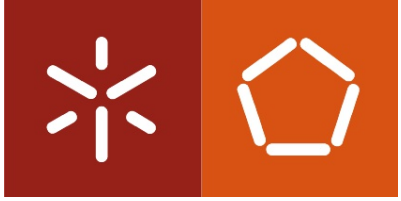
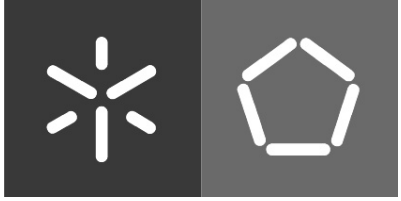


Universidade do Minho
Escola de Engenharia

João Pedro Cunha Gonçalves

Análise e desenvolvimento de extensões de controlo para o ambiente Scratch

Abril de 2012



Universidade do Minho
Escola de Engenharia

João Pedro Cunha Gonçalves

Análise e desenvolvimento de extensões de controlo para o ambiente Scratch

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação de
Prof. Doutor Fernando Mário Junqueira Martins

Abril de 2012

Declaração

Nome: João Pedro Cunha Gonçalves

Endereço Electrónico: pg15966@alunos.uminho.pt

Telefone: 253611966

Bilhete de Identidade: 13203329

Título da Tese: Análise e desenvolvimento de extensões de controlo para o ambiente Scratch

Orientador: Professor Doutor Fernando Mário Junqueira Martins

Ano de conclusão: 2012

Designação do Mestrado: Mestrado em Engenharia Informática

Universidade do Minho, 2 de Março de 2012

João Pedro Cunha Gonçalves

Agradecimentos

Gostaria de agradecer ao meu orientador, professor Mário Martins, pela orientação, motivação, aconselhamento e pelo tempo dispendido na melhoria desta dissertação.

Gostaria também de agradecer aos meus pais pelo apoio prestado ao longo desta jornada de trabalho.

Por fim, gostaria de agradecer aos meus colegas que me auxiliaram e com os quais também troquei ideias ao longo do trabalho.

Resumo

A Programação Visual permite a construção de programas usando imagens, ícones, entre outros elementos gráficos, em vez da usual escrita de texto. A linguagem Scratch é uma linguagem de programação visual lançada em 2007 pelo MIT Media Lab, que tem vindo a ser usada em contextos de ensino para crianças e em contextos de programação *lightweight*. O ambiente Scratch foi desenvolvido usando a linguagem de programação por objectos Squeak Smalltalk. A programação visual em Scratch baseia-se na metáfora do bloco de LEGO que se encaixa em blocos compatíveis para criar comportamentos, em geral animações. O sucesso deste ambiente de código aberto conduziu à necessidade de o rever e aumentar as suas capacidades, visando a sua eventual aplicação noutros contextos.

Este trabalho tem por objectivo criar um conjunto de extensões de controlo para a linguagem Scratch, quer pela criação de novos blocos básicos de programação, quer pela introdução de novas funcionalidades interactivas no ambiente, quer ainda pela criação de padrões reutilizáveis e adaptáveis a vários contextos. São também estudadas as formas de disponibilização de programas Scratch na *Web*, em particular as que já estão disponíveis no servidor de partilha de conteúdos, designado ScratchR.

Abstract

Visual Programming allows the construction of programs using pictures, icons, among other graphical elements, instead of typing text. Scratch is a visual programming language released in 2007 by MIT Media Lab, which has been used in teaching contexts for children and lightweight programming contexts. The Scratch environment was developed in Squeak Smalltalk, an object-oriented programming language. Visual programming in Scratch follows a similar approach to LEGO blocks, as it allows one to create behaviours, usually animations, by combining compatible blocks. The success of this open source environment led to the need to review and extend its capabilities, aiming its eventual application in other contexts.

This work aims to create a set of control extensions to the Scratch language, either by creating new programming blocks, by adding new interactive features to the environment, or by creating reusable patterns that can be adapted to various contexts. We also study ways to make Scratch programs available on the Web, in particular those that already exist in the ScratchR file sharing server.

Conteúdo

Lista de Acrónimos

AIF/AIFF	Audio Interchange File Format
API	Application Programming Interface
APV	Ambiente de Programação Visual
AU	Au File Format
BMP	Bitmap Image File
BYOB	Build Your Own Blocks
GIF	Graphics Interchange Format
GUI	Graphical User Interface
HSV	Hue, Saturation, Value
JPG	Joint Photographic Experts Group
LPV	Linguagem de Programação Visual
MIT	Massachusetts Institute of Technology
MP3	Moving Picture Experts Group Audio Layer III
MVC	Model–View–Controller
OLPC	One Laptop Per Child
PNG	Portable Network Graphics
PV	Programação Visual
RGB	Red, Green, Blue
SDU	Script Definido pelo Utilizador
UML	Unified Modeling Language
USB	Universal Serial Bus
WAV	Waveform Audio File Format
XML	Extensible Markup Language

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Contextualização

A Programação Visual permite a construção de programas usando imagens, ícones, entre outros elementos gráficos, em vez da usual escrita de texto. A linguagem de programação visual Scratch e o seu ambiente interactivo de desenvolvimento, programado usando a linguagem orientada pelos objectos Squeak Smalltalk, foram desenvolvidos pelo MIT Media Lab e lançados em 2007. A programação em Scratch consiste na combinação de blocos de diferentes formas (tal como os blocos LEGO), com o objectivo de criar comportamentos para uns objectos - *Sprites* - que actuam no **Palco**.

O Scratch baseia-se num paradigma "imagina-programa-partilha", onde os utilizadores dão asas à sua imaginação, programando os mais diversos tipos de projectos, podendo partilhá-los com outros utilizadores do Scratch. A programação com Scratch pretende ser simples, atractiva e intuitiva, para que qualquer tipo de utilizador se adapte facilmente a ela e a use para desenvolver soluções criativas para todo o tipo de problemas, actuando desta forma como agente activo na criação de conteúdos que, geralmente, tomam a forma de animações, simulações, histórias, etc. A componente de partilha, garantida por um portal *online* suportado na plataforma ScratchR, permite que os projectos Scratch possam lá ser depositados e expostos para toda a comunidade, que pode comentar, experimentar e até utilizar como base para novos projectos. Com um foco maioritariamente orientado para o ensino, o Scratch proporciona um ambiente de programação atractivo e simples para iniciação de crianças e jovens às tecnologias de informação, permitindo o desenvolvimento de competências na área da resolução de problemas por computador e o ensino de várias disciplinas de forma gráfica e interactiva.

A linguagem Scratch surge intimamente ligada ao projecto One Laptop Per Child (OLPC) do MIT, liderado pelo fundador do MIT Media Lab, Nicholas Negroponte, ao qual se associaram desde o início Seymour Papert (criador da linguagem Logo e pioneiro da Inteligência Artificial) e Alan Kay (um dos criadores da linguagem Smalltalk e um dos pioneiros da Programação Orientada pelos Objectos e dos ambientes interactivos baseados em janelas).

A nível nacional, em 2009, o Portal Sapo e a PT Inovação desenvolveram, em colaboração com o MIT, a primeira aplicação local do Scratch, adaptada para Português e para o computador Magalhães. É distribuída gratuitamente através do Portal Sapo (mais concretamente, no Sapo Kids), estando disponível para ser usada em todos os países de língua oficial portuguesa.

Com presença nos principais sistemas operativos (Windows, Linux e Mac OS), e com adaptações disponíveis para plataformas como os portáteis OLPC, Classmate e Magalhães, o Scratch conta ainda com projectos desenvolvidos por terceiros com vista a utilizá-lo em telemóveis e *tablets*.

1.2 Motivação

O enorme sucesso internacional deste ambiente de código aberto conduziu a expectativas da sua utilização noutros contextos mais amplos, ainda que tendo por base o mesmo paradigma, levando à necessidade de o rever e aumentar. Para além de considerações relativas à sua adaptação visando a sua utilização noutro tipo de terminais, coloca-se também a questão de aumentar as características do ambiente para que possa dar suporte ao ensino da programação para outros escalões etários. Em ambos os casos, ainda que usando técnicas diferentes (programação *lightweight*, por exemplo, no primeiro caso), a adaptação do Scratch a outros contextos envolve, necessariamente, o domínio completo da sua implementação pela exploração do seu código fonte, e o desenvolvimento e implementação de um conjunto de extensões de controlo, quer pela criação de novos blocos básicos de programação, quer pela introdução de novas funcionalidades interactivas no ambiente, quer ainda pela criação de padrões reutilizáveis e adaptáveis a vários contextos.

1.3 Objectivos

Dados o enquadramento e motivação supracitados, definiram-se os seguintes objectivos a alcançar neste trabalho:

- Estudo do código fonte das linguagens Squeak e Scratch e identificação completa da forma de implementação do ambiente gráfico, em especial os seus componentes principais: os blocos e os *Sprites*;
- Análise e identificação da forma de implementação do modelo de interactividade do ambiente Scratch, em especial no editor e no **Palco**, de forma a adquirir um domínio completo do mecanismo de mapeamento dos eventos interactivos do utilizador nos objectos animados;
- Desenvolvimento de novos controlos e novas funcionalidades interactivas associadas;
- Análise da possibilidade de criação de padrões de comportamento adequados a novos contextos de aplicação e que possam ser facilmente reutilizados;
- Criação de exemplos (e sua eventual disponibilização) baseados nas novas funcionalidades interactivas.

1.4 Contribuições

Este estudo traz para o público o resultado da análise detalhada do código fonte do Scratch, revelando a estruturação interna dos seus componentes, o modelo de interactividade existente e a forma como as principais funcionalidades estão implementadas. Ao mesmo tempo, as novas funcionalidades desenvolvidas sobem o tecto do Scratch, na medida em que introduzem a noção de encapsulamento de comportamentos, ou seja, *scripts* sob a forma de novos blocos. Estes blocos podem ser consultados, testados, modificados e eliminados. A isto acresce o mecanismo de exportação e importação desses blocos, que introduz a capacidade de partilha ao nível do *script* (até aqui apenas existia partilha ao nível do projecto e do *Sprite*), o que encaixa na vertente colaborativa que é um dos pontos-chave do Scratch. Esta componente destaca esta modificação do Scratch relativamente a outras já existentes.

1.5 Estrutura da dissertação

A apresentação do trabalho desenvolvido começa no Capítulo 2 com uma abordagem ao estado da arte relativo à área do conhecimento em que o Scratch se enquadra, a Programação Visual, bem como a exposição de ferramentas similares ao Scratch. No Capítulo 3 é feita uma descrição exaustiva das funcionalidades do ambiente e da linguagem de programação do Scratch, para que se tenha uma percepção das suas capacidades e limitações. A partir daqui, entra-se no estudo mais técnico do Scratch. O *framework* da interface com o utilizador, Morphic, é analisado no Capítulo 4. É a partir daqui que se percebe como funciona o modelo interactivo do Scratch. No Capítulo 5 expõe-se a estrutura interna do Scratch, apresentam-se as classes fundamentais que suportam os principais elementos da interface e mostra-se como foram implementadas as extensões do Scratch. Um caso de estudo é apresentado no Capítulo 6, mostrando a aplicabilidade das novas funcionalidades. Finalmente, são apresentadas as conclusões e o trabalho futuro no Capítulo 7.

Capítulo 2

Estado da arte

Neste capítulo enquadra-se o Scratch no contexto da Programação Visual, fazendo-se uma exposição do estado da arte relativo a esta temática, com incidência na sua aplicação em contextos de ensino. Termina-se com uma apresentação do Scratch e de projectos dele derivados.

2.1 Introdução

A História do Homem mostra que sempre existiu comunicação baseada em imagens e símbolos (pinturas rupestres, hieróglifos, pinturas artísticas, etc). Essas imagens possuíam um significado (semântica), que permitia transmitir uma ideia/mensagem, ou seja, formavam uma linguagem visual que servia como meio de comunicação. "Uma linguagem visual é uma representação pictórica de entidades conceptuais e operações e é, essencialmente, uma ferramenta através da qual os utilizadores compõem frases visuais" [Zha07]. Este tipo de comunicação pode ser considerado mais simples e com um alcance maior quando comparado com uma comunicação de base textual. O ser humano pensa numa circunferência através da sua representação gráfica e não pela sua representação textual que diz que "uma circunferência é o lugar geométrico de todos pontos de um plano equidistantes de um ponto fixo considerado como o centro da circunferência". A expressão deste conceito segundo a primeira forma torna-o perceptível para qualquer ser humano, enquanto que a segunda obriga, não só a um esforço adicional de concretização da ideia sobre a forma textual, mas também a que todo o processo comunicativo e seus intervenientes recorram das regras sintáticas e semânticas da linguagem utilizada. Da mesma forma, uma criança consegue transmitir uma ideia muito mais facilmente de forma visual do que de forma textual. Através de um processo imagético de comunicação, há a tradução directa das ideias, raciocínios e do processo criativo para uma linguagem universal, acelerando e facilitando o processo de comunicação, e tornando-o acessível a uma maior audiência. É com base nesta premissa que surge a investigação e desenvolvimento na área da Programação Visual (PV). Esta pretende ser uma forma de facilitar o processo de comunicação entre homem e computador, via programação com recurso a elementos pictóricos, abrindo assim as portas de um mundo de potencialidades onde a expressão directa da criatividade humana resulta num trabalho realizado com menor esforço e que cumpre a sua missão na transmissão de ideias. O Scratch surge, neste contexto, como um caso de estudo interessante neste domínio. Sendo uma linguagem de programação visual, fornece formas gráficas através das quais os programas ganham vida, tudo feito dentro de um ambiente de fácil utilização, atractivo, interactivo e leve. Prova disso é a sua adopção em projectos como o One Laptop Per Child (OLPC) e a sua orientação focada, maioritariamente, num público mais jovem que pretenda iniciar-se nas tecnologias de informação e desenvolver as suas competências

na área de resolução de problemas por computador.

Compreender a programação confere a capacidade de resolver problemas e de perceber, até certo ponto, os mecanismos que sustentam as ferramentas informáticas utilizadas pelas pessoas no quotidiano. No entanto, a programação é difícil e exige tempo e dedicação para ser dominada. É aqui que a programação visual tem o seu ponto forte: o facto de permitir a pessoas pouco ou nada experientes em programação construir programas através da simples ligação de componentes, simplificando o processo de programação. Isto não significa que a PV seja orientada exclusivamente para utilizadores com poucos conhecimentos no domínio da programação. Prova disso é que existem linguagens, como o Prograph, orientadas a programadores profissionais [Bur99]. Apesar de a PV trazer muita facilidade à programação, continua a ser necessário conhecer alguns princípios que a regem e aprender a usar o ambiente de desenvolvimento, conhecendo as suas vantagens e desvantagens.

2.2 Programação Visual

Estando o Scratch inserido no contexto da Programação Visual, faz-se nesta secção uma contextualização histórica da mesma, assim como uma apresentação dos conceitos intrínsecos à Programação Visual, os seus objectivos, classificação, vantagens, desvantagens e casos de uso actuais.

2.2.1 Enquadramento histórico

O conceito de programação visual não é recente, tendo crescido ao longo dos tempos, bebendo influências de áreas como a Interacção Humano-Computador e a Computação Gráfica. O sistema Sketchpad, desenvolvido por Ivan Sutherland em 1963 no computador TX-2 do MIT, é considerado o primeiro sistema de computação gráfica [Sut63, BD04]. Este permitia, através da utilização de uma caneta óptica, a criação de gráficos 2D como linhas e círculos, permitindo que sobre eles fossem aplicadas operações e definidas restrições sobre as formas geométricas, com o auxílio de um conjunto de botões [BD04]. Este sistema, apesar do seu contributo, não é considerado um sistema de programação visual [Mye90]. Um dos trabalhos mais antigos nesta área remonta a 1966, quando William Robert Sutherland, irmão de Ivan Sutherland, desenvolveu uma linguagem gráfica experimental para o computador TX-2, no MIT Lincoln Laboratory, por forma a descrever procedimentos de forma gráfica, através da disposição e ligação de elementos pictóricos [Sut66]. Outro ponto de referência no que toca à programação visual é o ambiente/linguagem Pygmalion (1975) [Smi75], implementado em Smalltalk, orientado para pessoas da área das Ciências da Computação, que introduziu o conceito de Programação por Demonstração/Exemplo (o utilizador define o comportamento de um procedimento baseando-se num exemplo concreto e, a partir daí, o sistema generaliza o procedimento para qualquer situação análoga) e o paradigma da programação baseada em ícones: as entidades gráficas que controlam a execução do programa, que podem ser criadas, modificadas e ligadas e que possuem semântica e dados a si associados. "Pygmalion é a origem do conceito de ícones tal como conhecemos hoje nas interfaces gráficas com o utilizador presentes nos computadores pessoais." [CHK+93] Dadas as limitações tecnológicas da altura, era impossível programar algoritmos complexos (e.g., não havia memória suficiente para calcular o factorial de 4).

2.2.2 Definição

"Programação Visual refere-se a qualquer sistema que permite especificar um programa usando expressões com duas ou mais dimensões" [Mye90, RSB05], chamadas de expressões visuais, que

incluem gráficos, desenhos, animações, ícones, entre outros. Existem dois conceitos distintos que devem ser clarificados: Linguagem de Programação Visual (LPV) e Ambiente de Programação Visual (APV). O primeiro refere-se a uma linguagem de programação dotada de semântica e cuja sintaxe inclui expressões visuais [Bur99]. O segundo refere-se a um ambiente onde é possível manipular e combinar expressões visuais de acordo com um conjunto de regras (gramática) que definem o modo como se podem construir programas à sua custa. Em alguma literatura [Bur99, BS94] o termo APV é associado aos ambientes de programação visual para linguagens textuais tradicionais (e.g., Visual Studio, Netbeans). LPV apresentam várias formas gráficas para a construção de programas: combinação de blocos, caixas ou outras entidades gráficas ligadas por setas ou linhas (muito ao estilo de um diagrama de actividades em UML). De ressalvar que as LPV não devem ser confundidas com a família de linguagens Microsoft Visual (Basic, C#, C++, etc), uma vez que estas últimas são linguagens textuais que são complementadas com um *kit* de construção de interfaces gráficas com o utilizador (GUI) por forma a facilitar o processo de desenvolvimento de aplicações.

2.2.3 Objectivos

A investigação na área das LPV tem-se focado em atingir determinados objectivos [Bur99]:

- Tentar melhorar o desenho das linguagens de programação, uma vez que as menores restrições sintácticas conferem mais liberdade para explorar novos mecanismos de programação;
- Facilitar a programação a um grupo específico de pessoas;
- Melhorar a correcção com que os programas são desenvolvidos;
- Diminuir o tempo que as pessoas demoram a programar.

Não têm como principal objectivo eliminar o texto do processo de programação pois muitas delas utilizam-no em determinados contextos.

2.2.4 Classificação

As LPV podem ser de carácter geral ou possuir um domínio específico de aplicação. Podem ser classificadas em categorias que não são mutuamente exclusivas, o que significa que uma linguagem pode pertencer a mais do que uma categoria [BD04]:

- Linguagens puramente visuais: esta é a categoria mais importante. Nestas linguagens, todo o processo de programação é baseado em técnicas visuais, isto é, os programas são compostos por representações gráficas e são compilados directamente a partir da sua representação visual sem recorrer a uma linguagem textual intermédia, sendo também executados e feito o seu *debugging* no mesmo ambiente visual. Podem ainda ser subdivididas em ícónicas, não icónicas, orientadas a objectos, funcionais ou imperativas. VIPR, Prograph e PICT são exemplos deste tipo de LPV;
- Sistemas híbridos textuais e visuais: combinam elementos textuais e visuais. Incluem dois subtipos: sistemas em que os programas são criados visualmente, sendo depois traduzidos para uma linguagem textual de alto nível (e.g., Rehearsal World [CHK+93]), e sistemas que permitem a utilização de elementos gráficos em linguagens textuais (e.g., intercalar código C/C++ com diagramas);

- Sistemas que seguem o paradigma da Programação por Exemplo: linguagens que generalizam um procedimento a partir da demonstração do seu comportamento para um caso concreto (e.g., Pygmalion [CHK+93], Rehearsal World);
- Sistemas orientados às restrições: permitem modelar objectos no ambiente visual e sujeitá-los a restrições. São utilizados no desenho de simulações e no desenvolvimento de GUI. E.g., Thinglab, ARK;
- Sistemas baseados em *forms*: vão buscar inspiração às folhas de cálculo, usando as células para representar dados e as fórmulas para representar computações. Os programas consistem num conjunto de células interligadas (*forms*) que mudam de estado ao longo do tempo, à medida que os programas executam. E.g., Forms/3.

2.2.5 Vantagens e desvantagens

De seguida apresentam-se as vantagens e desvantagens das LPV [Beg96].

Vantagens:

- Fornecem pistas visuais para programadores inexperientes e mais jovens: ligações entre objectos são mais facilmente identificáveis bem como os parâmetros de funções;
- Eliminam a necessidade de lidar com sintaxes complexas das linguagens textuais tradicionais;
- Permitem uma melhor visualização do fluxo de execução de um programa (vantagens claras no que toca ao paralelismo);
- Maior facilidade em descrever processos da vida real;
- Partilhar um programa torna-se mais fácil ao defini-lo como um bloco (efeito apenas alcançado nas linguagens textuais que seguem o paradigma da orientação a objectos);
- Mais fácil de perceber o que um programa faz se este for representado por uma figura do que se for representado por várias linhas de código dispersas por vários ficheiros.

Desvantagens:

- Programadores experientes acusam maior facilidade em expressar certas ideias via texto do que usando formas gráficas;
- "Limite de Deutsch", expressão cunhada por Fred Lakin, após uma intervenção de Peter Deutsch numa apresentação sobre o tema da Programação Visual feita por Scott Kim e Warren Robinett, na qual disse: "Bem, isto parece bom, mas o problema das linguagens de programação visual é que não se consegue ter mais de 50 primitivas visuais no ecrã ao mesmo tempo. Como é que se vai escrever um sistema operativo?". Esta citação revela uma desvantagem das LPV face às linguagens textuais, que é a grande densidade de informação que o formato texto consegue conter. As expressões visuais necessitam de maior espaço no ecrã para poderem ser identificadas ou conter uma etiqueta textual. Esta limitação é então declarada da seguinte forma: "O problema da programação visual é que não se consegue ter mais de 50 primitivas visuais no ecrã ao mesmo tempo.". Esta limitação pode, no entanto, ser ultrapassada recorrendo-se à modularização dos programas (quando existir suporte para tal), tal como nas linguagens textuais, o que promove a reutilização e aumenta a densidade de informação por elemento visual.

