relevant to the interactive layer definition earlier in the process. Usually this information would be splintered throughout the several views that RUP addresses.

The most important modelling phases of the adopted process are depicted bellow. The diagrams used in each step are also introduced.

1. Requirements gathering – both in HCD and RUP the requirements gathering phase is a crucial part of the process. However in RUP and in UML authoring tools the requirements related to the interactive layer are mixed with the overall requirements of the system. All these requirements are collected using the Use Case diagrams. Use case diagrams collect all the information about the requirements the system must meet. Among those requirements it is possible to collect the ones related to the interactive layer. For each use case identified it is necessary to depict scenarios that fully describe it. That information is valuable in order to establish the flow of the interactive dialogue between the user and the system.

2. Task analysis & design – these activities are supported by the use case model (cf. [3]). Besides the use case diagrams themselves, other UML behavioural diagrams may be used to formally describe the behaviour underlying the identified use cases. In [14] a translation from CTT to activity diagrams is proposed. In [6] sequence diagrams are used to model tasks.

   Although there is no complete formal mapping between the typical task modelling strategies used in a Hierarchical Task Analysis (HTA) approach (using for instance CTT diagrams [17]) and UML activity diagrams, we have adopted the latter to describe use case behaviour.

3. Dialogue design – the dialogue structure that supports the behaviour specified in the activity diagrams can be described in UML using a statechart diagram (c.f., [8]).

   In this design stage the requirements for the business logic layer are formally described by identifying the API methods needed for the state transitions at the UI level. This phase is of crucial importance since it allows, and promotes, the discovery of the business logic methods that the interactive layer will use.

4. Architectural design – during the previous stage, the software engineers, or software engineering students, will discover most of the methods associated with transitions in the statechart diagram. While discovering the methods the project team may also describe in the class diagram the classes (the entities) and their methods. From this point on, the rest of the process is very similar to the typical UML/RUP modelling approach.

5. Behaviour modelling – in this phase and using UML behavioural diagrams we can describe the different aspects that model how the system will behave. Special attention should be paid to the description of object interaction in order to fulfill the identified requirements.

6. Deployment – where the typical aspects concerning the installation, configuration and deployment are addressed.

Comparing the above list to the original one, introduced in section 1, it can be seen that we have introduced two new phases (task analysis and dialogue design). Additionally we made some changes to the other phases in order to accommodate the new ones. This is particularly true in the Requirements phase. Section 6.1 describe these three phases in more detail.

5.4 Insights on the architecture design

As stated previously, and concerning the structure of the code to be developed, we adopt a standard three-tiered model, and we propose an MVC approach based on the Observer/ Observable [5] software pattern. This strategy allows independence between the user interface layer and the business layer.

The architecture of the user interface layer must be understood as the set of software components that are available at the UI level together with their physical layout. This architectural design may be conducted by direct manipulation in the context of an IDE. Therefore the used modelling language does not need to support that description syntax, although it is important that the semantic distance between the two description levels is not considerable.

We note that a UI architectural design phase is not formally identified above. Students are encouraged to perform basic paper prototyping of the user interface during the dialog design phase. Given the nature of the course, going into detail in the UI graphical design area would be out of scope.

An interesting alternative is to consider a logical distinction between the architecture of the interactive layer and the architecture of the remaining software system. In the end that will lead to a clearer definition of the overall architecture promoting a separate management of the modelling process.

6. UML IN THE DEVELOPMENT PROCESS

In this section, in order to prove our point and to assess the proposed integration method, we describe how UML diagrams are used to model and develop the interactive layer.

RUP does not give enough, explicit support for these tasks, since it was not developed with such tasks in mind or main focus. RUP practitioners never thought that it could be possible to use RUP diagrams to incorporate interactive layer support into the usual modelling process. Therefore, we propose a non-official, under development methodology, based on using UML well known modelling constructs to also model, in an integrated way, the interactive layer, with no knowledge disruption from the process of modelling the business layer.

An alternative route to achieve this goal would be the creation been the creation of extensions to the language or to develop dedicated environments to model and specify the UI. That would be a straightforward path although that would close the process, making it heavily dependent on the used tools. On the other hand, we claim that both interactive and business logic layers are alike in what concerns software engineering principles. If we had chosen to build specific tools we would be implicitly recognizing that they are in fact separate worlds with separate rules.

Because we believe it is possible to maintain a single modelling effort and join both the interactive layer and the computational layer our proposal presents a set of good practices and a modelling process that combines UML diagrams to proper formalize the interactive layer description. This process can be extended to include the use of formal notations and tools such as prototyping and automatic validation. Although our students have formal methods background knowledge, that approach would lead to explore a completely, although also addressable, different way.