2.2.6 Casos de uso actuais

A Programação Visual tem vários casos de uso nos dias de hoje [BS94]:

- LabVIEW: orientado para cientistas e engenheiros, tem aplicações na área de aquisição de dados laboratoriais, processamento de imagens, controlo de qualidade, etc;
- Prograph: simulação 3D e *rendering*, software de análise do mercado financeiro, sistemas de gestão de contactos, etc;
- Google App Inventor para o Android: ferramenta *web* que permite a pessoas que não sejam programadoras criar aplicações para o sistema operativo Android. Está construído sobre o OpenBlocks, um *framework* escrito em Java que permite a construção de "sistemas de programação de blocos gráficos através da especificação de um único ficheiro XML" [Roq07]. A programação visual no OpenBlocks e o Scratch têm várias semelhanças. Para além do desenvolvimento de aplicações orientadas ao mercado *smartphone*, tem ainda grandes potencialidades ao nível do ensino e educação. Neste aspecto podem-se imaginar aplicações que permitam fotografar e catalogar espécies animais e vegetais, que permitam o ensino de conceitos elementares de Matemática, questionários para auxílio ao estudo, etc.

2.3 Programação Visual como um mecanismo de ensino

Num contexto educativo, as LPV podem ser interessantes ferramentas de auxílio ao ensino, não só de programação, mas também de outras áreas como a Matemática ou as Ciências Naturais. A Programação Visual é apelativa para um público jovem uma vez que evita a aprendizagem de uma sintaxe complexa, a maioria dos ambientes de desenvolvimento são interactivos e permitem a manipulação de imagens ou blocos. Estudos realizados [SL08] mostram que os jovens consideram divertido aprender conceitos de Ciências da Computação usando LPV, mudando favoravelmente a sua opinião em relação a esta área do conhecimento após a experiência.

2.3.1 Ensino de programação

No que toca ao ensino de programação e de conceitos de Ciências da Computação, são analisados três sistemas: LogoBlocks, Puck e Alice.

LogoBlocks. A linguagem Logo foi desenvolvida pela equipa de Seymour Papert no MIT, nos anos 60, com um objectivo maioritariamente educacional. O processo de programação com a linguagem Logo beneficia de um mecanismo visual, conhecido por Turtle Graphics, que permite visualizar a execução de um programa. Esse mecanismo toma a forma de um cursor representado por um triângulo ou uma tartaruga que desenha uma dada geometria de acordo com os comandos fornecidos, constituindo, desta forma, um auxílio ao ensino da programação a crianças. BrickLogo é uma extensão da linguagem Logo que é normalmente utilizada para programar o Programmable Brick, um pequeno computador de mão que controla até 4 motores e consegue ler valores de 6 sensores. O Programmable Brick pode ser usado para controlar carros ou robôs LEGO que estejam equipados com motores e sensores. LogoBlocks [Beg96] é a versão visual da linguagem BrickLogo. O ambiente do LogoBlocks permite a criação de programas através da combinação de blocos de diferentes formas e cores, os quais estão disponíveis numa palete de blocos. Os programas podem ser posteriormente descarregados para o Programmable Brick. É, no entanto, necessário salientar que, embora o LogoBlocks forneça um ambiente de iniciação à programação simples e divertido para crianças, ele é bastante limitado a nível de funcionalidades,

não fornecendo mecanismos fundamentais na área da programação como estruturas condicionais, argumentos e valores de retorno nas funções, entre outros, pelo que serve apenas como estímulo à iniciação da aprendizagem da programação, que poderá ser realizada com o auxílio de outros ambientes e linguagens mais completos.

Puck. Esta LPV foi desenvolvida na Friedrich Schiller University, Jena, Alemanha, para ser usada nas escolas [Koh07]. É orientada para o ensino das Ciências da Computação, ou seja, para um público iniciante no mundo da programação. Quando foi criada tinha como requisitos a possibilidade de ensinar conceitos como variáveis, tipos de dados, estruturas de controlo, funções com parâmetros e ainda permitir a geração de pseudo-código e código em linguagens textuais, como Java. Este último requisito tinha em mente permitir uma fácil transição do mundo da programação visual para o da programação textual. Evita os erros de sintaxe através da utilização de um sistema visual, libertando assim os seus utilizadores do trabalho de lidar com as mensagens de erros de sintaxe. O Puck segue a abordagem de combinação de blocos (representativos de diversos tipos de instruções como *input*, *output*, som, atribuição, desenho de linhas, estruturas condicionais, ciclos, invocação de funções, etc) e inclui um método de medição da complexidade dos programas, útil para comparar diferentes soluções para o mesmo problema. Os professores podem usar o pseudo-código gerado a partir de um programa como base para um exercício de escrita do mesmo na linguagem Puck. Possui um mecanismo de registo do histórico das actividades do utilizador, permitindo aos professores acompanhar o progresso dos seus alunos. Possui uma caixa de comentários que pode ser usada pelos professores para fornecer instruções aos alunos ou pode ser usada por estes últimos para expressar as suas ideias ou dificuldades. No entanto, tem algumas limitações como o facto de só se poder usar os blocos predefinidos e de apenas existirem os tipos de dados para representar valores inteiros e booleanos.

Alice. Alice [CDP00] é uma linguagem educacional desenvolvida em Python na Universidade de Carnegie Mellon que é utilizada na criação de animações, jogos interactivos e vídeos num ambiente 3-D. É orientada a objectos, sendo útil para introduzir conceitos como estruturas de controlo, objectos, classes, métodos, atributos, herança, tratamento de eventos, etc [Koh07]. Segue também o estilo *drag&drop* de blocos correspondentes às instruções existentes nas linguagens de programação como Java, C++ e C#, evitando os erros de sintaxe. Fornece *feedback* visual imediato permitindo ao utilizador perceber a relação entre as instruções e o comportamento dos objectos nas animações. Um facto interessante é que pretende captar a atenção do público feminino para o mundo da programação através da sua faceta de criação de histórias com animações.

2.3.2 Ensino de outras temáticas

As LPV também podem ser usadas para ensinar outros assuntos que não os de Ciências da Computação. O Squeak Etoys [Kay05] é um exemplo prático disso. O Etoys foi escrito em Squeak, uma linguagem orientada a objectos que é analisada na Secção 2.4.2, e foi fortemente influenciado pela linguagem Logo. Influenciou o desenvolvimento do Scratch e também está integrado no projecto OLPC. É um sistema de programação visual que permite criar projectos com base em texto, imagens 2D ou 3D, som, música e fotos, complementados com *scripts* que definem o comportamento destes elementos. É uma ferramenta de auxílio ao ensino que fomenta a participação dos alunos nos projectos, fornecendo *feedback* imediato. O ambiente capta a atenção do público mais jovem, constituindo uma forma eficaz de ensinar, por exemplo, Matemática, Ciências e Artes. O facto de permitir apresentar representações visuais de conceitos abstractos de Matemática ou Gramática facilita o processo de aprendizagem.

2.4 Scratch

2.4.1 Perspectiva geral

Scratch é uma LPV híbrida (os seus elementos visuais são traduzidos em código Squeak) e *lightweight* (leve para o sistema) que capta a atenção do utilizador pelo facto de possibilitar a programação de jogos, animações, simulações, histórias, etc, ao invés dos típicos algoritmos de ordenação de listas, entre outros. Actualmente na versão 1.4, Scratch é também um APV que permite a criação de projectos interactivos. Oferece a possibilidade de importar imagens e sons, ou utilizar o editor de imagens e o gravador de som para criá-los, e permite interacção com o utilizador via teclado. Este ambiente de programação visual permite o empilhamento de blocos com diferentes comportamentos e formas que combinam entre si (mediante determinadas regras) tal como os blocos LEGO, criando *scripts*. Esses blocos, manipulados via *drag&drop*, possuem etiquetas que apresentam os comandos dos programas. Mais uma vez, a sintaxe não constitui problema porque está "visualmente evidente e codificada nas formas dos blocos" [Roq07]. O Scratch fornece *feedback* imediato e perceptível, sendo possível acompanhar os efeitos das operações nos dados. O Scratch permite a criação de *Sprites* - objectos 2D com estado (variáveis) e comportamento (*scripts*) - e cada *Sprite* é controlado pelo seu conjunto de *scripts*. Os *Sprites* existem e movimentam-se numa área designada de **Palco** e a aparência de cada um é definida pelo seu conjunto de trajes. A Tabela 2.1 apresenta um resumo destas entidades fundamentais do Scratch.

Entidade	**Ilustração**
Blocos	
Script	
Sprite	
Palco	 

Tabela 2.1: Entidades fundamentais do Scratch.

O ambiente Scratch permite gravar projectos em ficheiro. A sua interface pode ser descrita como sendo formada por uma única janela com vários painéis. Na palete de blocos encontram-

se todos os blocos existentes separados por 8 grupos de funcionalidades (categorias), que estão identificados por cores. Nestas categorias encontram-se blocos para movimentação dos *Sprites*, alteração da aparência dos *Sprites* e efeitos visuais, tocar sons, desenhar, blocos de instruções de controlo dos *scripts* (início, fim, ciclos, estruturas condicionais, etc), detecção de eventos, operações matemáticas e lógicas e de criação e manipulação de variáveis e listas.

Estes blocos, que possuem diferentes formas dependendo do tipo de instrução que representam, podem então ser arrastados e combinados na área de *scripting* por forma a gerar *scripts* (Figura 2.1).

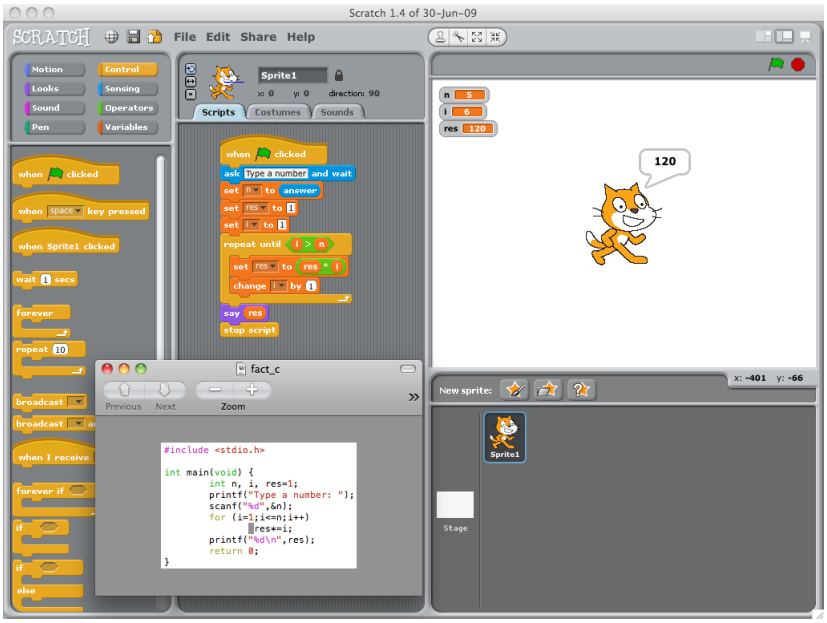


Figura 2.1: Ambiente do Scratch mostrando um *script* que calcula o factorial de um número inteiro, sendo o resultado apresentado pelo *Sprite* no **Palco**. Por baixo do *script* encontra-se o mesmo código escrito em C, permitindo ver a ligação entre os blocos e as respectivas instruções numa linguagem textual.

O Scratch não suporta mecanismos de programação orientada a objectos como classes e herança. Como tal, não pode ser considerada uma linguagem orientada a objectos, apesar de ser baseada neles. No que toca a limitações, enfrenta o problema do Limite de Deutsch (ver 2.2.5). Ao nível dos tipos de dados, embora os números, booleanos e *strings* sejam tipos de primeira classe, as listas não o são. Um tipo é dito de primeira classe se os valores desse tipo reunirem cinco condições: 1) podem ser atribuídos a variáveis; 2) podem ser passados como parâmetro a uma função; 3) podem ser devolvidos por uma função; 4) podem ser constituintes de um tipo mais complexo e 5) podem ser anónimos [MH11]. As mensagens apenas podem ser enviadas em *broadcast* (para todos os *Sprites*) e não a *Sprites* individuais, e os *scripts* por elas activados não podem ter argumentos nem devolver valores. Não possui mecanismos de controlo de concorrência como semáforos, monitores ou *locks*. O seu modelo de concorrência evita a maioria das condições de corrida e permite que o utilizador raciocine sobre um *script* de forma isolada dos outros, no entanto ainda possui falhas tais como um evento activar *scripts* numa ordem diferente da esperada. Uma perspectiva mais aprofundada sobre o ambiente e a programação com Scratch é fornecida no Capítulo 3.

A linguagem Scratch foi lançada em 2007 pelo MIT Media Lab e tem vindo a ser utilizada em contextos de ensino para crianças e em contextos de programação *lightweight*. Tem por base um paradigma designado "imagina-programa-partilha" e tem como finalidade proporcionar um ambiente atractivo e simples para o ensino e aprendizagem das tecnologias de informação e programação por parte de crianças e jovens, sendo focada numa aprendizagem activa através da experimentação [UCK+10]. É uma linguagem que se baseia nas ideias construcionistas do Logo e do Etoys (a pessoa obtém conhecimento construindo algo que seja do seu interesse, com recurso ao computador) [MRR+10]. Ambientes como o LogoBlocks e Alice também influenciaram a concepção do Scratch [MBK+04]. Scratch foca-se num grupo etário compreendido entre os 8 e os 16 anos [UCK+10], como se pode ver na Figura 2.2 [Köl10].

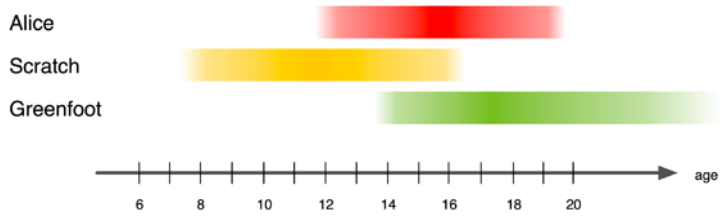


Figura 2.2: Faixa etária do Scratch.

Esta linguagem está ligada ao projecto OLPC, contando com a contribuição de figuras de renome na área das linguagens de progamação orientadas a ensino (Seymour Papert) e na área da programação orientada a objectos e dos ambientes interactivos baseados em janelas (Alan Kay).

Na visão de Mitchel Resnick [Res07, Res02], líder da equipa de desenvolvimento do Scratch, depois da Sociedade da Informação e da Sociedade do Conhecimento, encontrámo-nos agora numa Sociedade Criativa em que, mais do que possuir conhecimento, é necessário usá-lo para desenvolver soluções criativas para problemas inesperados. As novas tecnologias podem e devem ser usadas para fomentar e desenvolver a capacidade de pensar e agir de forma criativa, sendo esta a chave para o sucesso nesta sociedade. Defende ainda que, nesta sociedade, se devem redefinir as abordagens à educação e aprendizagem por forma a estimular a criatividade desde a infância [Res02] e que a aprendizagem recreativa deve ser feita ao longo da vida para "manter viva a criança que há dentro de nós" [Res03]. Critica o facto das novas tecnologias continuarem a ser adoptadas como um meio de reforçar a utilização de métodos de ensino obsoletos ao invés de introduzirem mudança. Como tal, afirma que é necessário repensar as abordagens ao ensino e educação e estudar de que forma as novas tecnologias podem suportar essas abordagens. Exemplifica o caso dos computadores, que ainda são vistos como um meio de transmitir informação aos alunos e não como um meio de potenciar as suas capacidades de desenho e criação.

Na sociedade existe uma falta de "fluência digital", isto é, as pessoas sabem usar as ferramentas tecnológicas, mas não sabem tirar partido delas para construir algo de relevante. Os jovens, considerados nativos digitais graças à facilidade que demonstram na utilização da tecnologia, têm um perfil mais consumidor do que criador, "é como se soubessem "ler" mas não soubessem "escrever"" [RMMH+09]. A programação apresenta-se como um meio de alterar este panorama, promovendo, ao mesmo tempo, o desenvolvimento do pensamento computacional (capacidade de resolver problemas usando os fundamentos das Ciências da Computação). Há, também, uma necessidade de desenvolver novas tecnologias, diferentes dos computadores tradicionais, que instiguem mais as camadas jovens à sua exploração.

De entre várias iniciativas desenvolvidas por forma a ultrapassar o problema da falta de "fluência digital" encontram-se as Computer Clubhouses, uma rede de centros tecnológicos comunitários de aprendizagem pós-aulas para jovens provenientes de comunidades desfavorecidas, criada numa parceria do The Computer Museum com o MIT Media Laboratory (responsável pelo Scratch). Estes centros, mais do que fornecerem as competências básicas de trabalho com computadores, ajudam os jovens a aprender a desenhar, criar e inventar usando as novas tecnologias. O desenvolvimento do Scratch foi orientado segundo as necessidades e restrições das Computer Clubhouses e dos seus membros. O Scratch foi então introduzido nestes centros por forma a criar uma cultura de programação que não existia, tentando assim desenvolver o pensamento abstracto que é fundamental para resolver todo o tipo de problemas (para além do campo das tecnologias da informação) [MBK+04].

As tentativas anteriores de introdução da programação aos jovens, com a linguagem Logo e outras, não foram bem sucedidas devido à utilização de linguagens difíceis de usar, à falta de ligação das actividades desenvolvidas com os interesses dos jovens e ao facto de não haver acompanhamento por parte de um mentor especializado [MBK+04, RMMH+09].

Resnick *et al.* [RMMH+09] referem que na visão de Seymour Papert, inventor da linguagem Logo, as características ideias de uma linguagem de programação são: 1) "chão baixo", ou seja, fáceis de iniciar; 2) "tecto elevado", fornecendo a oportunidade de criar projectos cada vez mais complexos ao longo do tempo; 3) "paredes amplas" por forma a suportar diferentes tipos de projectos, para que pessoas com diferentes interesses e formas de aprender possam sentir empatia com a linguagem. No desenvolvimento do Scratch pretendeu-se baixar ainda mais o chão e ampliar ainda mais as paredes relativamente a linguagens mais recentes (sejam elas de uso profissional, como o Flash/ActionScript, ou orientadas aos mais jovens, como o Alice e o Squeak Etoys), mantendo suporte ao desenvolvimento do pensamento computacional. Por forma a atingir estes objectivos, a equipa de desenvolvimento estabeleceu 3 princípios essenciais de desenho do Scratch: 1) dotá-lo de uma maior liberdade para inventar e experimentar; 2) torná-lo mais significativo; 3) torná-lo mais social do que outros ambientes de programação. Quanto ao primeiro princípio, a equipa de desenvolvimento pretendeu transportar para o Scratch todo o processo de brincar, construir e o evoluir de ideias que ocorre quando as crianças brincam com LEGOs, sendo tal ponto constatado através da interface do Scratch, com a sua facilidade e intuitividade na construção de programas. O segundo princípio diz respeito ao facto de se pretender que as pessoas usem o Scratch para criar projectos do seu interesse. Para tal, a equipa de desenvolvimento preocupou-se com 2 critérios importantes: diversidade (suporte a diversos tipos de projectos que abranjam diversos tipos de pessoas) e personalização (facilidade de personalização dos projectos). O terceiro princípio tem a ver com a componente "partilha" do Scratch: a constituição de uma comunidade de utilizadores do Scratch que dispõe de um portal *online*, suportado pela plataforma ScratchR, onde podem partilhar os seus projectos bem como comentar, analisar e aprender com os projectos de outros. Esta plataforma possui um carácter de rede social e tem como objectivo fomentar a aprendizagem criativa assim como a partilha e colaboração entre os utilizadores, uma vez que muitos dos projectos realizados resultam da chamada "apropriação criativa", isto é, da utilização das ideias de outrem para a criação de um novo projecto. [MH07, MHR08]. Há ainda uma vertente de colaboração muito importante que permite a criação de equipas de utilizadores que se complementam nas suas competências. Ou seja, existe um verdadeiro ecossistema em torno do Scratch.

O Scratch vem assim responder aos novos desafios propostos pela Sociedade Criativa do século XXI, fomentando nos estudantes a capacidade de "pensar de forma criativa, planear sistematicamente, analisar de forma crítica, trabalhar colaborativamente, comunicar de forma clara, desenhar iterativamente e aprender continuamente" [Res07].

O sucesso da utilização do Scratch estende-se desde os níveis de ensino mais básicos até ao En-

sino Superior. Nas Computer Clubhouses, o Scratch é muitas vezes preferido em relação a outros programas de criação de *media* ou actividades de entretenimento pelos jovens [MRR+08]. Em muitos casos, os jovens associam o Scratch a um dado tema escolar (Música, Artes, Matemática) mas não o associam à programação [MPK+08]. Malan e Leitner [ML07] referem o potencial do Scratch no Ensino Superior, enquanto linguagem introdutória da programação, uma vez que permite que os estudantes se foquem na lógica e nas construções programáticas em vez da sintaxe. Já em linguagens como o Java os alunos são obrigados a dominar a sintaxe antes de resolverem os problemas, daí que o Scratch deva ser visto como uma porta de entrada para essas linguagens textuais. A experiência de utilização do Scratch durante um curso de Verão em Harvard, com o objectivo não de melhorar os resultados ao nível da programação mas antes melhorar a experiência do primeiro contacto com a programação, mostra que o Scratch entusiasmou os alunos e, para aqueles que não tinham experiência em programação, permitiu-lhes familiarizar com os fundamentos da programação, sem os distrair com problemas de sintaxe. Os estudantes apontam como aspecto positivo o facto de o Scratch ser divertido, fácil de aprender e apresentar ao utilizador resultados imediatamente visíveis. Da experiência, salientam o facto do Scratch lhes ter permitido ganhar o tipo de raciocínio necessário para implementar programas simples. Outro exemplo é a utilização do Scratch no Harvard College como tema introdutório às Ciências da Computação (*Introduction to Computer Science I*).

2.4.2 Squeak Smalltalk

O ambiente de código aberto do Scratch foi desenvolvido em Squeak [BDNP07], uma implementação de código aberto do Smalltalk-80. Em Smalltalk tudo é um objecto, o que faz dela uma linguagem orientada a objectos pura. É baseada na troca de mensagens entre objectos, é interactiva (é possível escrever uma instrução e, logo de seguida, avaliá-la e ver o seu resultado), é uma linguagem totalmente suportada pelo mecanismo de Reflexão, pelo que está implementada nela própria, possui *garbage collector*, é dinamicamente tipada e possui um sistema forte de tipos, é extensível e multiplataforma. Utiliza o Morphic como *framework* da interface com o utilizador, substituindo a arquitectura *Model-View-Controller* (MVC) original do Smalltalk [Mal01]. Squeak possui uma máquina virtual desenvolvida também em Squeak, onde executa. Neste ambiente, é possível fazer modificações a objectos que já estejam criados e observar essas mudanças em tempo real. Também tem suporte a internacionalização e a compilação incremental. Tem aplicações ao nível do desenvolvimento *web*, educação (Etoys), jogos, multimédia (gráficos 2D e 3D, áudio, vídeo), entre outros.

2.5 Projectos paralelos

Dadas as limitações do Scratch, surgiram alguns projectos paralelos ao mesmo que oferecem mais funcionalidades com alguma relevância. De entre estes projectos destacam-se o Build Your Own Blocks (BYOB), o Chirp e o Panther.

2.5.1 Build Your Own Blocks (BYOB)

O BYOB é uma extensão do ambiente Scratch que permite, tal como o nome indica, construir blocos personalizados com base em blocos predefinidos do Scratch ou outros blocos definidos pelo utilizador. Os principais mecanismos que introduziu foram [Mön08]:

- Definição de procedimentos e funções;
- Passagem de parâmetros;
- Variáveis locais de funções e procedimentos;
- Recursividade;
- Atomicidade.

Tenta ultrapassar o Limite de Deutsch através da construção de programas baseados em funções (modularização) (Figura 2.3).

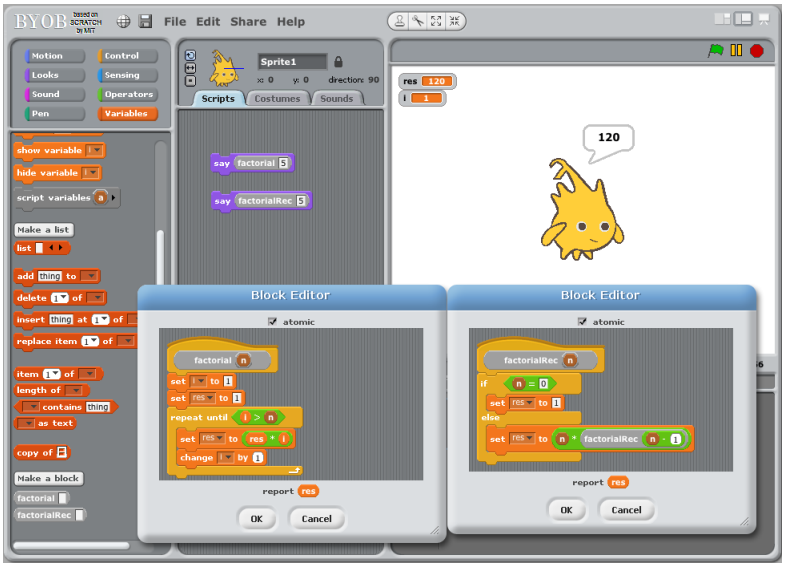


Figura 2.3: Definição de blocos no BYOB que calculam o factorial de um número inteiro, quer recursiva quer iterativamente.

Actualmente está na versão 3.1.1 e já recebeu muitas novas funcionalidades. Algumas delas são influência do Scheme, um dialecto de LISP, outras advêm do facto de que "um princípio fundamental do desenho do BYOB é que todos os dados devem ser de primeira classe" [Mön09a, MH11, HM10]:

- Duplicação de blocos personalizados entre *Sprites* via *drag&drop*;
- Funcionalidades de *debugging* que assinalam blocos personalizados com erros e introdução de blocos de *debugging*;
- Possibilidade de criar *Sprites* compostos por *Subsprites* (noção de composição), com um nível de aninhamento infinito, sendo que os *Subsprites* seguem o comportamento dos seus *Supersprites* (no entanto, também podem apresentar comportamento próprio);
- Possibilidade de partilhar *Sprites* normais e aninhados numa rede *Mesh* (permite interacção entre múltiplos projectos Scratch através de um mecanismo de memória partilhada ou troca de mensagens);

- Incluído um compilador que permite converter um projecto Scratch/BYOB num ficheiro executável (`.exe`);
- Editor de blocos redimensionável, melhorias ao nível do *drag&drop*, *scrolling* automático, *scrolling* ao arrastar, e "desfazer";
- GUI para inspeccionar o código Squeak dos blocos predefinidos do Scratch: projecto Elements [Mön09b]. Este projecto segue o mesmo paradigma de programação com blocos e foi desenvolvido com o objectivo de verificar se a forma como o Scratch foi pensado e desenhado se adequa, não só a ambientes educacionais, mas também a ambientes profissionais com grande suporte no paradigma da orientação a objectos;
- Possibilidade de criar funções de ordem superior;
- Expressões lambda e blocos personalizados anónimos, ou seja, funções de primeira classe;
- Listas de primeira classe, incluindo listas dinâmicas anónimas, listas de listas (que permitem a construção de estruturas de dados como árvores binárias, tabelas de dispersão, *heaps*) e listas de *scripts* e blocos;
- Procedimentos/funções locais e globais;
- *Sprites* de primeira classe;
- Blocos e *scripts* de primeira classe;
- Permite o envio de mensagens a um *Sprite* específico;
- Introduz um modelo de programação com *Sprites* baseado na programação orientada a objectos baseada em protótipos, em que um *Sprite* serve como modelo (protótipo) para todos os *Sprites* clonados a partir dele (relação pai-filhos). Os clones herdam propriedades dos seus pais (estado e comportamento), podendo ser partilhadas entre pai e filho ou não. O mecanismo de delegação de métodos de filhos para pais funciona de forma semelhante à procura dinâmica de métodos na hierarquia de classes numa linguagem orientada a objectos baseada em classes;
- Permite a criação explícita de classes e instâncias e do mecanismo de herança, ainda que de forma diferente do habitual;
- Introdução de blocos para os valores booleanos Verdadeiro e Falso, de verificação de tipos, manipulação dos códigos ASCII dos caracteres, de realização cópias *shallow* de listas, entre outros.

Tal como referido por [HM10], o BYOB resulta de um esforço de adicionar ao Scratch conceitos que o tornem adequado para utilizar em disciplinas introdutórias das Ciências da Computação, sem ser necessário criar duas versões diferentes da linguagem (uma para crianças e outra para utilizadores mais avançados).

2.5.2 Chirp

O projecto Chirp é uma versão ligeiramente modificada do Scratch 1.2.1, que adiciona algumas funcionalidades como exportação/importação de *scripts* como ficheiros XML, trocas de blocos através de um menu de contexto, *scrolling* na janela do ambiente, painel de edição de *scripts* redimensionável, instalador para Windows que permite abrir um projecto directamente em modo

de apresentação, possibilidade de utilizar projectos Scratch como protecção de ecrã do Windows e de distribuir projectos Scratch como ficheiros executáveis (`.exe`). Sendo este projecto da mesma autoria do BYOB, funcionalidades como a geração de executáveis foram integradas nesse projecto, no qual o autor tem vindo a colocar todo o seu esforço, estando o desenvolvimento do Chirp parado desde 2009.

2.5.3 Panther

O Panther é outra extensão do Scratch, desenvolvida a pensar nos utilizadores já experientes em Scratch. Esta extensão acrescenta novos blocos nas categorias já existentes que introduzem a possibilidade de clonar e eliminar *Sprites*, novas operações matemáticas, operações de manipulação de *strings*, etc. Possui duas novas categorias de blocos para manipulação de ficheiros externos e cores. À semelhança do BYOB, dá ao utilizador a possibilidade de criar os seus próprios blocos, numa funcionalidade designada de CYOB: *Code your own blocks*. A diferença é que esta funcionalidade foge ao paradigma da programação visual uma vez que o utilizador define o corpo do bloco através de código textual Squeak escrito numa caixa de texto (Figura 2.4), o que obriga a ter um conhecimento razoável desta linguagem.

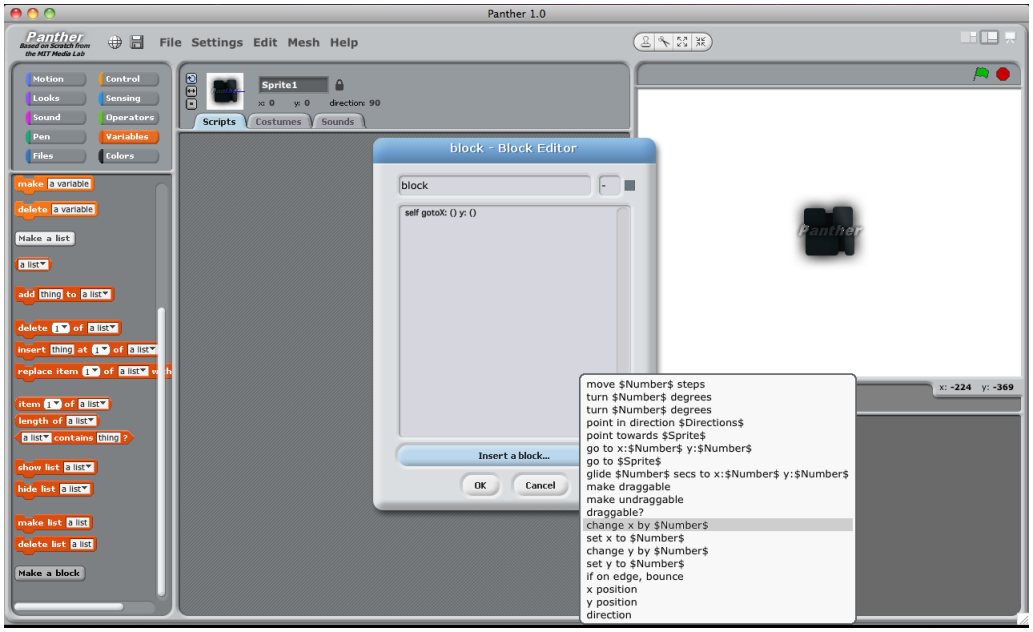


Figura 2.4: Ambiente de criação de blocos no Panther.

2.6 Conclusão

Neste capítulo apresentou-se a Programação Visual como sendo o produto de longos anos de investigação e de sinergias de áreas como a Interacção Humano-Computador e a Computação Gráfica, pretendendo facilitar o processo de transmissão de ideias do homem para o computador utilizando a programação. Fugindo ao tradicional debitar de linhas de texto, esta abordagem baseada em expressões visuais permite trazer o mundo da programação para um público menos experiente na área, tendo como vantagens a maior intuitividade e a despreocupação com a

aprendizagem de regras sintáticas. Como foi possível verificar, a programação visual aplicada a contextos de ensino bebe influências dos vários anos de investigação e desenvolvimento da área da programação orientada ao ensino com linguagens como o Logo. Neste âmbito oferece potencialidades no ensino de programação, quer em cursos introdutórios às Ciências da Computação, quer em simples experiências que as crianças realizam com robôs. Para o ensino de Matemática e outras ciências, ferramentas como o Etoys mostram-se úteis. O Scratch serve como caso de estudo da Programação Visual, tendo sido apresentado como uma linguagem que permite criar facilmente histórias interactivas, jogos e animações, com a possibilidade de partilhá-las com outros utilizadores na *web*. Este sistema possui limitações, algumas delas ultrapassadas em projectos derivados como o Build Your Own Blocks (BYOB).

Capítulo 3

Scratch: ambiente e linguagem de programação

Neste capítulo apresenta-se em detalhe o ambiente de programação Scratch. São apresentados os vários componentes da sua interface e descritas as suas funcionalidades. Analisam-se, também, os principais princípios de programação com Scratch e alguns elementos característicos da sua linguagem.

3.1 Elementos da interface

O paradigma base do Scratch é que o utilizador define comportamento para uns objectos programáveis, denominados de *Sprites*, que habitam numa área chamada de **Palco** (o palco da acção). Este comportamento é definido juntando peças correspondentes a instruções numa linguagem de programação comum, encaixando-as conforme a sua forma o permita, à semelhança de blocos LEGO, dando origem a *scripts*.

O ambiente de programação do Scratch está dividido em 4 painéis principais que coexistem numa única janela (Figura 3.1).

No painel da esquerda encontra-se a palete de blocos, onde estão disponíveis todos os 125 blocos do Scratch, divididos por categorias que possuem diferentes cores (Figura 3.2). Estas categorias agrupam conjuntos de blocos com funções relacionadas (Figura 3.3). São 8 categorias no total:

- **Motion**: blocos relacionados com a movimentação dos *Sprites*. Não existem blocos nesta categoria quando o **Palco** está seleccionado. Dentro desta categoria é possível ter ainda acesso aos blocos de motores, que permitem programar um motor ligado ao computador, podendo ser usados, por exemplo, com os motores LEGO Education WeDo Robotics Kit;
- **Looks**: blocos que alteram a aparência dos *Sprites* e do **Palco**;
- **Sound**: blocos que trabalham com os sons;
- **Pen**: blocos que permitem que os *Sprites* desenhem no **Palco**;
- **Control**: blocos que controlam o fluxo de execução dos *scripts*;
- **Sensing**: blocos que fornecem informação relativamente às teclas premidas, valores do rato, temporizador, volume do som, etc;

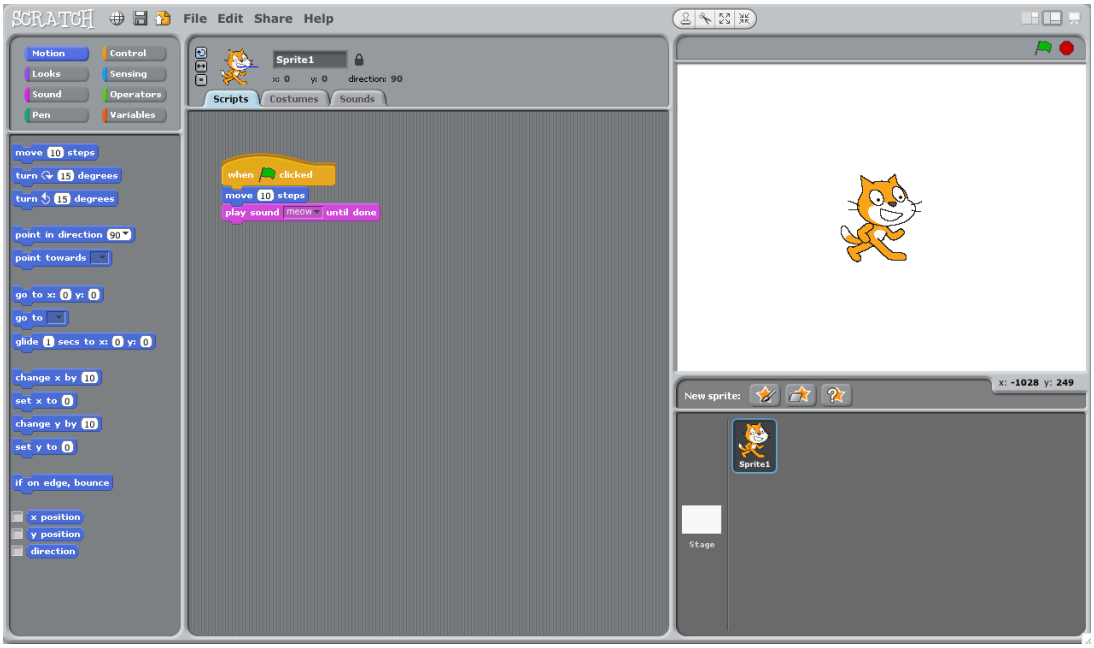


Figura 3.1: Interface do Scratch.

- **Operators**: blocos que realizam operações matemáticas;
- **Variables**: esta categoria tem 2 botões, um para criar variáveis e outro para criar listas. Quando é criada uma variável surgem nesta categoria os blocos que permitem utilizá-la, bem como um botão para eliminar a variável. O mesmo sucede para as listas.

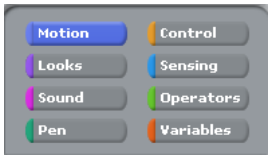


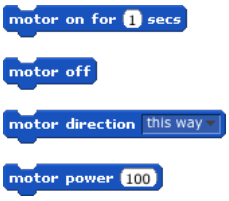
Figura 3.2: Categorias de blocos do Scratch.

Quando os blocos estão na palete, têm disponível um menu contextual que apresenta uma única opção: **help**. Esta opção permite visualizar um ecrã de ajuda: uma pequena imagem que explica a funcionalidade do bloco em questão, com o auxílio de exemplos.

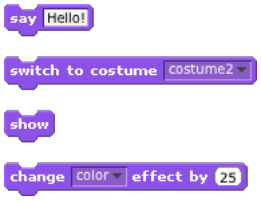
No painel central, o conteúdo é definido em função de uma de três abas que o utilizador pode abrir: **Scripts**, **Costumes** e **Sounds**. Seleccionando a aba **Scripts**, o utilizador tem acesso a uma área onde pode construir os seus programas, encaixando blocos provenientes da palete, como se pode ver na Figura 3.4.

O menu contextual desta área dá acesso a três opções: **clean up**, **save picture of scripts** e **add comment**. A primeira permite alinhar todos os *scripts* à esquerda, organizando-os de forma a não se sobreporem. A segunda permite tirar uma "fotografia" do estado actual e disposição dos *scripts* presentes nesta área, criando um ficheiro no formato GIF. A terceira permite criar uma caixa amarela onde se pode deixar um comentário (Figura 3.5). Esta caixa

(a) Exemplos de blocos da categoria **Motion**.

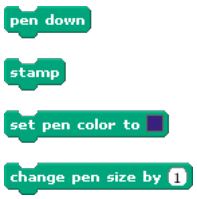


(b) Exemplos de blocos de motores.



(c) Exemplos de blocos da categoria **Looks**.

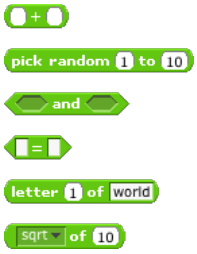
(d) Exemplos de blocos da categoria **Sound**.



(e) Exemplos de blocos da categoria **Pen**.

(f) Exemplos de blocos da categoria **Control**.

(g) Exemplos de blocos da categoria **Sensing**.



(h) Exemplos de blocos da categoria **Operators**.

(i) Exemplos de blocos de variáveis da categoria **Variables**.

(j) Exemplos de blocos de listas da categoria **Variables**.

Figura 3.3: Exemplos de blocos existentes nas diversas categorias.

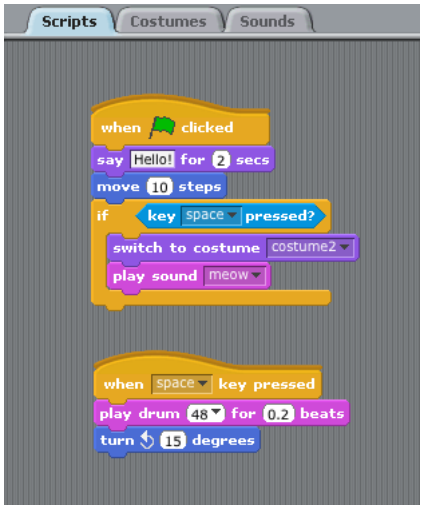


Figura 3.4: Aba **Scripts** do painel central.

pode ser redimensionada e expandida/colapsada. Um comentário pode ser anexado a um bloco arrastando-o para cima dele, sendo criada uma conexão entre ambos. Para quebrar esta conexão basta arrastar o comentário, afastando-o do bloco.

Figura 3.5: Comentário anexado a um bloco.

A aba **Costumes** apresenta os diversos "trajes" associados a um *Sprite*, que lhe conferem a aparência. Na Figura 3.6 é possível ver que existem 2 trajes para o *Sprite* em questão, estando o primeiro seleccionado.

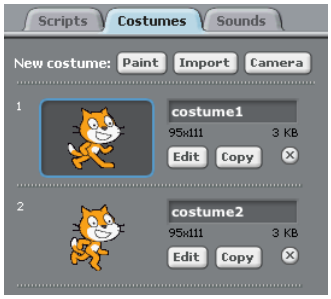


Figura 3.6: Aba **Costumes** do painel central.

Para um dado traje é possível alterar o seu nome, editá-lo através do editor de imagens integrado do Scratch (Figura 3.7), criar uma cópia sua ou eliminá-lo do conjunto de trajes de um *Sprite*. Através do seu menu contextual acede-se às opções **turn into new sprite** e **export**

this costume, que permitem criar um novo *Sprite* a partir do traje e exportar uma cópia do mesmo para um ficheiro no formato GIF ou BMP, respectivamente. É também possível alterar a ordem dos trajes.

Para criar um novo traje, pode-se optar por uma de 4 opções:

- Usar o editor de imagens;
- Importar um ficheiro de imagem no formato JPG, BMP, PNG ou GIF. No caso de ser um ficheiro GIF animado, cada uma das *frames* da animação é importada como um traje separado;
- Tirar uma foto a partir da *webcam*;
- Arrastar e largar um ficheiro de imagem no ambiente Scratch.

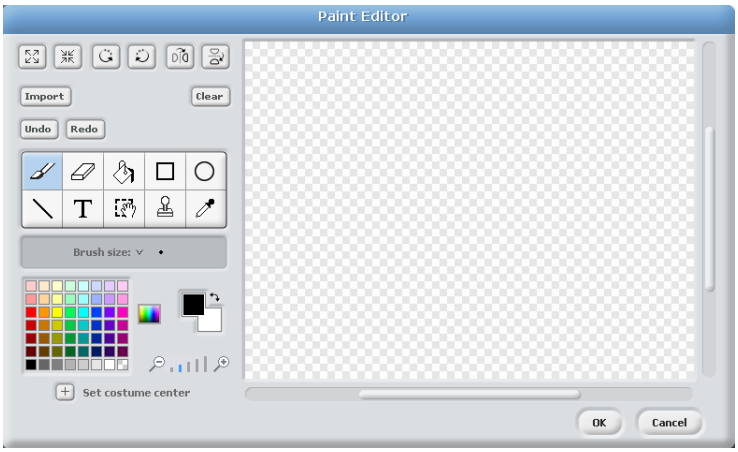


Figura 3.7: Editor de imagens integrado do Scratch.