In this section we further explore how the existing UML diagrams can be used in the context of the approach we propose.

6.1 Notations

The three most relevant phases of interactive layer modelling are Requirements Analysis, Task Analysis & Design, and Dialogue Design. For each of these activities we propose a methodological approach to the usage of UML in order to capture the information needed to model the interactive layer.

We will now briefly describe the proposed modelling process:

1. Requirements analysis – Use Case diagrams are used to capture the requirements identified by the end- users. This is the adequate diagram to use at this stage given the fact that by definition a Use Case is an informal description of the functional requirements, the involved actors and the expected results.

   Usability requirements, as others non-functional requirements, may be included in the model as notes describing restrictions.

2. Task analysis & Design – at this stage, the information obtained in the requirements analysis phase is used in order to extract the necessary information to build the task specification.

   Approaches such as CTT formally capture the intended information. Using UML we can use the Activity Diagrams to obtain a equally information rich description.

   Since task modelling is not a concern of UML models, some of the activity diagrams that we obtain may sometimes become somewhat more complex and hard to read and interpret than models in domain specific languages for task modelling. Recent changes in UML brought new functionalities. One of these new functionalities makes it possible to draw interruptible regions – concerning flow control – allowing for the specification of cancellation operations. Looking at CTT capabilities we still lack diagrammatic constructors to describe the temporary suspension of a given dialogue and its later resume (in CTT that corresponds to the |> constructor).

3. Dialogue design – at this stage, and based on the information of the Use Case diagrams and the Activity diagrams, it becomes possible to identify the interaction points needed at the UI. Those interactions points are most of the times dialogue windows provided by the application to support the interactive dialogue. To express the control flow between (and within) the windows of the application, Statechart Diagrams are used. These diagrams represent a change in the analysis focus. At this point, the focus of analysis changes from the activities executed at the user interface to how the system should be built in order to support those activities.

   Describing the actions and the associated transitions also provides valuable information to the model. When creating the statechart diagram it becomes necessary to decorate the transitions with the corresponding methods from the business logic layer. This methodological step allows for the continuous and iterative refinement of the model until all the needed methods to implement the requirements are gathered.

   Being this process highly iterative while designing the statechart, it allows for the gathering of information about the software system. This allows that besides describing the interactive layer and the dialogue control the software engineers will also acquire information about the component architecture of the user interface.

So far, we have presented the necessary steps to model the user interface and we have shown how to methodologically accommodate this within the UML framework. However there are some pre-conditions that must be met in order to consider that this process is coherent. Namely:

- there is one single activity diagram for each use case discovered in the requirements gathering phase. Each activity diagram shall contain all the information necessary to describe the different scenarios a use case can be expanded to;
- there must exist a path in the statechart diagram for every flow that can be drawn in the activity diagrams. This assures that no information is lost between diagrams;
- the set of all methods discovered while decorating the statechart diagrams for a given entity, represents the observable behaviour of that entity. We can say that there is no behaviour associated to an entity that is not obtained through our process (excluding utility methods).

6.2 Refinement of the business logic

The process of designing the statechart diagrams makes visible all the business logic methods. Although it is not feasible to tell when this information is completely obtained since the process is clearly iterative, our proposal makes it possible to obtain the essential pieces of the business logic layer.

We definitely think this is also the natural process to discover business logic API's since the methods being discovered derive from the task analysis being conducted. In that sense while being task oriented the software engineer will obtain almost the complete set of needed methods to fully answer the requirements.

Of course there will always exist utility methods or even methods obtained from some refactoring process, but the majority of the API's from the interactive layer is gathered throughout our process.

7. AN EXAMPLE

To illustrate our approach we briefly present some diagrams that address the interactive layer modelling. The example below is taken from project work developed by students in the context of the course. The students were told to develop a generic task management system, a typical component of a Personal Information Management (PIM) system, and had to derive the requirements for the system during the practical and tutorial sessions and from scenarios that were subsequently provided.

More than twenty student teams carried out this particular project. Due to obvious space constraints we are unable to present a thorough description of the work carried out by any of the teams. Instead, for each of the stages described above we present examples of the diagrams obtained. With this eaxmples we hope to illustrate how UML can be used to reason about and to model the interactive layer. These diagrams illustrate that traditional software engineering techniques can be applied to model HCI aspects.

7.1 Requirements Analysis

In the requirements analysis stage we use the Use Case diagrams to identify the users of the system, their objectives and even usability related objectives.