Na aba **Sounds**, o utilizador tem acesso ao conjunto de sons associados a um *Sprite* (Figura 3.8). É possível ouvir um dado som, alterar o seu nome, exportá-lo para um ficheiro no formato WAV (através da opção **export this sound** do menu contextual) ou eliminá-lo do conjunto de sons do *Sprite*. É também possível gravar um som usando o gravador integrado (Figura 3.9) ou importar um som gravado num ficheiro no formato AIF, AIFF, AU, WAV ou MP3. É ainda possível importar um som, arrastando e largando um ficheiro de aúdio no ambiente Scratch.

Figura 3.8: Aba **Sounds** do painel central.

À direita, o painel superior apresenta o **Palco**, o local onde se visualiza o efeito dos programas desenvolvidos, isto é, onde toma lugar a acção (Figura 3.10). É aqui que os *Sprites* coexistem e interagem entre si.

Figura 3.9: Gravador de sons integrado do Scratch.

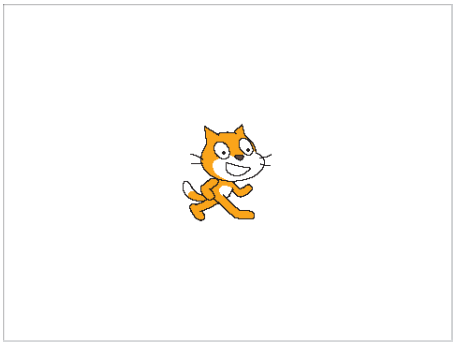


Figura 3.10: **Palco**.

O **Palco** é uma área rectangular de 480 unidades de largura e 360 de altura, sendo o seu centro a origem do referencial cartesiano que define os valores das coordenadas `x` e `y` do cursor do rato, apresentadas por baixo do mesmo (Figura 3.11).

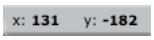


Figura 3.11: Coordenadas x e y do rato.

O menu contextual do **Palco** fornece duas opções: **grab screen region for new sprite** e **save picture of stage**. A primeira possibilita a criação de um novo *Sprite* cujo traje consiste na região do **Palco** seleccionada. A segunda guarda num ficheiro de imagem uma "fotografia" do estado actual do **Palco**. Os *Sprites*, quando visíveis no **Palco**, possuem também um menu contextual com várias opções:

- **grab screen region for new costume**: permite criar um traje para o *Sprite* a partir de uma área seleccionada do **Palco**;
- **export this sprite**: permite exportar um *Sprite*, e respectiva informação associada, para um ficheiro de extensão `.sprite`;
- **duplicate**: permite criar uma cópia do *Sprite*;
- **delete**: elimina o *Sprite*;
- **resize this sprite**: permite redimensionar o *Sprite*;
- **rotate this sprite**: permite rodar o *Sprite*.

O painel do canto inferior direito apresenta a lista dos *Sprites* do projecto na forma de miniaturas, destacando o *Sprite* que está seleccionado (Figura 3.12). Cada *Sprite* está identificado pelo seu nome.

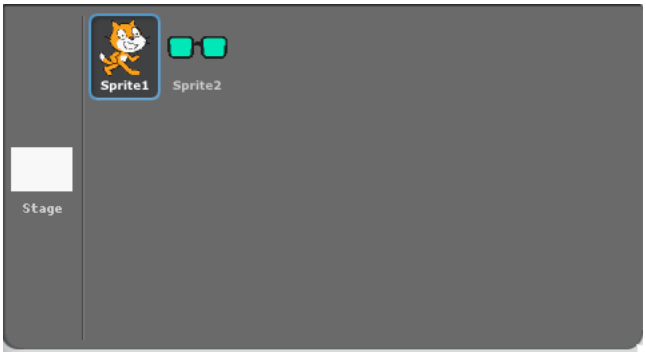


Figura 3.12: Miniaturas do **Palco** e dos *Sprites*.

Seleccionando um deles (clicando na sua miniatura ou fazendo duplo clique sobre o mesmo no **Palco**), é alterado o conteúdo das abas do painel central, para apresentar os *scripts*, trajes e sons associados ao *Sprite* seleccionado. É possível alterar a disposição dos *Sprites* nesta lista arrastando as suas miniaturas. O menu contextual associado a cada miniatura fornece quatro opções: **show**, **export this sprite**, **duplicate** e **delete**. A primeira permite mostrar um *Sprite* que esteja para lá dos limites do **Palco** ou que esteja escondido por outros *Sprites*. As outras três têm a mesma funcionalidade das opções homólogas existentes para os *Sprites* no **Palco**, atrás referidas.

Juntamente com a lista de *Sprites*, este painel apresenta ainda uma miniatura do **Palco**. Da mesma forma que a selecção de um *Sprite* faz alterar o conteúdo mostrado no painel central, também aqui, seleccionando a miniatura do **Palco** ou fazendo duplo clique sobre o próprio **Palco**, esse mesmo conteúdo se altera. A única diferença é que a aba **Costumes** altera o seu nome para **Backgrounds**, uma vez que diz respeito aos diferentes panos de fundo de toda a acção.

Por cima da lista de *Sprites* existem três botões dedicados à criação de novos *Sprites* (Figura 3.13).

Figura 3.13: Botões de criação de *Sprites*.

O primeiro permite abrir o editor de imagens do Scratch por forma a pintar um traje, que será atribuído ao novo *Sprite*. O segundo permite importar um *Sprite* a partir de um ficheiro. O ficheiro pode ser relativo a um *Sprite* previamente exportado (tendo a extensão `.sprite`) ou pode ser um ficheiro de imagem num dos formatos referidos para a criação de trajes, sendo que neste caso é criado um novo *Sprite* tendo como traje a imagem importada. O terceiro e último botão introduz um carácter aleatório à acção de importação de um *Sprite* a partir de um ficheiro de imagem, ou seja, escolhe um ficheiro de imagem à sorte e adiciona-o como novo *Sprite*.

Na parte superior do painel central encontra-se a informação relativa ao *Sprite* seleccionado (Figura 3.14).

Figura 3.14: Informação relativa ao *Sprite* seleccionado.

De entre o conjunto de informações apresentadas destaca-se o nome do *Sprite* (editável), as suas coordenadas em valores de `x` e `y`, relativamente ao centro do **Palco**, e a direcção em que aponta. A direcção possui quatro valores principais: 0, 90, 180 e -90, para as direcções cima, direita, baixo e esquerda, respectivamente. Por omissão, um *Sprite* aponta para a direita. Utilizando a linha azul que sai do centro do *Sprite*, o utilizador pode definir a sua direcção. O cadeado localizado ao lado direito do nome do *Sprite*, quando aberto via clique do utilizador, permite que o *Sprite* seja arrastado no modo de apresentação. O rectângulo colorido do lado direito do cadeado indica a cor seleccionada para os desenhos feitos pelos *Sprites*. Do lado esquerdo da imagem do *Sprite* encontram-se três botões que permitem definir o estilo de rotação do *Sprite*, isto é, de que forma o seu traje se comporta aquando de mudanças de direcção. O botão de cima, quando seleccionado, permite que o traje rode à medida que o *Sprite* muda de direcção. O botão do meio permite que o traje apenas se vire para a esquerda ou direita. O botão de baixo faz com que o traje fique sempre estático, mesmo quando o *Sprite* muda de direcção.

Imediatamente acima do **Palco**, do lado direito, situam-se 2 botões destinados à execução e paragem de *scripts*: uma bandeira verde e um botão vermelho em forma de STOP (Figura 3.15).

Figura 3.15: Botões de execução de *scripts*.

A bandeira verde, quando accionada, faz iniciar todos os *scripts* cujo bloco de topo é o bloco da categoria **Control** associado à bandeira verde (Figura 3.16).

Figura 3.16: Bloco associado à bandeira verde.

Enquanto os *scripts* executam, a bandeira fica destacada. A execução dos *scripts* pode ser interrompida accionando o botão vermelho.

Na barra superior do ambiente Scratch, na zona situada acima do canto superior esquerdo do **Palco**, encontra-se uma barra de ferramentas que permitem realizar determinadas acções sobre determinados objectos. Como se pode ver na Figura 3.17, existem 4 ferramentas.

Figura 3.17: Barra de ferramentas.

A primeira, **Duplicate**, permite duplicar *Sprites*, trajes, sons, blocos ou *scripts*. A segunda, **Delete**, permite apagar estes mesmos elementos. A terceira, **Grow sprite**, permite aumentar o tamanho de um *Sprite*. A quarta, **Shrink sprite**, faz o inverso, ou seja, diminui o seu tamanho.

Já do lado direito, acima dos botões de execução de *scripts*, existem 3 ícones que permitem ao utilizador aceder a diferentes modos de utilização do Scratch (Figura 3.18). O modo com o qual o Scratch inicia por omissão é o modo de edição e é representado pelo ícone do meio. Neste modo, o **Palco** ocupa uma maior área do ambiente, tirando espaço ao painel central. O ícone da esquerda, por seu turno, permite tornar o **Palco** mais pequeno, expandindo a área reservada ao painel central. Este modo tem vantagens no que toca ao espaço disponibilizado para a criação de *scripts* e facilita a utilização do Scratch em ecrãs de pequenas dimensões. O ícone da direita faz com que o Scratch entre em modo de apresentação. Entrando neste modo, o **Palco** passa a ocupar a totalidade do ecrã, podendo o utilizador apresentar o seu projecto.

Figura 3.18: Modos de utilização do Scratch.

No canto superior esquerdo do ambiente existe uma barra de menus (Figura 3.19).

Figura 3.19: Barra de menus do Scratch.

O ícone em forma de globo permite aceder a uma lista de cerca de 50 idiomas para os quais o Scratch possui tradução, podendo o utilizador trocar de linguagem dinamicamente, o que contribui para a vertente colaborativa multicultural do Scratch [MBK+04]. Os dois ícones seguintes permitem gravar um projecto e enviá-lo para a página *Web* do Scratch, respectivamente. Para além destes, existem os menus **File**, **Edit**, **Share** e **Help**. O menu **File** permite aceder às seguintes opções:

- **New**: permite criar um novo projecto;
- **Open**: permite abrir um projecto;
- **Save** e **Save As**: permitem gravar um projecto num ficheiro de extensão `.sb`;
- **Import project**: permite fundir um dado projecto com o projecto actual, importando os *Sprites* e os panos de fundo do **Palco** desse projecto;
- **Export sprite**: à semelhança das opções de exportação de *Sprites* já analisadas, também esta permite exportar o *Sprite* seleccionado para um ficheiro de extensão `.sprite`;
- **Project Notes**: permite associar notas escritas a um projecto, tais como questões ou instruções;
- **Quit**: permite encerrar o Scratch.

O menu **Edit** possui as seguintes opções:

- **Undelete**: permite recuperar o último elemento apagado (bloco, *script*, *Sprite*, traje ou som);
- **Start Single Stepping**: permite ver o programa a executar passo a passo, sendo cada bloco destacado quando está a executar;
- **Set Single Stepping**: apresenta um menu de opções para escolher a velocidade da execução passo a passo;
- **Compress Sounds** e **Compress Images**: comprimem (podendo haver perda de qualidade) os sons e as imagens do projecto por forma a diminuir o tamanho em disco do mesmo;
- **Show Motor Blocks**: adiciona os blocos dos motores à categoria **Motion**.

O menu **Share** permite enviar o projecto para a página *Web* do Scratch e fornece um atalho para aceder directamente à página de entrada na *Web* do Scratch. Uma vez *online*, os projectos ficam disponíveis para todo o mundo, podendo outros utilizadores comentar o projecto ou até descarregá-lo para estudar o seu código fonte ou desenvolver em cima dele. O menu **Help** possui as opções:

- **Help Page**: permite aceder a material de auxílio à utilização do Scratch, como guias, perguntas frequentes, etc;
- **Help Screens**: abre uma página com todos os ecrãs de ajuda dos diferentes blocos do Scratch;
- **About Scratch**: abre uma pequena janela com alguma informação sobre o Scratch.

3.2 Programação em Scratch

Os principais componentes de um projecto Scratch são os objectos chamados *Sprites*, para os quais o utilizador define comportamento através da combinação de blocos que representam instruções. Os blocos, provenientes da palete, são arrastados pelo utilizador para a área de *scripting* da aba **Scripts**, sendo encaixados uns nos outros, formando empilhamentos (*stacks*). As diferentes formas dos blocos sugerem de que forma estes podem ser encaixados. Quando o utilizador os está a tentar encaixar nalguma *stack* existente, surge uma marca branca que identifica os locais susceptíveis de encaixe para o bloco em questão (Figura 3.20). A ausência dessa ajuda visual indica que o bloco não tem ponto de encaixe na localização em questão. Estas características fazem com que o encaixe de blocos resulte sempre na construção de programas que estão sintaticamente correctos e que fazem sentido. A questão da algoritmia fica, contudo, a cargo do utilizador.

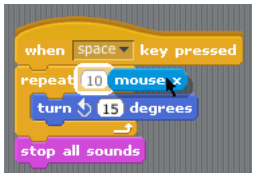


Figura 3.20: Ajuda visual mostra ao utilizador os locais onde pode encaixar o bloco.

O utilizador, ao clicar num bloco de uma *stack*, faz com que ela execute cada um dos seus blocos sequencialmente, começando no topo. Para eliminar um bloco, o utilizador pode arrastá-lo de volta para a palete. Pode também optar por usar a opção **delete** do menu contextual do bloco. Este menu também fornece a opção **help** que mostra o ecrã de ajuda do bloco, e também a opção **duplicate** que permite criar uma cópia do bloco. Quando o utilizador arrasta o bloco de topo de uma *stack*, toda ela é arrastada. Se, no entanto, arrastar um bloco que esteja no meio da *stack*, apenas a *stack* formada por esse bloco e pelos blocos que estão encaixados por baixo é arrastada. O utilizador pode copiar uma *stack* de blocos de um *Sprite* para outro arrastando-a para cima da miniatura do *Sprite* presente na lista de *Sprites*.

Alguns blocos possuem argumentos, que tanto podem ser números ou *strings* passíveis de serem introduzidos em campos de texto editáveis como podem ser menus a partir dos quais se pode escolher uma opção (nome de um som, de uma variável, etc) ou selectores de cores. Esses argumentos têm valores por omissão que ajudam o utilizador a perceber qual a sua funcionalidade. As *strings* são avaliadas com o valor 0 quando usadas como argumentos de blocos

de operações matemáticas ou de blocos cujos argumentos devam ser números. Blocos como o **if-else**, **and**, entre outros, possuem argumentos onde se podem encaixar blocos que devolvam valores booleanos.

Os blocos estão dispostos nas categorias em função da sua funcionalidade. No entanto, dentro da mesma categoria, existem blocos com diferentes formas. Pode-se tentar encaixar os blocos em 3 tipos diferentes, consoante a sua forma:

- Blocos do tipo *stack*;
- Blocos do tipo *hat*;
- Blocos do tipo *reporter*.

Os primeiros são blocos que possuem encaixes no topo e/ou no fundo. Dentro deste grupo de blocos podem incluir-se os blocos em forma de 'C', como o **forever** ou o **repeat until**, que podem conter uma sequência de blocos aninhada representando o corpo do ciclo, e os blocos que terminam *scripts*, como o **stop script** ou o **stop all**. Estes últimos, quando usados, são sempre os últimos blocos de uma *stack*. Os blocos do tipo *hat* são os que são colocados no topo das *stacks*, sendo a sua forma arredondada no topo, fazendo lembrar um chapéu. Este tipo de blocos responde a um dado evento, executando as *stacks* que encabeçam. Os blocos do tipo *reporter* devolvem valores que podem ser aproveitados por outros blocos, daí serem usados como argumentos de outros blocos. Estes blocos dividem-se em dois tipos: os que devolvem valores numéricos ou *strings* e os que devolvem valores booleanos. Os primeiros têm uma forma arredondada e encaixam em argumentos que tenham forma arredondada (números) ou rectangular (*strings*). Os segundos têm forma hexagonal e encaixam em argumentos que tenham forma hexagonal (booleanos) ou rectangular. Estes pormenores prendem-se com a preocupação de não violação das restrições dos tipos de dados. Blocos deste tipo podem ser combinados para formar expressões mais complexas.

Alguns blocos do tipo *reporter* existentes na palete, que estão associados a variáveis, sejam elas predefinidas (como o temporizador) ou criadas pelo utilizador, têm uma caixa de selecção que serve para que o valor por eles devolvido seja mostrado no **Palco**, através de um monitor (Figura 3.21). Este monitor pode ter diversos formatos: um em que apresenta o nome do bloco e o valor (Figura 3.22a), um em que apenas apresenta o valor (Figura 3.22b), e outro em que apresenta o nome, valor e um cursor deslizante que permite alterar o valor (Figura 3.22c). Este último formato está apenas disponível para variáveis criadas pelo utilizador e permite a definição de um valor máximo e mínimo.

Figura 3.21: Bloco do tipo *reporter* com caixa de selecção.

Quando uma variável é criada, o seu monitor surge imediatamente no **Palco**. Da mesma forma, quando se cria uma lista, surge um monitor no **Palco** que mostra o conteúdo da mesma (Figura 3.23). Estando inicialmente vazia, é possível adicionar elementos à lista directamente no monitor. É também possível, via menu contextual do mesmo, exportar/importar uma lista para/de um ficheiro de texto. Quando uma dada operação num *script* envolve um dado elemento de uma lista, o índice desse elemento no monitor pisca, funcionando assim como uma ajuda visual para o utilizador.

(a) Monitor com o nome do bloco e o valor.

(b) Monitor apenas com o valor do bloco.

(c) Monitor com nome, valor e cursor deslizante.

Figura 3.22: Diferentes tipos de monitores existentes para os blocos do tipo *reporter*.

Figura 3.23: Monitor de uma lista.

Programar em Scratch não segue a abordagem típica de escrita de código seguida de compilação e execução. À medida que vai juntando blocos, o utilizador pode clicar neles para ver o seu efeito, podendo até alterar a composição do *script* enquanto este executa. Daí a componente experimental do Scratch, que permite ir construindo grandes programas através da junção e teste de pequenas fracções do mesmo. Um bloco pode ser executado isoladamente, bastando para isso clicar nele, mesmo quando se encontra na palete. Este procedimento, quando aplicado a um bloco do tipo *reporter*, faz com que este mostre o seu valor, recorrendo para isso a um balão de diálogo (Figura 3.24).

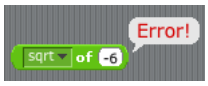
Figura 3.24: Balão de diálogo de um bloco do tipo *reporter*.

Um dos pontos fortes do Scratch é fornecer *feedback* imediato durante a execução dos *scripts*, contornando a *stack* que está a executar com uma linha branca (Figura 3.25). Isto permite saber qual a *stack* que está a executar e durante quanto tempo esta executa. À medida que os programas executam, as suas acções têm efeitos visíveis no **Palco**, permitindo ao utilizador associar mais facilmente um bloco à sua funcionalidade.

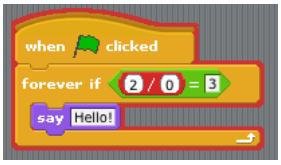
Os erros de programação, tais como as divisões por zero ou raízes quadradas de números negativos, são assinalados de duas formas. Quando são gerados por um bloco do tipo *reporter* que se encontre isolado, este reporta o erro através da mensagem "Error!" apresentada no seu balão de diálogo (Figura 3.26a). Quando são gerados por um bloco presente numa *stack*, toda a *stack* fica contornada por uma linha vermelha, ficando o bloco que gerou o erro assinalado com a cor vermelha (Figura 3.26b).

O facto de o Scratch eliminar qualquer problema relacionado com a sintaxe, codificando-a nas formas dos blocos e na forma como estes encaixam entre si, permite que as únicas situações de erro sejam aquelas geradas por problemas matemáticos (como os atrás referidos). Isto faz com que o utilizador nunca tenha de lidar com mensagens de erro derivadas de problemas de sintaxe.

Figura 3.25: Linha branca a contornar *stack* em execução.



(a) Balão de diálogo com mensagem de erro.



(b) *Stack* e bloco com erro assinalados.

Figura 3.26: Sinalização de erros no Scratch.

A opção **Start Single Stepping** do menu **Edit**, anteriormente referida, destaca cada bloco que está a executar com uma cor amarela, por forma a auxiliar o utilizador a visualizar o fluxo de execução das acções (Figura 3.27).

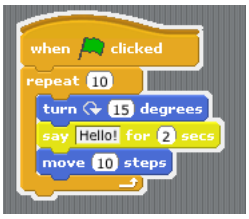


Figura 3.27: Bloco que está a executar assinalado a amarelo.

Alguns blocos agrupam conjuntos de operações relacionadas, fornecendo um menu de escolha da operação pretendida (evitando-se ter um bloco distinto por operação). Outros blocos apenas se tornam visíveis na palete quando são necessários. Dentro do primeiro caso encontram-se blocos como o de funções matemáticas científicas, presente na categoria **Operators** (Figura 3.28).

Figura 3.28: Bloco de funções matemáticas científicas.

No segundo caso encontram-se os blocos das variáveis e listas (que só surgem após a criação de uma variável ou lista) (Figura 3.29) e os blocos de controlo de motores, que surgem mediante escolha dessa opção no menu **Edit** ou quando é ligado um dispositivo USB WeDo ao computador.

(a) Categoria **Variables**, sem variáveis ou listas.

(b) Categoria **Variables**, após criação de variável.

Figura 3.29: Categoria **Variables**, antes e depois de ser criada uma variável.

Desta forma, evita-se ter um grande número de blocos que, pelo facto de se tratar de uma linguagem visual, levaria a um maior espaço ocupado por estes na palete, o que obrigaria a criar mais categorias de blocos ou a que o utilizador tivesse uma lista longa de blocos por categoria que o obrigasse a fazer *scroll*. Estas soluções não interferem na capacidade do Scratch de criar projectos de diversos tipos.

Em Scratch existem apenas 3 tipos de dados de primeira classe: booleanos, números e *strings*. As listas não são de primeira classe. Não podem ser atribuídas a variáveis, não é possível passar uma lista como argumento de um bloco, nem é possível ter listas de listas. As variáveis não são tipadas, podendo conter números, *strings* ou booleanos, evitando assim que o utilizador seja forçado a indicar o tipo da variável aquando da sua criação. Aquando da criação de variáveis ou listas, o utilizador tem a possibilidade de as definir como locais a um dado *Sprite* ou globais

a todos eles. A conversão entre *strings* e números é automaticamente realizada pelo Scratch, sempre que necessário, em função do contexto.

Apesar da linguagem suportar a existência de objectos programáveis (*Sprites*), com estado e comportamento próprio, definidos pelas variáveis e *scripts* respectivamente, o Scratch não suporta classes nem herança, não podendo, portanto, ser classificada de linguagem orientada a objectos [MRR+10]. Estas características tornam o desenvolvimento de *scripts* mais fácil de assimilar, no entanto, torna mais difícil a partilha e alteração de comportamento comum a vários *Sprites*.

Um *Sprite* não pode invocar um *script* de outro directamente. A comunicação e sincronização entre *Sprites* é feita através de um mecanismo de envio de mensagens (que mais não são do que *strings*) em *broadcast*. Um dado *Sprite* pode enviar uma mensagem através dos blocos **broadcast** ou **broadcast and wait**, e todos os *Sprites* que souberem responder a essa mensagem, isto é, todos aqueles que possuírem *stacks* encabeçadas por blocos **when I receive** associados a essa mensagem, iniciarão a execução dessas *stacks* (Figura 3.30). Esta comunicação é assíncrona caso se use o bloco **broadcast** e síncrona caso se use o **broadcast and wait**, em que o bloco aguarda que os *scripts* activados terminem a sua execução. No entanto, estes *scripts* não podem receber argumentos nem devolver valores resultantes da sua execução. Este modelo confina a definição de comportamento dos *Sprites* ao seu conjunto de *scripts* e permite manter os *Sprites* como entidades independentes que podem ser facilmente exportadas e importadas para outros projectos sem gerar problemas ao nível das dependências. Esta componente de partilha fomenta a reutilização (vários utilizadores utilizam um mesmo *Sprite* para fins distintos) e a colaboração (vários utilizadores desenvolvem *Sprites* que depois podem ser combinados num mesmo projecto).

(a) Bloco que envia a mensagem "msg".

(b) *Script* activado pela mensagem "msg".

Figura 3.30: Mecanismo de comunicação entre *Sprites*.

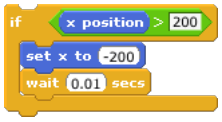
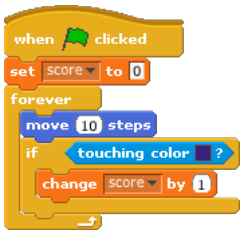
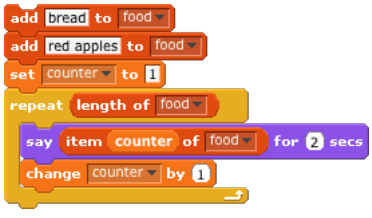
É possível ter várias *stacks* a executar simultaneamente no Scratch, quer em *Sprites* diferentes, quer no mesmo *Sprite*. Para que tal aconteça, essas *stacks* devem ser encabeçadas por um bloco do tipo *hat* que responda a um dado evento, que deverá ser o mesmo para todas elas. Através do lançamento do evento, é possível activar a execução dessas *stacks* (Figura 3.31).

Figura 3.31: *Stacks* que executam simultaneamente.

Há que ressalvar que, apesar de se falar em execução simultânea de *stacks*, na verdade esta não existe. Apesar de cada *stack* activada por um evento ter um processo associado, apenas é executado um processo de cada vez, ocorrendo mudanças de contexto de execução para alternar entre as *stacks*. O utilizador tem a sensação de execução simultânea, mas tal não acontece. Na

falta de mecanismos clássicos de controlo de concorrência como semáforos, *locks* ou monitores, o Scratch limita as mudanças de contexto de execução a duas situações: quando um bloco faz uma espera explícita (como nos blocos **wait x secs** ou **broadcast and wait**) ou no fim de um ciclo. Este esquema evita a maioria das condições de corrida e permite que o utilizador se concentre na construção de *scripts* sem ter grandes preocupações acerca de problemas de concorrência [MRR+10]. Para além disso, existe arbitrariedade da ordem de execução simultânea de várias *stacks*, pelo que o utilizador não sabe a ordem pela qual as *stacks* vão executar como resposta a um evento. Apenas na situação em que há espaço de variáveis partilhado entre as *stacks* é que o utilizador pode ter necessidade de implementar explicitamente mecanismos de controlo de concorrência. Fora isso, independentemente da ordem de execução das *stacks*, é garantido o determinismo na obtenção do resultado final.

O Scratch suporta, portanto, diversos conceitos de programação, sumarizados na Tabela 3.1 (adaptada da tabela desenvolvida por Rusk [Rus09]):

Conceito	**Explicação**	**Exemplo**
Sequência	Para criar um programa em Scratch é preciso pensar de forma sistemática na ordem de execução das instruções.	
Iteração (ciclos)	Os blocos **forever** e **repeat** podem ser usados para iteração (repetir um bloco de instruções).	
Instruções condicionais	Os blocos **if** e **if-else** testam uma condição.	
Variáveis	Os blocos das variáveis permitem criar variáveis e usá-las num programa. As variáveis podem armazenar números ou *strings*. O Scratch permite a definição de variáveis locais e globais.	
Listas	Os blocos das listas permitem armazenar e aceder a uma lista de números ou *strings*. São estruturas de dados dinâmicas.	

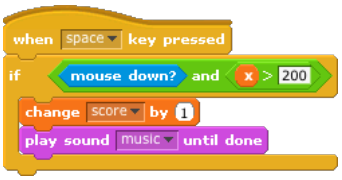
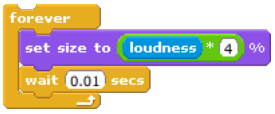
Tratamento de eventos	Os blocos **when key pressed** e **when sprite clicked** são exemplos de tratamento de eventos - respondem a eventos gerados pelo utilizador ou por outra secção do programa.	
Execução paralela	Lançar duas *stacks* ao mesmo tempo cria dois fluxos independentes que executam em paralelo.	
Coordenação e sincronização	Os blocos **broadcast** e **when I receive** permitem coordenar acções de múltiplos *Sprites*. O bloco **broadcast and wait** permite a sincronização de execução.	
Entrada de dados via teclado	O bloco **ask and wait** solicita ao utilizador que escreva. O bloco **answer** armazena o que foi escrito no teclado.	
Números aleatórios	O bloco **pick random** escolhe números inteiros aleatoriamente dentro de um certo intervalo.	
Lógica booleana	Os blocos **and**, **or** e **not** são exemplos de lógica booleana.	
Interacção em tempo real	Os blocos **mouse_x**, **mouse_y** e **loudness** podem ser usados dinamicamente para interacção em tempo real.	
Desenho de interface do utilizador	No Scratch é possível desenhar interfaces de utilizador interactivas - por exemplo, usando *Sprites* clicáveis para criar botões.	

Tabela 3.1: Conceitos de programação suportados pelo Scratch.

No entanto, outros mecanismos comuns na maioria das linguagens de programação tradicionais não existem em Scratch, tais como a criação de funções, passagem de parâmetros e retorno de valores, recursividade, escrita e leitura de ficheiros, tratamento de excepções, entre outros. De notar que estes mecanismos foram deixados de fora do Scratch para não o tornar complexo para o seu público-alvo.

3.3 Conclusão

Neste capítulo abordaram-se dois temas: o Scratch enquanto ambiente de programação e enquanto linguagem de programação visual. Foram descritos os principais componentes que constituem a interface do ambiente de programação, por forma a fornecer uma perspectiva acerca das possibilidades e alcance deste ambiente. Posteriormente, analisou-se a linguagem em si. Foram referidos aspectos essenciais de programação com esta linguagem (as entidades básicas e como é feito o desenvolvimento), para que qualquer pessoa consiga programar nesta linguagem. Foram também abordados aspectos mais técnicos, potencialidades e ainda limitações da linguagem. Esta visão global do Scratch enquanto ambiente e linguagem permite ao leitor ter as bases para compreender o trabalho desenvolvido na dissertação.

Capítulo 4

Morphic: *framework* da interface com o utilizador

Antes de se proceder à fase de análise e desenvolvimento das extensões para o ambiente Scratch, é necessário conhecer o modelo de interactividade que nele é utilizado. Sendo o Scratch desenvolvido em Squeak, a interface que é apresentada ao utilizador do Scratch não é mais do que a representação, sob a forma gráfica, de um conjunto de objectos Squeak que se interligam e actuam em conjunto. Portanto, o objecto de estudo deste capítulo é o *framework* da interface com o utilizador do Squeak: o Morphic [Mal01].

4.1 Introdução

Morphic é o nome dado ao principal *framework* da interface com o utilizador do Squeak. Foi originalmente desenvolvido por John Maloney e Randy Smith para a linguagem Self, tendo sido reescrito na totalidade em Smalltalk para ser integrado no Squeak. Nas versões 3.8 e inferiores, o Squeak também disponibiliza o *framework* MVC, que tem o mesmo nome do padrão arquitectural que implementa (*Model-View-Controller*). O Morphic trata de todo o processo de actualização do ecrã, tratamento de eventos, *drag&drop*, animação e disposição automática, permitindo que o programador se concentre nas tarefas de desenho. As suas potencialidades ao nível da criação de interfaces vão desde a criação de *widgets* personalizados, composição de componentes existentes nas suas bibliotecas, criação de visualizadores e editores de todo o tipo de informação até à construção de todo um sistema de janelas, como o do Squeak.

4.2 Morph

Toda a mecânica do Morphic gira à volta de uma entidade central: o objecto gráfico denominado *morph*. Um *morph* é um objecto do Squeak que tem uma representação visual. O utilizador pode pegar num *morph*, largá-lo em cima de outros *morphs*, redimensioná-lo, rodá-lo ou apagá-lo. Estabelecendo uma analogia com o mundo real, os *morphs* comportam-se como objectos que podem ser colocados em camadas, sobrepondo-se, dando a falsa sensação de se estar perante um mundo 3-D. Criar um novo *morph* consiste em criar uma subclasse vazia da classe concreta `Morph`, adicionando-lhe estado e comportamento de forma incremental. Uma vez que esta classe já define, por omissão, comportamento aceitável para os diferentes aspectos de um *morph* (aparência, resposta a acções do utilizador, menus, *drag&drop*, etc), a nova subclasse

criada herda de uma só vez esse comportamento, o que facilita a criação de novos *morphs*. A existência deste comportamento por omissão possibilita a instanciação e utilização imediata do novo *morph*. Vários aspectos de um *morph* podem ser manipulados directamente através do seu halo, um conjunto de controlos que circundam o *morph* e que permitem alterar a sua posição e forma, copiá-lo, apagá-lo, apresentar menus, etc.

O utilizador também pode definir estas características para o *morph* conforme deseje, podendo testá-lo e alterá-lo à medida que o desenvolve. No que toca à aparência, o utilizador tem a possibilidade de personalizar a aparência do novo *morph* bastando, para isso, implementar um único método. O Morphic não limita as possibilidades de personalização pois permite, entre outros: traçar ou preencher rectângulos, polígonos, curvas e elipses; desenhar linhas e píxeis; desenhar imagens baseadas em píxeis, com diferentes profundidades; desenhar texto num dado tipo de letra e cor. Cada *morph* possui um rectângulo que o engloba, chamado *bounds* (limites). Nunca um *morph* deve desenhar para além dos seus limites. Ao nível da interacção, um *morph* pode tratar um dado tipo de evento (representado pela classe `MorphicEvent`), definindo acções a executar quando o utilizador pressiona uma tecla ou quando usa o rato para interagir com o ambiente. No que toca ao *drag&drop*, um *morph* pode ou não aceitar que um outro seja largado sobre si. Quer o *morph* receptor quer o que é largado, podem definir acções a executar quando tal evento ocorre.

4.3 Morphs compostos

O Morphic permite combinar *morphs* simples por forma a criar estruturas gráficas complexas, recorrendo à incorporação de *morphs*, isto é, um *morph* pode incorporar outros por forma a criar um *morph* composto. Aos *morphs* incorporados num *morph* composto dá-se o nome de *submorphs*. Um *submorph* refere-se ao *morph* que o incorpora como sendo o seu *owner*. Qualquer *morph* pode ter *submorphs*, ser um *submorph*, ou ambos, sem qualquer limitação na profundidade da relação. Ao *morph* de topo de uma composição dá-se o nome de *root* (raíz). Cada *morph* apenas pode ser incorporado num único *morph* composto. Com base nestas premissas, chega-se à conclusão de que a estrutura que sustenta um *morph* composto é uma árvore, sendo que cada *morph* da árvore conhece o seu *owner* e todos os seus *submorphs* directos. Ao conhecer o seu *owner*, um *morph* fica também a conhecer o contexto em que está inserido.

Um *morph* composto comporta-se como um único objecto (atómico), ou seja, qualquer acção sobre ele tomada (mover, copiar, apagar, etc) reflecte-se em todos os *morphs* que o compõem. Se um *morph* composto for decomposto, cada um dos seus componentes torna-se num *morph* concreto que pode ser visto e manipulado.

A área rectangular que preenche o ecrã é o *World* (mundo), o *morph* que fica no topo da hierarquia e que contém todos os outros *morphs* que aparecem no ecrã (são seus *submorphs*, portanto). O objecto que representa o cursor do utilizador chama-se *hand* (mão). Quando o utilizador pega num *morph*, este é removido do *World* e adicionado à *hand*. Largando-o, ocorre o processo inverso. Quando um *morph* é apagado, é removido do seu *owner*, sendo a sua referência para este último eliminada. A mensagem `root` permite saber qual a raíz de um *morph* composto. Aquando do envio desta mensagem, toda a cadeia de *owners* é percorrida até se chegar ao *morph* cujo *owner* seja um *World* ou *hand* ou que não tenha *owner*, o qual será a raíz.

Como resultado de uma composição, os limites de um *morph* composto passam a ser definidos pela agregação dos limites dos seus *submorphs*. Um *morph* composto pode ser criado de duas formas diferentes: usando o ambiente de programação do Morphic ou através de código. Da primeira forma, basta colocar um *morph* sobre outro e invocar o comando `embed`, passando o morph da frente a ser um submorph do que está por trás. Através de código, é necessário usar o

método `addMorph:`. Em ambos os casos há um conjunto de operações que ocorrem: é actualizado o *owner* do novo *submorph* bem como as listas de *submorphs* do seu antigo e novo *owner*.

Dentro do processo de composição de *morphs*, surge a questão do posicionamento dos mesmos por forma a tomarem a disposição pretendida pelo utilizador. O Morphic faz a disposição automática de *morphs*, permitindo que o programador se concentre na ordem e aninhamento dos *submorphs* nas linhas e colunas do *morph* composto, evitando perder tempo a alinhar e redimensionar componentes. Para além disso, permite que os *morphs* se adaptem automaticamente a alterações nas dimensões. Existem, no entanto, *morphs* direccionados a tratar da disposição de outros. Estes permitem, por exemplo, que se disponham *morphs* em linhas ou colunas, sem se sobreporem, havendo possibilidade de definir o alinhamento ao longo da linha ou coluna. É possível usar esse tipo de *morphs* para preencher espaços vazios, esticando ou encolhendo conforme o espaço disponível, introduzindo expansibilidade nas composições de *morphs*. Também é possível fazer com que alterem o seu tamanho em função do tamanho dos seus *submorphs*. De salientar que o Morphic permite ainda que o utilizador crie a sua própria forma de disposição de *morphs* sem recorrer a nenhum esquema predefinido.

4.4 Princípios de concepção do Morphic

O Morphic foi concebido com quatro importantes princípios em mente: *concreteness*, *directness*, *liveness* e uniformidade.

4.4.1 *Concreteness* e *directness*

O Morphic pretende criar a ilusão de que os objectos manipulados no computador são objectos concretos que possuem as mesmas propriedades dos objectos reais. Estabelecendo uma analogia com o mundo real, o utilizador consegue ter um melhor entendimento das acções que ocorrem no ecrã. A este princípio chama-se *concreteness*. Todos os elementos presentes no ecrã podem ser tocados e manipulados, os *morphs* compostos podem ser decompostos e podem-se inspeccionar e alterar os seus *submorphs*. Estas acções são tomadas apontando directamente ao *morph* em questão, constituindo um outro princípio chamado *directness*. "*Directness* significa que um *designer* da interface com o utilizador pode iniciar o processo de examinação ou alteração dos atributos, estrutura e comportamento dos componentes da interface com o utilizador, apontando directamente à sua representação gráfica" [MS95]. Estes dois princípios permitem que o utilizador raciocine acerca dos *morphs* da mesma forma que faz para os objectos do mundo real. Existem ainda determinados factores que permitem atingir estes dois princípios:

- Durante a actualização dos componentes da interface, o utilizador nunca vê *morphs* a serem redesenhados;
- Quando se move um objecto, o utilizador vê todo o objecto a ser movido, contrariamente a certas interfaces que representam o objecto apenas com uma linha delimitadora;
- Quando o utilizador pega num objecto, aparece uma sombra com a sua forma por trás do mesmo;
- Os *morphs* podem sobrepor-se, escondendo partes de outros *morphs*, e podem ainda ter buracos que permitam ver os *morphs* que se encontram por trás;
- Os *morphs* são responsáveis por qualquer elemento gráfico que esteja visível.

Existem ainda dois conceitos importantes: acção-por-contacto (*action-by-contact*) e acção-à-distância (*action-at-a-distance*). O primeiro está relacionado com a possibilidade de se manipular directamente diferentes aspectos de um *morph* através do seu halo. O segundo está relacionado com a possibilidade de manipular objectos utilizando controlos que estão afastados deles (e.g., alterar o tamanho de um *morph* usando um menu de opções). O conceito de acção-por-contacto, equivalente à noção de manipulação directa, é usado pelo Morphic para reforçar os princípios de *concreteness* e *directness*.

Toda a manipulação que se pode fazer de um *morph* composto e seus *submorphs*, o facto de todo e qualquer *morph* ter uma representação concreta e visível e as possibilidades de disposição automática já referidas permitem dizer que o Morphic reifica a estrutura composta e a disposição automática, ou seja, traz esses conceitos do mundo abstracto para o mundo concreto [Mal01].

4.4.2 *Liveness*

Relativamente à animação gráfica, isto é, o movimento e alteração da aparência dos objectos ao longo do tempo, ela faz parte de uma característica mais genérica das interfaces com o utilizador, chamada *liveness*. "*Liveness* significa que a interface com o utilizador está sempre activa e os objectos reactivos respondem às acções do utilizador, as animações executam, a disposição ocorre, e os mostradores de informação actualizam constantemente" [MS95]. No Morphic, *liveness* e acções do utilizador são concorrentes, o que permite que decorram animações de diversos *morphs* enquanto o utilizador executa alguma acção, enriquecendo a experiência do utilizador. A *liveness* de um *morph* é definida pelo seu método `step`. O Morphic envia, periodicamente, mensagens `step` ao *morph*, as quais, na maioria das vezes, resultam no seu desenho. Associada a este método, existe uma taxa de *stepping*, `stepTime`, que define o tempo mínimo desejado entre `step`s. A taxa de *stepping* deve ser adequada e o método `step` deve ser eficiente para que a animação surta o efeito pretendido pelo utilizador. O facto de vários *morphs* poderem estar a executar o `step` simultaneamente sem ser necessário sequenciar as suas actualizações do ecrã propicia a *liveness*. Para além disso, qualquer acção como manipular um *morph*, editar o seu código, abrir um menu, etc, pode ser efectuada enquanto decorre o *stepping* e a animação, sem a necessidade de parar algum evento ou esperar pelo seu término.

4.4.3 Uniformidade

O Morphic pretende criar uma uniformidade dos objectos presentes no ecrã que ajude os utilizadores a raciocinar acerca do sistema e a manipular os *morphs* de formas não previstas pelos *designers*. Para atingir a uniformidade, o Morphic tenta evitar casos especiais. Como tal, tudo o que existe no ecrã é um *morph*, todos os *morphs* são subclasses de `Morph`, qualquer *morph* pode ter *submorphs* ou ser um *submorph*, e os *morphs* compostos comportam-se como *morphs* atómicos. Desta forma agregam-se vários aspectos sob um único modelo geral.

4.5 Ciclo de actualização da interface

O processo de actualização da interface com o utilizador é baseado no clássico ciclo *read-eval-print*. Neste caso, o *read* é substituído pelo processamento de eventos e o *print* actualiza os componentes da interface, executando operações de desenho. No Morphic, este ciclo sofre algumas alterações por forma a suportar *liveness* e disposição automática, podendo ser descrito da seguinte forma:

```
while (1) {
        processar eventos
        enviar "step" a todos os morphs activos
        actualizar a disposição dos morphs
        actualizar os componentes da interface
}
```

A imagem 4.1 representa este processo através de instruções Squeak [Squ08].

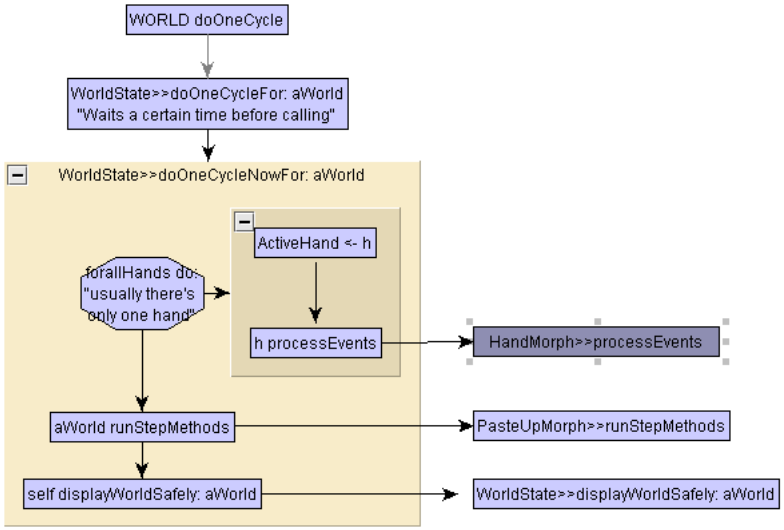


Figura 4.1: Diagrama de sequência de acções de actualização da interface.

Na primeira fase deste ciclo é feito o processamento de eventos. O processamento de eventos consiste em delegar os eventos aos *morphs* apropriados. Os eventos do teclado são delegados ao *morph* que possui o *focus* do teclado. Caso nenhum *morph* o possua, o evento é descartado. Já os eventos de *mouse down* do rato são delegados em função da localização do rato, sendo que o *morph* que estiver "mais próximo do utilizador" (mais à frente) na localização do evento é que fica encarregue de os tratar. Esse *morph* deverá indicar previamente que pretende receber eventos do rato. Dentro de um *morph* composto, os eventos do rato seguem a mesma lógica de serem tratados pelo *submorph* que aparece mais à frente. Caso esse *submorph* não pretenda tratar o evento, então delega essa tarefa ao seu *owner* e assim sucessivamente até que algum *morph* trate o evento.