Figure 2 presents a view of the use case model showing the high level use case diagram. The diagram identifies the types of users of the system (in this case only one), and the major functionalities it should provide. To each use case, a description, relevant scenarios, and relevant non- functional requirements can be attached. This feature was supported, in textual form, by the modelling tool in use.

To illustrate the following steps we will choose a given use case, "Change Calendar Task", and we present the following diagrams in that context.

7.2 Task Analysis & Design

Once the users' objectives identified it becomes necessary to describe the interaction between the entities that are part of the use case. We must create a task model that will allow

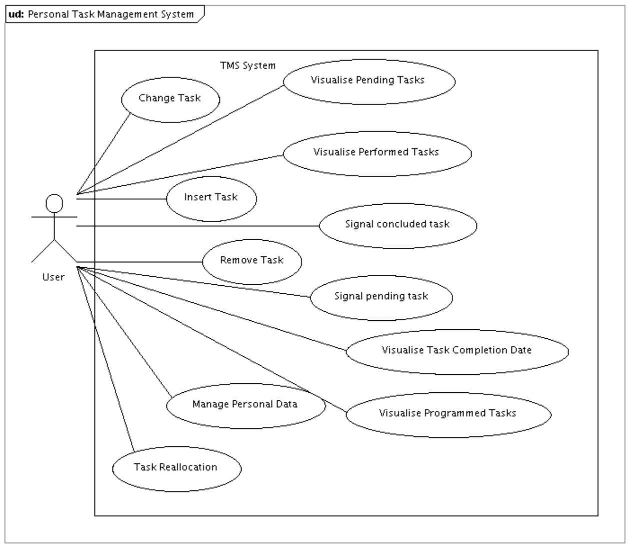

Figure 2: High level Use Case diagram

us to reason about the system and to guide the software development process. Using UML as our modelling language, activity diagrams are used to describe the behaviour that is specified in the corresponding use case (considering the different scenarios the use case may enclose).

Figure 3 presents a task model represented using an activity diagram. Partitions are used to assign activities to users and system. And interruptible regions to model dialogue cancelling. Notice that due to the intelligibility of the diagram it can be easily understood both by software engineering and HCI practitioners.

The level of bstraction used is similar to that of Canonical Abstract Prototypes. That is, interactions between users and device are kept at a high level of abstraction. Concrete instantiations of the users' actions are left to the next phase (Dialogue Design).

One possible drawback of using activity diagrams is that the hierarchical approach favoured by task modelling languages such as CTT is not as easily expressed. Nevertheless, activity diagrams can also be composed hierarchically.

7.3 Dialogue Design

Once the task model is established it becomes necessary to develop the corresponding UI. As stated before we will focus on the dialogue modelling. Assuming that we are dealing with an UI based on a windows management system it is necessary to identify the windows and to specify their dialogue control. As this stage we use statechart diagrams to describe dialogue control.

Figure 4 presents the statechart for the selected task. The overall model identifies not only the application windows but also the relevant situations around those windows. The dialogue control is modelled trough the transitions between states, which also helps the business layer refinement task.

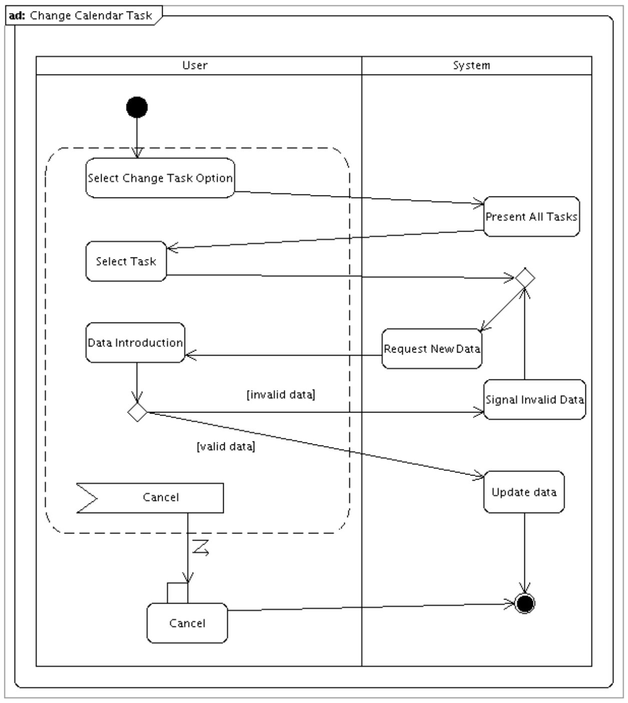

Figure 3: An Activity Diagram

Finally, Figure 5 shows an implementation of the system. The menu containing the options for changing the status of a task is presented. The transitions which are possible from the UI controls are all documented in the UML diagrams made in the analysis and design phase. Regardless of the look and feel of the application, the dialogue control was specified and developed accordingly.