A propriedade *liveness* é garantida através de uma lista de *morphs*, aos quais, em cada iteração do ciclo, deve ser enviada a mensagem `step` e cujo `stepTime` seguinte é actualizado. Os *morphs* que foram apagados são removidos dessa lista por forma a que não se faça o *stepping* de *morphs* que já não estão no ecrã e para que estes possam ser recolhidos pelo *garbage collector*.

A disposição de um *morph* é mantida de forma incremental. Quando um *morph* é alterado de tal forma que possa influenciar a disposição (e.g., adicionar um novo *submorph*), é desencadeado um conjunto de operações para a reconstruir. Em primeiro lugar, a disposição do *morph* alterado é actualizada. Se tal afectar o espaço disponível para os seus *submorphs*, então a disposição destes também é actualizada. Depois, se o próprio *morph* necessita de mais ou menos espaço, então a disposição do seu *owner* é também actualizada, bem como a dos *owners* que se seguem na

hierarquia, seguindo o mesmo algoritmo, caso seja necessário.

Para efectuar a actualização final que faz com que as alterações existentes fiquem visíveis para o utilizador é utilizado um algoritmo eficiente e de elevada qualidade, por forma a que o utilizador não se aperceba que o ecrã está a ser redesenhado. O Morphic mantém uma lista, chamada "lista de danos", que contém todas as partes do ecrã que têm de ser redesenhadas. De cada vez que um *morph* altera a sua aparência, envia a si mesmo a mensagem `changed`, que adiciona o rectângulo que o engloba (*bounds*) à lista de danos. Para cada rectângulo da lista de danos são redesenhados todos os *morphs* que o intersectem. O algoritmo garante que este processo é escondido do utilizador, sendo-lhe apenas apresentado o resultado final. Desta forma o utilizador nunca visualiza estágios intermédios deste processo, resultando numa animação suave. No final deste processo, a lista de danos é limpa para a preparar para a próxima iteração do ciclo.

4.6 Comparação com o *framework* MVC

Tendo o *framework* MVC existido no Squeak, e dada a sua utilização em muitos ambientes de programação, torna-se relevante estabelecer uma comparação entre ele e o Morphic.

Apenas como forma de enquadramento, recorde-se que o padrão MVC, implementado pelo *framework* do Smalltalk-80 com o mesmo nome, define três conceitos principais: o modelo, a vista e o controlador. O modelo reflecte o domínio da aplicação, definindo estado e comportamento para as suas entidades. A vista apresenta a informação sobre o modelo através de uma interface com o utilizador. O controlador trata da entrada de dados pelo utilizador e indica ao modelo para realizar determinada acção, que pode mudar o seu estado, sendo este reflectido para a vista.

No Morphic, um *morph* combina os papéis de controlador e vista, uma vez que trata da entrada de dados pelo utilizador e da apresentação de informação. Isto acontece por motivos de simplificação e porque, muitas vezes, as classes da vista e do controlador são muito inter-dependentes. Quanto ao modelo, existem *morphs* que são simplesmente objectos gráficos que não precisam de um modelo de suporte e existem outros que são o seu próprio modelo. Ainda assim, o Morphic, tal como o MVC, permite ter várias vistas sobre o mesmo modelo, utilizando o tradicional mecanismo de actualização para informar as vistas acerca de alterações no modelo.

Outro aspecto diferenciador é a *liveness*. No MVC, apenas uma vista de topo tem o controlo num dado momento e apenas ela pode desenhar, dentro dos seus limites. Caso desenhe fora dos seus limites, tem de guardar e restaurar os píxeis afectados. Este mecanismo é mais eficiente do que o usado pelo Morphic, no entanto torna mais difícil suportar a *liveness* porque não é trivial conseguir que várias vistas activas intercalem as suas actualizações do ecrã sem desenharem umas por cima das outras. Já no Morphic é mais fácil graças à utilização da lista de danos na fase de actualização da interface e ao mecanismo de actualização incremental do ecrã.

O Morphic também apresenta diferenças ao nível da *concreteness*. No MVC, quando se move ou redimensiona uma janela, o *feedback* dessa operação é dado sob a forma de um rectângulo vazio, enquanto que no Morphic move-se o próprio objecto. A abordagem do MVC é, mais uma vez, mais eficiente, visto que apenas alguns píxeis são actualizados à medida que se arrasta o rectângulo. De notar que o Morphic também suporta esta abordagem.

4.7 Conclusão

Este capítulo debruçou-se sobre o modelo de interactividade presente no Scratch, que é fornecido pelo *framework* Morphic. Foi apresentada a entidade base deste sistema, o objecto gráfico *morph*, bem como as possibilidades de construção de interfaces interactivas usando estes componentes

de forma simples ou agregados em estruturas complexas e com comportamentos personalizados. O modo de funcionamento e as ideias chave por detrás deste *framework* foram abordadas por forma a salientar os seus pontos diferenciadores, tendo-se terminado com um comparativo com o *framework* MVC. Estes conceitos ganham importância na fase de análise do código fonte do Scratch, permitindo perceber não só a nomenclatura adoptada para os nomes de determinadas entidades, como também o porquê de certas instruções Squeak existentes nalguns métodos.

Capítulo 5

Análise e desenvolvimento de extensões

As novas extensões desenvolvidas para o ambiente Scratch são apresentadas neste capítulo. É apresentada a descrição dos objectivos a alcançar, o estudo feito da estrutura interna do Scratch e são explicadas, uma a uma, as novas funcionalidades adicionadas, sendo acompanhadas de imagens e, por vezes, de excertos de código fonte que facilitem a sua compreensão.

5.1 Objectivos

Nesta extensão do Scratch, pretende-se introduzir a possibilidade do utilizador criar os seus próprios *scripts* sob a forma de blocos. A funcionalidade de cada bloco é definida por um *script* que este encapsula. Doravante, neste documento, estes blocos serão referidos como "*scripts* definidos pelo utilizador", usando-se o acrónimo SDU para os referenciar. Criando um *script*, este fica disponível numa biblioteca de *scripts* definidos pelo utilizador para que, mais tarde, este o possa utilizar no contexto de um outro *script*, importando-o. A acção de importação fica a cargo de um novo bloco de controlo que permite executar a funcionalidade encapsulada pelo *script* definido pelo utilizador. À volta desta nova entidade surgem funcionalidades como consulta da definição do *script* definido pelo utilizador, teste isolado do *script*, actualização da definição do *script*, criação de um novo *script* a partir da definição de outro (modificando-a ou não) e eliminação de um *script*. A possibilidade de escrever e ler projectos que tirem partido desta nova entidade também é de grande interesse. Por fim, pretende-se introduzir a capacidade de exportar e importar estes *scripts* para um formato persistente (ficheiros).

5.2 Notas prévias

Antes de entrar na exploração do código fonte do Scratch é necessário saber alguns detalhes sobre o ambiente de trabalho. À data de escrita deste documento, o Scratch encontra-se na versão 1.4 e o Squeak na versão 4.2. O desenvolvimento na linguagem Squeak foge ao tradicional conceito de múltiplos ficheiros de código editáveis num ambiente de desenvolvimento integrado e ao processo de programação "escrever → compilar → executar". O que o Squeak proporciona é uma imagem (ficheiro de extensão `.image`) que contém o código das classes e métodos existentes no Squeak, o qual pode ser explorado usando diversas ferramentas, sendo a mais utilizada o *System Browser*. A leitura e escrita de código é toda feita dentro do *browser*. A existência de

menus de contexto, dentro e fora das ferramentas como o *System Browser*, permite o acesso a diversas funcionalidades e a outras ferramentas, como o *Debugger*, *Inspector* e *Explorer* (para "olhar" para o interior de um objecto e ver os valores associados às suas variáveis de instância e explorar directamente a hierarquia dos tipos dessas variáveis), *Process Browser* (para saber que processos estão a executar num dado momento), ferramentas para encontrar métodos (quer indicando parte ou totalidade do nome, ou fornecendo um exemplo do seu comportamento), ferramentas para exportar/importar código feito por outros e ainda um sistema de controlo de versões distribuído chamado *Monticello*. O código fonte do Scratch é distribuído, como tal, como uma imagem do código do Squeak juntamente com as classes do Scratch. No entanto, esta imagem traz a versão 2.8 do Squeak, pelo que das ferramentas referidas, apenas são fornecidas o *System Browser*, *Debugger*, *Inspector* e as que permitem encontrar métodos.

No que toca a modificações do código fonte do Scratch, é importante reter algumas restrições impostas pela sua licença:

- A licença permite a distribuição de projectos derivados a partir do código fonte do Scratch para uso não comercial;
- Não é permitido o uso da palavra "Scratch" para referir projectos derivados (excepto na frase "Baseado no Scratch do MIT Media Laboratory");
- Não é permitido uso do logotipo nem do gato oficial do Scratch em projectos derivados;
- Não é permitida a implementação da capacidade de enviar projectos para qualquer página do Scratch do MIT [Scr12a];
- Cópias ou projectos derivados têm de manter a licença e nota de direitos de autor do Scratch;
- Projectos derivados têm de disponibilizar publicamente o seu código fonte.

Antes de se passar à apresentação das funcionalidades desenvolvidas, tem interesse salientar que a referência a um dado método de uma dada classe seguirá uma convenção tipográfica usada por Black *et al.* [BDNP07], `Classe`≫`método`, que é comum entre os programadores de Smalltalk.

5.3 Análise da estrutura interna do Scratch

A estrutura interna do Scratch é representada pelo conjunto de classes que lhe dão suporte. No Squeak, as classes estão organizadas em pacotes, aos quais se dá o nome de "categorias do sistema" (*system categories*). Dentro de cada classe, o explorador de classes do Squeak ainda fornece um outro nível de organização chamado "protocolo", que é um agrupamento de métodos relacionados. Numa primeira fase, torna-se importante a análise das categorias de classes que suportam o Scratch:

- `Scratch-Objects`: classes que implementam os *Sprites* e o **Palco** e que definem os elementos *media* (imagens, vídeos e sons);
- `Scratch-Blocks`: classes que representam os diferentes tipos de blocos;
- `Scratch-Execution Engine`: classes relativas à execução de um *script* (eventos, processos e *stack frame*);
- `Scratch-Object IO`: classes responsáveis pela serialização de objectos;

- `Scratch-UI-Dialogs`: classes que implementam janelas de interacção com o utilizador;
- `Scratch-UI-Panes`: classes que representam os vários painéis da interface do Scratch (palete, área de *scripting*, etc);
- `Scratch-UI-Watchers`: classes que implementam os monitores de variáveis, com as suas diversas opções;
- `Scratch-UI-Support`: classes de suporte à interface;
- `Scratch-Paint`: classes que representam o editor de imagens integrado do Scratch;
- `Scratch-Sound`: classes para edição e manipulação de sons;
- `Scratch-Translation`: classes que suportam a utilização de múltiplos idiomas no Scratch;
- `Scratch-Networking`: contém apenas a classe `ScratchServer`, que permite a comunicação ponto-a-ponto entre instâncias do Scratch ou entre o Scratch e outras aplicações.

Esta estrutura pode ser visualizada nos diagramas de classes das Figuras A.1 a A.12, presentes no Anexo A.

Olhando em particular para as classes que representam os blocos do Scratch, obtém-se o diagrama presente na Figura A.13 do Anexo A. As classes que nele figuram são:

- `Kernel-Objects :: Object`: superclasse de todas as classes, fornece comportamento por omissão a todas as suas subclasses (e.g., acesso, cópia, comparação, tratamento de erros, envio de mensagens e reflexão);
- `Morphic-Kernel :: Morph`: classe que representa a entidade *morph*;
- `Scratch-Blocks :: BlockMorph`: superclasse de todas as classes que representam os blocos;
- `Scratch-Blocks :: CommandBlockMorph`: classe que representa blocos do tipo *stack* ou *reporter*;
- `Scratch-Blocks :: IfElseBlockMorph`: classe que representa um bloco em forma de 'E', implementando a estrutura de controlo `if-then-else`;
- `Scratch-Blocks :: CBlockMorph`: classe que representa um bloco que pode conter uma sequência aninhada de vários blocos (e.g., um ciclo);
- `Scratch-Blocks :: SetterBlockMorph`: classe que representa um bloco que permite alterar o valor de uma variável;
- `Scratch-Blocks :: ReporterBlockMorph`: superclasse de todas as classes que representam blocos do tipo *reporter*;
- `Scratch-Blocks :: ListContentsBlockMorph`: classe que representa um bloco do tipo *reporter* que devolve o conteúdo de uma lista;
- `Scratch-Blocks :: VariableBlockMorph`: classe que representa um bloco do tipo *reporter* que devolve o valor de uma variável, seja ela predefinida ou criada pelo utilizador;
- `Scratch-Blocks :: HatBlockMorph`: superclasse de todas as classes que representam blocos do tipo *hat*;

- `Scratch-Blocks :: EventHatMorph`: superclasse das classes que representam blocos do tipo *hat* que são activados por eventos do Scratch;
- `Scratch-Blocks :: KeyEventHatMorph`: classe que representa blocos do tipo *hat* que são activados por eventos do teclado;
- `Scratch-Blocks :: MouseClickEventHatMorph`: classe que representa blocos do tipo *hat* que são activados pelo clique do rato num objecto;
- `Scratch-Blocks :: WhenHatBlockMorph`: classe que representa blocos do tipo *hat* que suportam argumentos booleanos. Esta classe encontra-se obsoleta, não sendo utilizada na versão actual do Scratch;
- `Scratch-Blocks :: CommentBlockMorph`: classe que representa um bloco que é um comentário. Esta classe encontra-se obsoleta, não sendo utilizada na versão actual do Scratch;
- `Morphic-Basic :: BorderedMorph`: classe que define uma borda com uma dada largura (em píxeis) e uma cor para a preencer;
- `Scratch-Blocks :: ArgMorph`: superclasse abstracta de todos os tipos de *morphs* que podem ser usados como argumentos de blocos;
- `Scratch-Blocks :: BooleanArgMorph`: classe que representa os argumentos booleanos de alguns blocos;
- `Scratch-Blocks :: ChoiceArgMorph`: classe que representa os argumentos que permitem ao utilizador escolher uma opção a partir de um menu de opções;
- `Scratch-Blocks :: AttributeArgMorph`: classe que representa os argumentos que são atributos de *Sprites* ou do **Palco**;
- `Scratch-Blocks :: ChoiceOrExpressionArgMorph`: classe que representa os argumentos que tanto podem ser opções escolhidas a partir de um menu como podem ser substituídos por blocos do tipo *reporter* que sejam numéricos;
- `Scratch-Blocks :: ColorArgMorph`: classe que representa um argumento que é uma cor, que pode ser alterada através dum selector de cores;
- `Scratch-Blocks :: EventTitleMorph`: classe que representa o nome de eventos a usar em blocos do tipo *hat*;
- `Scratch-Blocks :: ExpressionArgMorph`: classe que representa um argumento que pode ser um número ou uma *string* e que pode ser editado;
- `Scratch-Blocks :: ExpressionArgMorphWithMenu`: classe que adiciona à sua superclasse `ExpressionArgMorph` a possibilidade de definir o valor do argumento através de um menu;
- `Scratch-Blocks :: SpriteArgMorph`: classe que representa um argumento que pode ser um *Sprite* ou o **Palco**.

Ainda que não sendo uma classe que implemente um bloco, é importante referenciar a classe `ScratchCommentMorph`, responsável pelos comentários que o utilizador do Scratch pode colocar nos seus *scripts*. As suas duas superclasses não são abordadas, estando presentes no diagrama apenas para estabelecer a ligação entre a hierarquia principal e a classe dos comentários.

Por forma a mapear as classes dos blocos nas suas representações gráficas, construiu-se a Tabela 5.1.

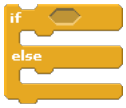
Classe	Bloco
`CommandBlockMorph`	
`IfElseBlockMorph`	
`CBlockMorph`	
`SetterBlockMorph`	
`ReporterBlockMorph`	
`ListContentsBlockMorph`	
`VariableBlockMorph`	
`EventHatMorph`	
`KeyEventHatMorph`	
`MouseClickEventHatMorph`	

Tabela 5.1: Classes dos blocos do Scratch e respectivas representações gráficas.

Para facilitar a identificação dos componentes da interface do Scratch e associá-los mais facilmente às variáveis que os representam no código fonte (bem como às classes que os representam) foi elaborado um esquema de cores (Figuras B.1a e B.1b do Anexo B), em que cada cor identifica um componente (classe).

Estes esquemas ajudam a conhecer os principais componentes do Scratch e a memorizar os seus nomes, auxiliando no processo de desenvolvimento das extensões.

5.4 Desenvolvimento das extensões

Com os objectivos mencionados e tendo já analisado de forma geral o ambiente, parte-se para a próxima fase do trabalho: o desenvolvimento das extensões do ambiente.

5.4.1 Adição de uma nova aba

Como primeiro objectivo pretende-se adicionar uma nova aba ao painel central, cuja estrutura original contém apenas 3 abas (Figura 5.1).

Figura 5.1: Painel central com 3 abas.

Pretende-se que a nova aba permita, tal como a aba **Scripts**, a criação de *scripts*. Da Figura 5.2, sabe-se que a área ocupada pelo Scratch é definida por um `ScratchFrameMorph`, o painel central é um `ScratchScriptEditorMorph`, e a área onde se escrevem os *scripts* é representada por um `ScratchScriptsMorph` dentro de um `ScrollFrameMorph2`.

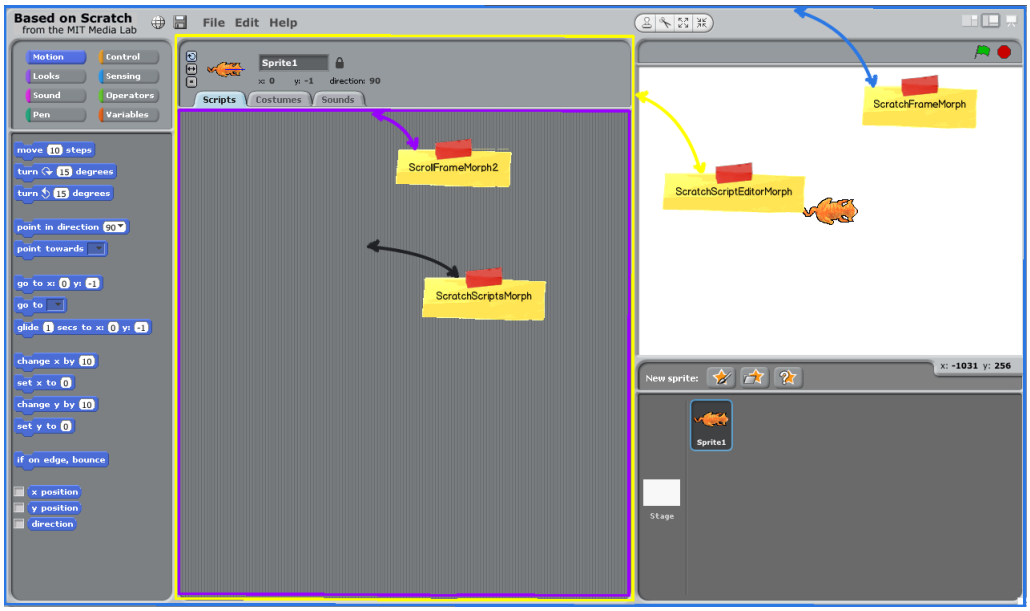


Figura 5.2: Identificação por cores de alguns elementos da interface do Scratch.

Nesta composição de *morphs* é possível ver que o componente das abas se encontra definido na classe `ScratchScriptEditorMorph`, através da sua variável de instância `tabPaneMorph` (Figura 5.3).

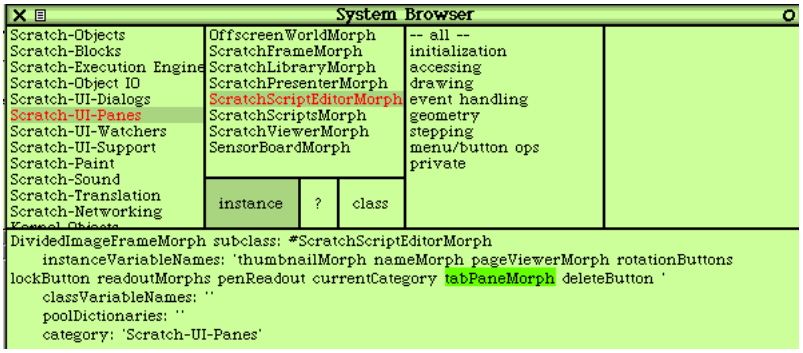


Figura 5.3: Classe ScratchScriptEditorMorph.

Para perceber como é criado o componente, é necessário olhar para o construtor desta classe, que é dado pelo método `initialize`. No corpo do mesmo, verifica-se que a última instrução é a responsável por inicializar o componente das abas (Código 5.1):

```
self createTabPane.
```

Código 5.1: ScratchScriptEditorMorph»initialize

É invocado o método `createTabPane` da mesma classe. Inspeccionando este método, verifica-se que este se encontra dividido em 3 etapas: criação e definição das características do *morph* que representa o componente (`ScratchTabPaneMorph`), adição das abas ao componente, definição da aba inicial e adição do componente ao componente de topo que o incorpora. No passo de adição das abas, as mesmas são definidas através de um *array* estático com os seus nomes, ao qual se tem de acrescentar o nome da nova aba: **Test** (Código 5.2).

```
createTabPane
...
    "add the tabs"
    #(Scripts Costumes Sounds Test) do: [:spec |
...
```

Código 5.2: ScratchScriptEditorMorph»createTabPane

A partir daqui passa a ser possível ver a nova aba (Figura 5.4).

Figura 5.4: Aba **Test** criada.

Esta aba, no entanto, ainda não tem conteúdo, limita-se a existir dentro do componente `ScratchTabPaneMorph`. Isto pode ser verificado pelo facto de a aba mostrar como seu conteúdo o conteúdo da última aba seleccionada. Para adicionar conteúdo próprio à aba é necessário olhar para o método `currentCategory:` da classe `ScratchScriptEditorMorph`, onde são definidos os conteúdos das abas já existentes. Como se pretende adicionar uma área que permita a edição de *scripts*, é necessário atentar no código que faz o mesmo para a aba **Scripts** (Código 5.3).

```
currentCategory: aString
...
    currentCategory = 'Scripts' ifTrue: [
            pageViewerMorph contents: self target blocksBin].
...
```

Código 5.3: ScratchScriptEditorMorph»currentCategory:

Analisando, verifica-se que o conteúdo da aba é construído enviando a mensagem `target` à instância da classe `ScratchScriptEditorMorph`, sendo depois enviada a mensagem `blocksBin` ao resultado desta última operação. O método `target` devolve uma de duas coisas: uma referência para o *Sprite* seleccionado na lista de *Sprites* ou uma referência para o **Palco**, caso este esteja seleccionado. A referência para um *Sprite* é dada por uma instância da classe `ScratchSpriteMorph`, que representa o(s) *Sprite(s)*, enquanto que o **Palco** é representado pela classe `ScratchStageMorph`. Estas duas classes são subclasses da classe `ScriptableScratchMorph`, como se pode ver na hierarquia da Figura 5.5.

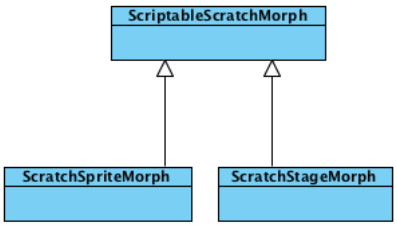


Figura 5.5: Hierarquia de classes dos objectos programáveis do Scratch.

À referência devolvida é então enviada a mensagem `blocksBin`, o método selector da variável de instância com o mesmo nome. Analisando a superclasse `ScriptableScratchMorph`, verifica-se que esta possui a variável de instância de nome `blocksBin`, bem como o método referido. Esta variável é inicializada nesta classe com uma instância de `ScratchScriptsMorph`, classe que representa o componente que contém os *scripts* (completos ou parcialmente completos) de um `ScriptableScratchMorph`. É preciso então adicionar suporte à nova aba para conter *scripts*. Para tal, declara-se uma nova variável de instância na classe `ScriptableScratchMorph`, denominada `testBlocksBin`, e inicializa-se a mesma no método `initialize` com uma nova instância da classe dos contentores de *scripts* (Código 5.4).

```
initialize
...
    blocksBin _ ScratchScriptsMorph new.
    testBlocksBin _ ScratchScriptsMorph new.
...
```

Código 5.4: ScriptableScratchMorph»initialize

Dentro do protocolo `blocks` da mesma classe, define-se o método selector da nova variável (Código 5.5).

```
testBlocksBin

    ^ testBlocksBin
```

Código 5.5: ScriptableScratchMorph»testBlocksBin

Falta apenas adicionar no método `currentCategory:` da classe `ScratchScriptEditorMorph` o código necessário para que a nova aba passe a ter a sua própria área de *scripting* (Código 5.6).

```
currentCategory: aString
...
    currentCategory = 'Test' ifTrue: [
            pageViewerMorph contents: self target testBlocksBin].
...
```

Código 5.6: ScratchScriptEditorMorph»currentCategory:

Neste momento a aba apresenta um fundo semelhante ao da aba **Scripts** (Figura 5.6).

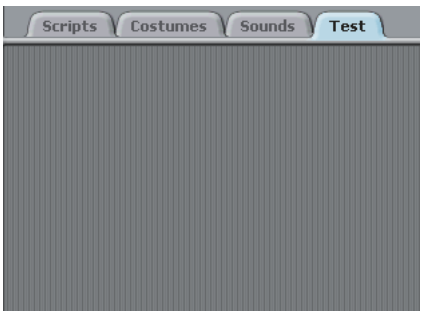


Figura 5.6: Aba **Test** com área de *scripting*.

Para que se possam efectivamente escrever *scripts* na área desta nova aba falta ainda ajustar um pormenor. Sempre que se tenta colocar um bloco nesta área, a aba seleccionada deixa de ser a **Test** e passa a ser a **Scripts**. Analisando o protocolo `event handling` da classe `ScratchScriptEditorMorph`, é possível consultar o método `mouseEnterDragging:`, o qual informa que altera a aba para a **Scripts** sempre que algum bloco está a ser arrastado. Como tal, basta adicionar uma condição para que não execute tal comportamento caso a aba seja a **Test** (Código 5.7).

```
mouseEnterDragging: evt
    "Switch the tabs to script if a block is current being dragged"

    (currentCategory = 'Scripts' or: [currentCategory = 'Test']) ifFalse:[
            self currentCategory: 'Scripts'.
            tabPaneMorph currentTab: 'Scripts'].
```

Código 5.7: ScratchScriptEditorMorph»mouseEnterDragging:

Esta última alteração conclui a tarefa de criação de uma nova aba com a capacidade de conter *scripts* (Figura 5.7).

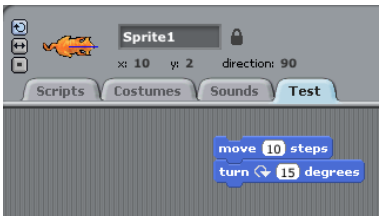


Figura 5.7: Aba **Test** com blocos.

5.4.2 Adição de uma nova categoria

O segundo objectivo passa por adicionar uma categoria de blocos às já existentes (Figura 5.8), com o nome **MyScripts**, na qual o utilizador terá à disposição os *scripts* por si definidos.

Figura 5.8: Categorias de blocos.

Uma vez que as categorias correspondem a imagens que são carregadas na *skin* do Scratch, é preciso efectuar alterações na mesma. Investigando o método de classe `readSkinFrom:` da classe `ScratchFrameMorph`, verifica-se que os diversos *forms* da *skin* são lidos de uma directoria e armazenados na variável de classe `ScratchSkin`, que é um dicionário (`Dictionary`). Na documentação do método, onde indica que essa directoria também tem o nome de `ScratchSkin`, é apresentada uma instrução de carregamento dos elementos da *skin* (Código 5.8).

```
self readSkinFrom: (FileDirectory default directoryNamed: 'ScratchSkin')
```

Código 5.8: ScratchFrameMorph»readSkinFrom:

Olhando para o conteúdo desta directoria no sistema de ficheiros é possível encontrar, para cada categoria de blocos existente, 3 imagens. Tomando como exemplo a categoria **Control**, existe:

- `control.gif`
- `controlOver.gif`
- `controlPressed.gif`

Estas imagens são as que são apresentadas quando o botão da categoria **Control** está no seu estado normal por omissão, quando o rato passa por cima e quando é pressionado.

Se se inspeccionar a variável de classe `ScratchSkin`, é possível encontrar 3 entradas correspondentes às imagens atrás referidas: `#control`, `#controlOver` e `#controlPressed` (Figura 5.9).

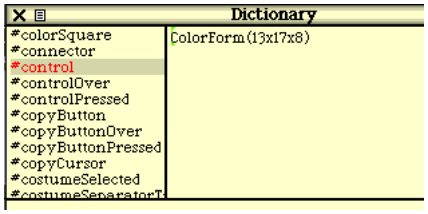


Figura 5.9: Entradas do dicionário ScratchSkin.

Assim, para introduzir uma nova categoria, é necessário que o dicionário passe a ter entradas para 3 novas imagens. Para isso, cria-se uma cópia das imagens referidas na pasta `ScratchSkin`, renomeando-as para `MyScripts.gif`, `MyScriptsOver.gif` e `MyScriptsPressed.gif`. Com a ajuda de um programa de edição de imagem, editam-se as imagens e altera-se a sua cor para

o preto. De seguida, basta executar o método `readSkinFrom:` com os parâmetros indicados anteriormente, para que a *skin* seja carregada a partir da pasta `ScratchSkin`, ficando assim carregadas as novas imagens para a nova categoria (e o dicionário `ScratchSkin` imediatamente actualizado) (Código 5.8).

Posto isto, apenas é necessário introduzir o novo botão na interface. A classe responsável por tal zona da interface é a `ScratchViewerMorph`. O método de instância desta classe responsável pela criação das categorias é o `rebuildCategorySelectors`. Neste, é necessário adicionar o nome da nova categoria ao *array* que as contém (Código 5.9).

```
catList _ #(
        motion          control
        looks           sensing
        sound           operators
        pen             variables
        MyScripts).
```

Código 5.9: ScratchViewerMorph»rebuildCategorySelectors

Como se pretende ter um número ímpar de categorias, é preciso ainda alterar a forma como é calculado o espaço reservado aos botões das mesmas, fazendo um arredondamento no cálculo do tamanho do *array* de categorias (Código 5.10).

```
catButtonsExtent _ ((2 * maxExtent x) + (3 * pad)) @ (((catList size / 2) rounded
                    * (maxExtent y + 6)) + 25).
```

Código 5.10: ScratchViewerMorph»rebuildCategorySelectors

A partir deste momento, já é possível ver a nova categoria (Figura 5.10).

Figura 5.10: Nova categoria: **MyScripts**.

Para criar coerência visual, os blocos desta categoria devem apresentar a cor que a identifica. Para tal, no método de classe `blockColorFor:` da classe `ScriptableScratchMorph`, cuja documentação indica que devolve a cor de uma determinada categoria, é necessário indicar a cor que deve ser devolvida para os blocos da categoria **MyScripts**. Esta cor pode ser devolvida usando valores HSV (*Hue*, *Saturation*, *Value*) ou RGB (*Red*, *Green*, *Blue*). Os valores usados na edição das imagens para obter a cor preta são (H,S,V) = (0,0,0.25), que correspondem ao (R,G,B) = (64,64,64), sendo usados agora de forma programática (Código 5.11).

```
        'MyScripts' = aCategory ifTrue: [^ (Color h:0 s:0 v:0.25)].
```

Código 5.11: ScriptableScratchMorph»blockColorFor:

5.4.3 Criação do bloco de importação de SDUs

Esta etapa concentra-se na criação de um novo bloco personalizado cuja funcionalidade consiste na importação de SDUs, para utilização no contexto de outros *scripts*. A criação de um bloco envolve dois passos:

1. Definir a especificação do bloco;
2. Definir o método que lhe é associado.

Existem três classes potenciais candidatas a possuírem a especificação e método de um bloco: `ScriptableScratchMorph`, `ScratchSpriteMorph` e `ScratchStageMorph` (Figura 5.5). Todas estas classes possuem um método de classe `blockSpecs`, onde são definidas as especificações dos vários blocos existentes. Escolher em qual destas classes é que deve ser escrita a especificação do novo bloco consiste em determinar se se quer que o novo bloco esteja disponível para ser usado apenas pelos *Sprites* (caso em que deverá ser definido na classe `ScratchSpriteMorph`), apenas pelo **Palco** (caso em que deverá ser definido na classe `ScratchStageMorph`), ou por ambos (classe `ScriptableScratchMorph`). Portanto, a classe `ScriptableScratchMorph` define os blocos que são comuns aos `ScratchSpriteMorph`s e aos `ScratchStageMorph`s. Os restantes blocos existentes estão definidos nas suas duas subclasses. Daí que quando se selecciona um *Sprite* ou o **Palco** no Scratch, há blocos na palete que desaparecem e outros que aparecem (e.g., seleccionando o **Palco**, verifica-se que a categoria **Motion** não tem nenhum bloco).

As categorias (e seus blocos) estão assim divididas por estas classes:

- `ScratchStageMorph`: **Sensing**, **Looks**, **Pen**;
- `ScratchSpriteMorph`: **Motion**, **Pen**, **Looks**, **Sensing**;
- `ScriptableScratchMorph`: **Control**, **Operators**, **Sound**, **Variables**. Ainda inclui os blocos dos motores da categoria **Motion**.

Existem casos em que uma mesma categoria possui blocos definidos nas duas subclasses, no entanto são blocos que realizam acções diferentes em função do objecto a que estão associados (*Sprite* ou **Palco**), e.g., na classe `ScratchStageMorph` está definido o bloco **switch to background <background>** e na classe `ScratchSpriteMorph` está definido o **switch to costume <costume>**, ambos da categoria **Looks**.

Explicado este cenário, pode-se passar à criação do bloco. O bloco a construir tem o nome **import** e, numa primeira versão, trata de executar o *script* de blocos que engloba. Como é um bloco que trata do controlo da execução de um *script*, faz sentido que seja colocado na categoria **Control**. Como também se pretende que esteja disponível tanto para os *Sprites* como para o **Palco**, a classe adequada para ser definido é a `ScriptableScratchMorph` (embora a categoria **Control** esteja definida nesta classe, se se quisesse que o bloco estivesse disponível apenas para os *Sprites* ou para o **Palco**, nada impedia que se definisse o bloco numa das subclasses, apenas seria necessário introduzir a categoria a que ele pertence juntamente com a sua especificação). Nesta classe, é necessário alterar o método de classe `blockSpecs` que define especificações de blocos, sob a forma de *arrays*, intercaladas pelos nomes das categorias. Uma especificação de um bloco é um *array* da forma:

```
(<string de especificação> <tipo> <selector> [valores iniciais para argumentos])
```

A *string* de especificação do bloco contém o texto que o bloco vai apresentar na sua representação gráfica, ou seja, é o nome do bloco. Esta *string* de especificação é definida na classe `CommandBlockMorph` pela variável `commandSpec`. Esta define o texto do bloco, número e ordem dos argumentos. Esta *string* consiste numa sequência de termos intercalados por especificações de argumentos que indicam onde eles aparecem no bloco (e.g., **repeat %n times**). Os argumentos dos blocos são declarados na *string* de especificação usando o formato `%letra`, onde `letra` pode assumir vários valores, consoante o tipo de argumento que se deseje. Estes valores estão definidos no método de instância `uncoloredArgMorphFor:` da classe `CommandBlockMorph` e são os seguintes:

```
a: atributo de um Sprite (e.g., valor da sua coordenada x);
b: valor booleano;
c: cor seleccionada via selector de cor com palete;
C: cor seleccionada via selector de cor com conta-gotas;
d: menu da direcção do Sprite / valor numérico;
D: menu dos tambores midi;
e: nome de evento;
f: menu de funções matemáticas (módulo, seno, raíz quadrada, etc);
g: menu dos vários efeitos gráficos;
h: menu de valores booleanos dos sensores;
H: menu de sensores;
i: menu de acesso ao valor de um índice de uma lista;
I: menu de instrumentos midi / valor numérico;
k: menu das teclas do teclado;
l: menu com os nomes dos trajes do Sprite;
L: menu das listas (criadas na categoria Variables);
m: menu dos Sprites / cursor do rato;
M: menu dos nomes dos motores;
n: valor numérico;
N: menu das notas do teclado / valor numérico;
s: valor textual (string);
S: menu dos nomes dos sons;
v: menu das variáveis (criadas na categoria Variables);
W: menu das direcções dos motores;
x: menu dos nomes das cenas (scenes);
y: menu dos índices dos elementos a apagar de uma lista.
```

O tipo do bloco define o formato do bloco. Os valores usados para indicar o tipo estão explicados na documentação do método de classe `blockSpecs` da classe `ScriptableScratchMorph`:

```
-   sem valor (cria um bloco do tipo stack);
b   bloco do tipo reporter que devolve um valor booleano;
c   bloco em forma de 'C' que contém uma sequência de comandos;
r   bloco do tipo reporter que devolve um valor numérico ou string;
s   comando especial com uma regra de avaliação própria;
t   comando temporal;
E   bloco do tipo hat activado via mensagem (broadcast);
K   bloco do tipo hat activado via evento do teclado;
M   bloco do tipo hat activado via evento do rato;
S   bloco do tipo hat activado via clique na bandeira de execução de scripts;
W   bloco do tipo hat activado quando uma condição se verifica.
```

Os valores, quer da *string* de especificação quer do tipo de bloco, estão associados com a descrição, em Inglês, do efeito que causam.

O selector é o nome do método que fica associado ao bloco e que define o seu comportamento. O último parâmetro corresponde aos valores que poderão aparecer por omissão nos argumentos dos blocos.

Sabendo isto, para definir o novo bloco, é preciso adicionar, no método de classe `blockSpecs` da classe `ScriptableScratchMorph`, a especificação do novo bloco dentro da categoria **Control** (Código 5.12).

```
| blocks |
    blocks - #(
        'control'
            ('when %m clicked'              S    -)
            ...
            -
            ('stop script'                  s    doReturn)
            ('stop all'                     -    stopAll)
            ('import'                       c    importScript)
...
```

Código 5.12: ScriptableScratchMorph»blockSpecs

O bloco de nome **import**, adquire a forma de 'C' e pode conter uma sequência de comandos, e o seu comportamento é definido pelo método `importScript`. Desta forma passa a ser possível visualizar o bloco dentro da categoria **Control** (Figura 5.11).

Figura 5.11: Bloco **import** dentro da categoria **Control**.

É possível ajustar o posicionamento do bloco dentro da categoria para que fique ligeiramente afastado dos blocos imediatamente acima ou abaixo, por forma a destacar o novo bloco (Figura 5.12). Para isso utiliza-se o caracter '-' antes e/ou depois da especificação do bloco (Código 5.13).

```
('stop script'                  s    doReturn)
('stop all'                     -    stopAll)
-
('import'                       c    importScript)
```

Código 5.13: ScriptableScratchMorph»blockSpecs

Figura 5.12: Bloco **import** espaçado dos blocos que o precedem.

Apesar de já se poder colocar o bloco dentro da área de *scripting*, ele ainda não tem comportamento definido, pelo que não consegue executar nenhuma acção, sendo apresentada uma linha vermelha a delineá-lo para alertar sobre esse facto (Figura 5.13).

Figura 5.13: Bloco **import** sem comportamento.

Para definir o comportamento dum bloco, caso o mesmo seja um bloco comum ao **Palco** e aos *Sprites*, então o método deve ser declarado na classe `ScriptableScratchMorph`. Se um bloco for especificado apenas numa das duas subclasses e for específico apenas da entidade por ela representada, o seu método associado deve ser declarado nessa classe. Nestes casos, definir o método consiste em consultar a "visão" que o *System Browser* apresenta da parte dedicada aos métodos de instância de uma classe e encontrar um protocolo adequado para o definir. Existem diversos protocolos que agrupam os métodos associados aos blocos das diversas categorias e cujo nome reflecte qual a categoria em questão (e.g., `looks ops` agrupa os métodos de blocos da categoria **Looks**, `sound ops` faz o mesmo para a categoria **Sounds**, etc). Não existindo um protocolo de nome `control ops`, o protocolo mais adequado para inserir o método associado ao bloco **import** é o `other ops`. No entanto, o método tem de ser declarado numa classe que lhe permita o acesso directo ao empilhamento de blocos de um *script*, pois o bloco **import** influencia a execução do *script*, não é um bloco destinado a fazer alterações no *Sprite* ou no **Palco**. Essa classe é a `ScratchProcess`, a qual representa um processo que dá vida a um empilhamento de blocos. Esse processo sabe qual o próximo bloco a executar, avalia os argumentos dos blocos, trata das estruturas de controlo, etc. Esta classe tem a seguinte estrutura:

- `stackFrame`: instância de `ScratchStackFrame` que descreve o estado actual do processo;
- `readyToYield`: valor booleano que indica se o controlo deve ser cedido a outro processo;
- `errorFlag`: valor booleano que indica se foi encontrado um erro;
- `readyToTerminate`: valor booleano que indica se o método `stop` (que termina um processo) foi invocado.

Consultando os seus protocolos, verifica-se que o mais adequado para declarar o novo método é o `private-special forms`, uma vez que nele estão também contidos métodos como o `doForever` e o `doUntil`, correspondentes a blocos da mesma categoria do **import**. Da estrutura da classe, verifica-se que a variável mais relevante para este método é a `stackFrame`. Esta última é instância da classe `ScratchStackFrame`, que possui uma variável de instância, `expression`, a qual pode tomar vários valores, entre os quais um argumento de um bloco (`ArgMorph`), um bloco (`BlockMorph`) ou um conjunto de blocos para avaliar. O bloco **import** é um bloco em forma de 'C', logo instância de `CBlockMorph`. Esta classe é subclasse de `CommandBlockMorph` e esta, por sua vez, de `BlockMorph`. Assim, o primeiro passo a tomar na definição do método é obter a referência desse bloco da *stack frame* actual. De seguida é necessário colocar na *stack frame* a sequência de blocos englobada pelo **import**. O método `popStackFrame` da classe `ScratchProcess` indica na sua documentação que é responsável por fazer o *pop* da *stack frame* actual para que a próxima passe a ser a actual. Como tal, verifica-se que há dois passos a tomar: fazer *pop* da *stack frame* actual e introduzir a nova *stack frame* com o conjunto de blocos que estão aninhados no bloco **import**. Este último passo é feito recorrendo ao método `pushStackFrame:`. Este método recebe como argumento uma *stack frame* que tem de ser construída e cujo conteúdo é definido pelo método `expression:` da classe que a representa. Esse conteúdo, a sequência de blocos aninhados no bloco **import**, é obtido pelo método `firstBlockList` da classe do bloco (`CBlockMorph`). Com estes passos, obtém-se a definição do método `importScript`, associado ao bloco **import** (Código 5.14).

```
importScript
    "Runs the sequence of blocks nested within it."

    | importBlock |
    importBlock _ stackFrame expression.
    self popStackFrame.
    self pushStackFrame: (ScratchStackFrame new expression: importBlock
        firstBlockList)
```

Código 5.14: ScratchProcess»importScript

Se se testar o bloco, verifica-se que possui o efeito desejado, ou seja, executa o *script* que contém (Figura 5.14).

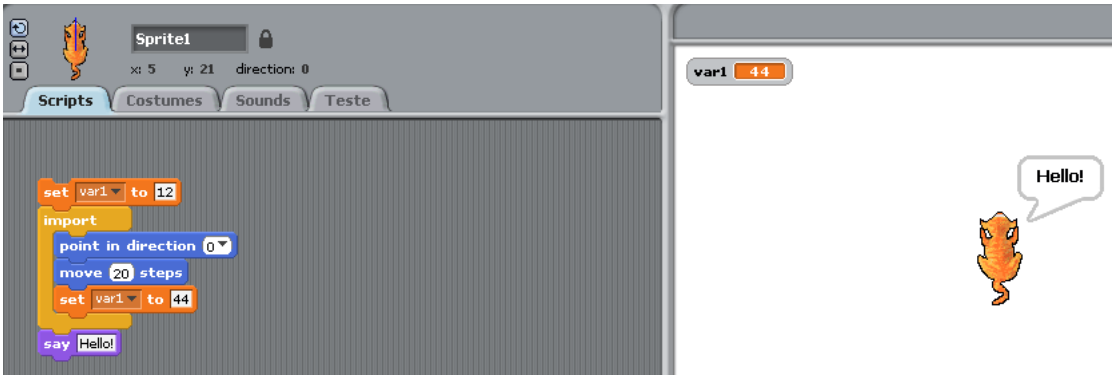


Figura 5.14: Bloco **import** com comportamento definido.