The example given, although is not complex allowed us to briefly present the motivation and the envisaged method. More complex examples will decorate the model with HCI-specific, or related, annotations and will probably include scenarios and the inclusion of non-functional aspects. Our thesis is that this will not be disruptive with typical software engineering applications modeling.

As shown in the example of the positive gains of this approach is that anyone familiar with UML will be able to understand the model. In what concerns to the UML meta-modeling we did not introduce new constructors so what really happens is that UML is used under a different light. Nevertheless that does not compromise the comprehension of software engineers who do not have these HCI concerns in mind. They will find in the model information related to UI layer that they will understand, but to which they can pay more or less attention, depending on their focus.

The images that are presented above were taken from a project made by the students. We notice an improve in the overall quality of the projects, given the fact that the envisaged approach introduces new factors in the modeling phases. In a qualitative approach the projects clearly demonstrate that the students acquire a vision of what should be an information system by opposition to their previous ex-

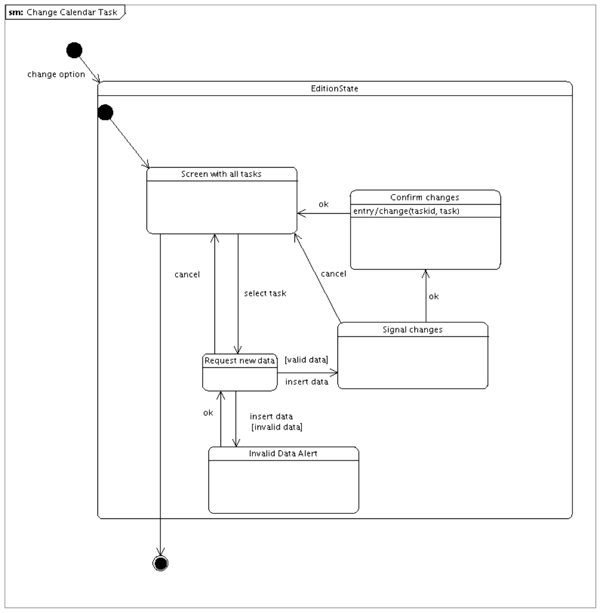

Figure 4: A statechart diagram

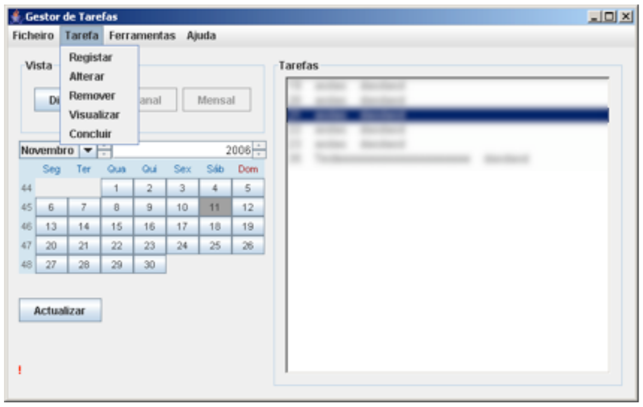

Figure 5: A screen from a project's result

periences in which they just used to model the business layer. The task management system, although simple, allowed us to introduce to the students that once an information system is a combination of a *human* and an *application*, it is also valuable to model this interaction, using state of the art languages and notations.

8. DISCUSSION AND CONCLUSIONS

Models are particularly good tools for communication and play an important role in a typical software engineering curricula. Models taught in typical software engineer courses are mainly focused in capturing and designing the business logic layer and usually neglect the UI layer. Our experience within a software engineering course showed that there is a need for promoting the communication between HCI and software engineering communities and techniques. We believe the proposed methodological process promotes a better communication between both layers and, by shortening the gap, allows students to acquire HCI concepts that are of great relevance.

The envisaged process does not add new constructors to the chosen modelling language, UML, ensuring that no language disruption needs to occur during the entire process. This was a main contraint from the ouset, since we wanted students to focus on the UML language.

Although we lack concrete statistical data it is still possible to detect some indicators that point to the success of the approach. Obtaining concrete hard statistical data would have been difficult because we specifically did not want to introduce the possibility of unequal treatment among students of the same class. Since the approach was developed and introduced incrementally and in parallel with other changes to the course, data from different years cannnot also be reliably compared. Nevertheless we were able to perceive an increase in the quality of the final result of the projects, both in terms of the user interfaces designed and built by the students, and in the number of teams that were able to hand in projects with a GUI component. We could also witness an augmented perception of the relevance of HCI topics among the students, and, interestingly, an improved usage of the UML was attained. Students could better understand the scope of each diagram and its role in development, and the previously asked question "where to put the user interface in my model?" was mostly eliminated.

Another advantage that the presented process has lies on the fact that the use of well grounded modelling techniques makes the software architecture of the interactive layer as robust, maintainable and reusable as the remaining components of the system. Regarding the three layers that applications should clearly have, namely Presentation Layer, Business Layer and Data Layer, our approach allowed us to reinforce their importance and even to clearly identify different modeling stages for each one of them.

Therefore we are convinced of the merits of our proposal namely regarding the add-ons it brings. It allows us as software engineers and HCI researchers to shorten the gap between the two areas and creates a methodological background to software engineering students. Usage of these techniques has proven very useful to the students, since it made possible the elaboration and validation of UI prototypes during the analysis and design phases. The students had a clear perception and understanding about how different a software project becomes when UI issues are, from the beginning brought into the general modelling and design concerns of the business layer, and how both layers may influence each other. Finally, the final product is one product indeed, and not two separate software products to be integrated at last.

Acknowledgements

The authors whish to thank the students of the Information Systems Development Course (2005/06), whose project works were used in this paper.

This work was partially carried out in the context of the IVY Project supported by FCT, the Portuguese Foundation for Science and Technology, and FEDER, the European regional development fund, under contract POSC/EIA/56646/2004.

9. REFERENCES

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, MA, 1998.

[2] Larry L. Constantine. Canonical abstract prototypes for abstract visual and interaction design. In J. Jorge, N.Nunes, and J. Falc?o e Cunha, editors, *Interactive Systems – Design, Specification, and Verification*, volume 2844 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 2003. (invited paper).

[3] Larry L. Constantine and Lucy A.D: Lockwood. *Object-Modeling and User Interface Design*, chapter Structure and Style in Use Cases for User Interface Design. Addison-Wesley, 2001.

[4] Alan Dix, Janet Finlay, Gregory D. Abowd, and Russel Beale. *Human-Computer Interaction.* Pearson Education Ltd., third edition edition, 2004.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[6] Jeremie Guiochet, Gilles Motet, Claude Baron, and Guy Boy. Toward a human-centered uml for risk analysis. In C.W. Johnson and P. Palanque, editors, *Proc. of the 18th IFIP World Computer Congress (WCC), Human Error, Safety and Systems Development (HESSD04)*, pages 177–191. Kluwer Academic Publisher, 2004.

[7] E. Hollnagel. *Human reliability analysis: context and control.* Academic Press, London, 1993.

[8] Ian Horrocks. *Constructing the User Interface with Statecharts.* Addison-Wesley, Harlow, England, 1999.

[9] IEEE Computer Society, Los Alamitos, California. *Guide to the Software Engineering Body of Knowledge: Trial Version*, May 2001.

[10] ISO. *ISO standard 13407 – Human-centered design processes for interactive systems.* International Organization for Standardization, first edition, June 1999.

[11] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Object Technology Series. Addison-Wesley, Reading, MA, 1999.

[12] Bonnie E. John, Len Bass, and Rob J. Adams. Communication across the HCI/SE divide: ISO 13407 and the Rational Unified Process. In *Proceedings of the 10th International Conference on Human Computer Interaction*, Crete, Greece, June 2003.

[13] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, August/September 1988.

[14] Leonel N?brega, Nuno Jardim Nunes, and Helder Coelho. Mapping concurtasktrees into uml 2.0. In Stephen W. Gilroy and Michael D. Harrison, editors, *Interactive Systems – Design Specification and Verification*, volume 3941 of *Lecture Notes in Computer Science*, pages 237–248. Springer-Verlag, 2006.

[15] William M. Newman and Michael G. Lamming. *Interactive System Design.* Addison-Wesley, 1995.

[16] N. J. Nunes and J. Falcão e Cunha. Wisdom — A UML based architecture for interactive systems. *Lecture Notes in Computer Science*, vol. 1946, 2001.

[17] F. Paternò. *Model Based Design and Evaluation of Interactive Applications.* Applied Computing. Springer Verlag, 1999.

[18] F. Paternò. ConcurTaskTrees and UML: how to marry them? Position paper at TUPIS'00 – a UML 2000 Workshop. York, UK, October 2000.

[19] P. P. Silva. *Object Modelling of Interactive Systems: The UMLi approach.* PhD thesis, Department of Computer Science - University of Manchester, 2002.