5.4.4 Teste de *scripts* na nova aba

Uma das funcionalidades pretendidas para esta etapa é a execução isolada do conjunto de blocos constituinte dum SDU (bloco) presente na categoria **MyScripts**, isto é, pretende-se que o utilizador possa visualizar e executar o corpo desse bloco. O bloco, como um todo, pode ser testado isoladamente na aba **Scripts**, mas de forma atómica, enquanto que aqui se pretende ver instrução a instrução a sua execução, permitindo observar o efeito de cada instrução, ao mesmo tempo que se podem adicionar ou remover instruções ao corpo do bloco, testando-o durante esse processo.

Estes procedimentos podem ser feitos tirando partido da aba **Test** (Secção 5.4.1). Recorde-se que esta aba define o seu conteúdo através da variável de instância `testBlocksBin` da classe `ScriptableScratchMorph`, que é inicializada com uma instância de `ScratchScriptsMorph` (um contentor de *scripts*).

Portanto, a ideia consiste em ter a possibilidade de ter uma nova bandeira, junto da bandeira verde de execução dos *scripts* da aba **Scripts**, a qual fica responsável por executar os *scripts* da aba **Test**. Para tal, torna-se importante ter um bloco semelhante ao da bandeira verde que tenha a imagem da nova bandeira no seu corpo. Este bloco está sempre presente na aba **Test** como o seu bloco de topo, tendo duas utilizações possíveis: é o primeiro bloco a ser apresentado aquando da consulta de um SDU existente na categoria **MyScripts**, ficando o corpo do *script* encaixado por baixo, ou pode simplesmente existir nessa aba, permitindo que o utilizador componha um *script* debaixo deste. Em ambos os casos, a nova bandeira permite executar isoladamente o *script*, independentemente do conteúdo da aba **Scripts**.

Como tal, o primeiro passo a tomar é a criação da nova bandeira. A criação deste elemento da interface é realizada aquando da inicialização da janela principal, mais concretamente no método `createStageButtonsPanel` que se encontra no protocolo `initialization` da classe `ScratchFrameMorph`, método este invocado pelo `initialize` da mesma classe. Neste método é criada uma variável `buttonSpecs` que contém as especificações da bandeira verde e do círculo vermelho que se encontram por cima do **Palco**. Para cada um destes elementos são definidos 3 parâmetros: o seu nome, o nome do método que lhes é associado e uma *tooltip* (legenda que indica o que o elemento faz). Como se pretende adicionar a nova bandeira, é necessário especificá-la nesta variável. Para que a bandeira seja posicionada antes da bandeira verde, a sua especificação tem de ser declarada antes da da bandeira verde (Código 5.15).

```
"buttonSpecs defines the toolbar buttons; first is icon name, second is selector"
    buttonSpecs _ #(
        "name    selector          tool tip"
        (goScript   shoutGoScript   'Start blue flag scripts')
        (go     shoutGo     'Start green flag scripts')
        (stop   stopAll     'Stop everything')).
```

Código 5.15: ScratchFrameMorph»createStageButtonsPanel

Deste modo, a primeira linha de especificação indica que a nova bandeira tem o nome de `goScript`, tem a si associado o método `shoutGoScript` (ainda por definir) e serve para iniciar a execução dos *scripts* activados pela nova bandeira (de cor azul).

O passo seguinte consiste em desenhar a bandeira na interface. Olhando para o código do mesmo método verifica-se que após a definição deste *array* de especificações, o algoritmo, de seguida, processa cada uma das entradas do *array* (Código 5.16).

```
buttonSpecs do: [:spec |
        bName _ spec first.
        button _ ToggleButton
            onForm: (ScratchFrameMorph skinAt: (bName, 'ButtonGrayPressed')
                asSymbol)
            offForm: (ScratchFrameMorph skinAt: (bName, 'ButtonGray') asSymbol)
            overForm: (ScratchFrameMorph skinAt: (bName, 'ButtonGrayPressed')
                asSymbol).
```

Código 5.16: ScratchFrameMorph»createStageButtonsPanel

Primeiro, guarda numa variável o nome do botão (a bandeira é representada por um botão com uma imagem). Em segundo lugar, cria o botão usando a classe `ToggleButton` e passando valores aos parâmetros `onForm`, `offForm` e `overForm`. Estes parâmetros representam as diferentes imagens que o botão da bandeira deve apresentar em função das diferentes interacções com o utilizador que este suporta. Estas imagens são carregadas a partir da *skin* do Scratch (`ScratchSkin`), e são identificadas nessa pasta através de um nome que resulta da concatenação do nome do botão com as *strings* "`ButtonGrayPressed`" e "`ButtonGray`". Como tal, é necessário que exista na pasta `ScratchSkin` um mesmo conjunto de imagens representativas da nova bandeira semelhantes às da bandeira verde, variando apenas o nome e a cor. As imagens encontradas para a bandeira verde são:

- `goButton.gif`
- `goButtonBlack.gif`
- `goButtonBlackPressed.gif`
- `goButtonGray.gif`
- `goButtonGrayPressed.gif`
- `goButtonPressed.gif`

De notar que todas elas têm um prefixo comum - "`go`" - que corresponde ao nome dado ao botão da bandeira verde no *array* de especificações. Portanto, é preciso criar um conjunto idêntico para a nova bandeira. Através de um programa de edição de imagem criam-se cópias das imagens referidas com uma cor diferente (azul clara, dada pelos valores (H,S,V) = (184,83,87) ou (R,G,B) = (38,210,222)). Ao renomear as imagens, é necessário dar-lhes o prefixo "`goScript`", conforme especificado no código, dando origem às imagens `goScriptButton.gif`, `goScriptButtonBlack.gif`, etc. O algoritmo prossegue associando o método e a *tooltip* ao novo botão criado e adicionando-o ao painel de botões. O algoritmo termina criando um *morph* transparente que permite espaçar os botões criados. Para o botão da bandeira verde é executado o conjunto de instruções presentes no Código 5.17.

```
bName = #go ifTrue: [
            flagButton _ button.
            stageButtonsPanel addMorphBack: (Morph new color: Color transparent;
                                             extent: 2@5)].
```

Código 5.17: ScratchFrameMorph»createStageButtonsPanel

A variável `flagButton` é uma variável de instância que identifica o botão da bandeira verde e que é utilizada noutros métodos para reflectir mudanças na bandeira em função da interacção do utilizador. Como tal é necessário criar uma nova variável de instância para a nova bandeira e replicar o código anteriormente citado, mas afectando a nova variável. Assim cria-se a variável de nome `scriptFlagButton` e adicionam-se as instruções do Código 5.18.

```
bName = #goScript ifTrue: [
            scriptFlagButton _ button.
            stageButtonsPanel addMorphBack: (Morph new color: Color transparent;
                                             extent: 2@5)]].
```

Código 5.18: ScratchFrameMorph»createStageButtonsPanel

Antes de se poder visualizar a bandeira na interface é preciso recarregar a *skin* do Scratch, abrindo um `Workspace` (uma área de trabalho onde se podem executar métodos) e executando o Código 5.19.

```
ScratchFrameMorph readSkinFrom: (FileDirectory default directoryNamed: '
    ScratchSkin')
```

Código 5.19: Carregamento da *skin*.

O resultado fica imediatamente visível (Figura 5.15).

Tendo a bandeira já pronta, falta só adicionar-lhe comportamento, ou seja, definir o seu método associado: `shoutGoScript`. Pretende-se obter um efeito semelhante ao da bandeira verde que, quando accionada, executa o *script* da aba **Scripts** cujo primeiro bloco é o bloco do tipo *hat* proveniente da categoria **Control**, cujo texto presente no seu corpo é **when [green**

Figura 5.15: Nova bandeira.

flag] clicked, onde [**green flag**] é a imagem da bandeira verde. Para isso tem de se compreender como é feita toda a interacção desde o clique na bandeira verde até à execução do *script*.

Tal como visto anteriormente, a bandeira verde tem a si associado o método `shoutGo`, presente no protocolo `menu/button actions` da classe `ScratchFrameMorph` (Código 5.20).

```
shoutGo
    "Broadcasts the start event to all objects and processes."

    self stopAll.
    workPane broadcastEventNamed: 'Scratch-StartClicked' with: 0.
    flagButton on.
    World displayWorldSafely.  "force button flash"
    Delay waitMSecs: 20.
```

Código 5.20: ScratchFrameMorph»shoutGo

Este método faz *broadcast* de um evento de nome `Scratch-StartClicked` com o valor 0. Este método é invocado sobre a variável de instância `workPane`, a qual é inicializada no método `createBasicPanes`, da mesma classe, com uma instância de `ScratchStageMorph` (Código 5.21).

```
createBasicPanes
   "Create and add my palette (viewer), script editor, stage, and library panes."
...
   workPane _ ScratchStageMorph new extent: WorkpaneExtent.
...
```

Código 5.21: ScratchFrameMorph»createBasicPanes

Consultando o método `ScratchStageMorph»broadcastEventNamed:with:`, presente no protocolo `scratch processes/events`, verifica-se que ele faz *broadcast* de um `ScratchEvent` com um dado nome e valor para todos os objectos Scratch e devolve uma colecção dos novos processos criados. Para tal, pesquisa por todos os *scripts* que respondam ao evento, iniciando novos processos para aqueles que ainda não estejam em execução (Código 5.22).

```
broadcastEventNamed: name with: value
    | event objList newProcs |
    scratchServer ifNotNil: [scratchServer queueBroadcast: name].
    event _ ScratchEvent new name: name argument: value.
    newProcs _ OrderedCollection new.

    "start scripts"
    objList _ submorphs select: [:m | m isKindOf: ScriptableScratchMorph].
    objList do: [:obj |
        newProcs addAll: (obj eventReceived: event)].
    newProcs addAll: (self eventReceived: event).

   ^ newProcs asArray
```

Código 5.22: ScratchStageMorph»broadcastEventNamed:with:

É criada uma instância da classe `ScratchEvent`, a classe que representa um evento do Scratch, com os valores recebidos pelo método. De seguida seleccionam-se todos os *submorphs* do **Palco** que sejam instâncias da classe `ScriptableScratchMorph` ou de uma das suas subclasses (é este

o comportamento definido pelo método `Object»isKindOf:`). Os *submorphs* que verificam essa condição são todos os *Sprites* presentes no **Palco**, uma vez que a classe que representa os *Sprites* é subclasse de `ScriptableScratchMorph`. Sobre as instâncias recolhidas é invocado o método `ScriptableScratchMorph»eventReceived:`. No fim, o mesmo método é invocado sobre o **Palco**, visto que é possível definir *scripts* afectos apenas ao **Palco**. Este sabe responder ao método visto ser também subclasse de `ScriptableScratchMorph`.

O método `eventReceived:` inicia todos os *scripts* cujo primeiro bloco é um bloco do tipo *hat* activado pelo evento dado como argumento do método e devolve uma colecção de novos processos. Se um *script* já tiver um processo a si associado a executar, não é criado um novo (Código 5.23).

```
eventReceived: event
    "Start all non-running stacks with an EventHat matching the given events and
        answer a collection of the new processes. If a process is already running
        for a given stack, don't start a new one."

    | targetScripts newProcs |
    targetScripts _ #().
    event name = 'Scratch-KeyPressedEvent'
        ifTrue: [
            targetScripts _ self scripts select: [:s |
                (s class == KeyEventHatMorph) and:
                 [s respondsToKeyEvent: event argument]]]
        ifFalse: [
            event name = 'Scratch-MouseClickEvent'
                ifTrue: [
                    self isHidden not ifTrue: [
                        targetScripts _ self scripts select: [:s |
                            s class == MouseClickEventHatMorph]]]
                ifFalse: [
                    targetScripts _ self scripts select: [:s |
                        (s class == EventHatMorph) and:
                         [s eventName caseInsensitiveEqual: event name]]]].

    newProcs _ targetScripts asArray collect: [:script | script startForEvent:
        event].
    ^ newProcs select: [:p | p notNil]
```

Código 5.23: ScriptableScratchMorph»eventReceived:

Neste método são desencadeadas acções diferentes conforme o evento recebido: evento do teclado (`Scratch-KeyPressedEvent`), do rato (`Scratch-MouseClickEvent`) ou outro. Em cada um dos casos, é invocado o método `scripts` da classe `ScriptableScratchMorph` que devolve os *scripts* presentes na aba **Scripts**, isto é, a colecção dos `HatBlockMorph`s lá existentes (Código 5.24).

```
scripts
    "Answer my scripts, a collection of HatBlockMorphs."

    (blocksBin isKindOf: Morph) ifFalse: [^ blocksBin].
    ^ blocksBin submorphs select: [:m | m isKindOf: HatBlockMorph]
```

Código 5.24: ScriptableScratchMorph»scripts

É importante verificar que este método devolve os *scripts* como sendo uma colecção de blocos do tipo *hat* (`HatBlockMorph`), que são usados como primeiro bloco dos *scripts*, permitindo que a partir destes se aceda ao bloco imediatamente abaixo, e assim sucessivamente. No conjunto de blocos da Figura 5.16, o primeiro bloco é um `EventHatMorph` (subclasse de `HatBlockMorph`)

que tem como *submorph*, entre outros, uma instância de `CommandBlockMorph` que representa o bloco de cor azul imediatamente abaixo. Por sua vez, este possui um *submorph* do tipo `CommandBlockMorph` que representa o bloco de cor roxa. O método `BlockMorph»blockSequence` devolve a sequência de blocos existente a partir dum dado bloco.

Figura 5.16: Exemplo de um *script*.

Dos *scripts* devolvidos pelo método `ScriptableScratchMorph»scripts` são seleccionados aqueles que representam instâncias da classe `EventHatMorph` ou de uma das suas subclasses, de acordo com o tipo de evento que representam. No final é criado um processo para cada *script* que é iniciado, através do método `EventHatMorph»startForEvent:`.

Estando percebido todo este processo, chega-se à altura de definir como é feito todo este processo para a nova bandeira. Para criar o novo bloco associado à bandeira azul é necessário perceber como se cria o bloco **when [green flag] clicked** (Figura 5.17).

Figura 5.17: Bloco associado à bandeira verde.

Os blocos são colocados na palete pela classe `ScriptableScratchMorph`, através do seu método `viewerPageForCategory:`, que devolve um *morph* contendo os blocos de uma determinada categoria, para ser colocado no `ScratchViewerMorph` (componente da interface). Para aceder aos blocos de uma dada categoria, este método invoca o método `blocksFor:` da mesma classe, que devolve uma colecção dos blocos da categoria passada como parâmetro. Este começa por invocar o método de classe `blockColorFor:` para obter a cor da categoria que está a processar. De seguida, invoca o método de classe `blockSpecs` onde estão definidas as especificações dos blocos, para de seguida as processar. Para cada uma das especificações é invocado o método `blockFromSpec:color:`. Este último é responsável por criar um bloco a partir da sua especificação. É neste método que se faz a associação entre a letra que indica o tipo de bloco na sua especificação e a classe (representativa do bloco) que lhe fica associada (Código 5.25).

```
blockFromSpec: spec color: blockColor
    "Create a block from the given block specification. Answer nil if I don't
    implement the block spec selector."

    | blockLabelSpec blockType selector defaultArgs block rcvr argPermutation |
    blockLabelSpec _ ScratchTranslator translationFor: (spec at: 1).
    argPermutation _ CommandBlockMorph argPermutationForSpec: (spec at: 1)
                                       withTranslation: blockLabelSpec.
    blockType _ spec at: 2.
    selector _ (spec at: 3) asSymbol.
    defaultArgs _ self defaultArgsFor: spec.

    (#(E K M S W) includes: blockType) ifTrue: [
        ^ (self hatBlockType: blockType) color: blockColor].
...
```

Código 5.25: ScriptableScratchMorph»blockFromSpec:color:

Neste método verifica-se se o tipo do bloco definido na especificação é dado por alguma das letras do conjunto {'E','K','M','S','W'}. Caso seja, então invoca-se o método `hatBlockType:` (Código 5.26).

```
hatBlockType: blockType

    | stage evtName |
    'E' = blockType ifTrue: [
        evtName _ ''.
        (stage _ self ownerThatIsA: ScratchStageMorph)
            ifNotNil: [evtName _ stage defaultEventName].
        ^ EventHatMorph new scriptOwner: self; eventName: evtName].

    'K' = blockType ifTrue: [^ KeyEventHatMorph new scriptOwner: self].
    'M' = blockType ifTrue: [^ MouseClickEventHatMorph new scriptOwner: self].
    'S' = blockType ifTrue: [^ EventHatMorph new forStartEvent scriptOwner: self].
    'W' = blockType ifTrue: [^ WhenHatBlockMorph new scriptOwner: self].
```

Código 5.26: ScriptableScratchMorph»hatBlockType:

Olhando para este código e para a hierarquia dos `HatBlockMorph`s (Figura A.13 do Anexo A), verifica-se que é este o método responsável por definir um bloco do tipo *hat* (todas as classes envolvidas são sublcasses da classe `HatBlockMorph`). Para se saber qual é a classe que cria o bloco associado à bandeira verde, é preciso voltar a consultar o método de classe `blockSpecs` da classe `ScriptableScratchMorph` e ver qual a letra identificativa do tipo desse bloco, presente na categoria **Control** (Código 5.27).

```
blockSpecs
...
'control'
            ('when %m clicked'                  S    -)
```

Código 5.27: ScriptableScratchMorph»blockSpecs

Assim, verifica-se que a classe que cria o bloco em questão é a `EventHatMorph`. No método `hatBlockType:` constata-se que, após criar uma nova instância de `EventHatMorph`, esta recebe a mensagem `forStartEvent` (Código 5.28).

```
forStartEvent

    | parts s m |
    super initialize.
    self removeAllMorphs.
    parts _ ScratchTranslator labelPartsFor: 'when %m clicked'.

    s _ StringMorph new contents: parts first; font: (ScratchFrameMorph getFont: #
        Label); color: Color white.
    self addMorphBack: s.

    m _ ImageMorph new form: (ScratchFrameMorph skinAt: #goButton).
    self addMorphBack: m.

    s _ s fullCopy contents: parts second.
    self addMorphBack: s.

    "create scriptNameMorph but don't add it"
    scriptNameMorph _ EventTitleMorph new eventName: 'Scratch-StartClicked'.

    self fixBlockLayout.
```

Código 5.28: EventHatMorph»forStartEvent

Este método pega na *string* "**when %m clicked**" e parte-a para a transformar na *string* apresentada no corpo do bloco. Pode-se verificar que a imagem da bandeira verde está a ser carregada da *skin*. A penúltima instrução atribui à variável de instância `scriptNameMorph` da classe `HatBlockMorph` uma instância de `EventTitleMorph`, indicando que o evento que lhe fica associado tem o nome de `Scratch-StartClicked`. É agora perceptível porque é que o bloco associado à bandeira verde responde ao evento com o mesmo nome por ela lançado.

Sabendo como criar este tipo de bloco, apenas se tem de fazer um trabalho semelhante a nível de codificação para criar o novo bloco. Este bloco tem no seu corpo o texto **when [blue flag] clicked** e deve responder ao evento `Scratch-StartScriptClicked`. O primeiro passo consiste em criar um método semelhante ao `forStartEvent`, que defina o corpo do bloco e o evento ao qual responde. Como tal, define-se o método `EventHatMorph»forStartScriptEvent` (Código 5.29).

```
forStartScriptEvent
...
    m _ ImageMorph new form: (ScratchFrameMorph skinAt: #goScriptButton).
...
    scriptNameMorph _ EventTitleMorph new eventName: 'Scratch-StartScriptClicked'.
...
```

Código 5.29: EventHatMorph»forStartScriptEvent

Este método é em tudo semelhante ao `forStartEvent`, só que carrega da *skin* a imagem identificada pelo símbolo `#goScriptButton`, e o evento `Scratch-StartScriptClicked` fica associado ao `EventTitleMorph`.

De seguida é preciso indicar ao método `ScriptableScratchMorph»hatBlockType:` que vai ter de saber construir um novo tipo de bloco *hat*. Para isso define-se uma nova letra identificativa, como o 'X' (de "*execute*"), e associa-se a esta letra a criação de um `EventHatMorph` capaz de responder ao novo evento (Código 5.30).

```
hatBlockType: blockType
...
    'X' = blockType ifTrue: [^ EventHatMorph new forStartScriptEvent scriptOwner:
        self].
```

Código 5.30: ScriptableScratchMorph»hatBlockType:

É também necessário actualizar o método `ScriptableScratchMorph»blockFromSpec:color:` com a nova letra (Código 5.31).

```
blockFromSpec: spec color: blockColor
...
    (#(E K M S W X) includes: blockType) ifTrue: [
        ^ (self hatBlockType: blockType) color: blockColor].
...
```

Código 5.31: ScriptableScratchMorph»blockFromSpec:color:

Posteriormente codifica-se o método associado à bandeira azul para fazer *unicast* do evento `Scratch-StartScriptClicked`. Desta forma, na classe `ScratchFrameMorph` define-se o método `shoutGoScript`, dentro do protocolo `menu/button actions` (Código 5.32).

```
shoutGoScript

    self stopAll.
    workPane unicastEventNamed: 'Scratch-StartScriptClicked' with: 0.
    scriptFlagButton on.
    World displayWorldSafely.
    Delay waitMSecs: 20.
```

Código 5.32: ScratchFrameMorph»shoutGoScript

Neste método é preciso activar a variável que referencia a bandeira azul (`scriptFlagButton`). De notar que é feito *unicast* (envio de uma mensagem a apenas um objecto) e não *broadcast* (envio para vários objectos), pois pretende-se obter o efeito de execução isolada do *script* sobre o objecto que se está a manipular, e.g., numa situação em que existam 2 *Sprites* no **Palco**, deve poder-se testar individualmente um *script* diferente para cada um deles (os dois *Sprites* mais o **Palco**). Logo, o evento tem de ser recebido apenas pelo objecto seleccionado e não por todos (pois ocorreria a execução simultânea dos seus *scripts*). É isso que faz o método `ScratchStageMorph»unicastEventNamed:with:` (Código 5.33).

```
unicastEventNamed: name with: value
   "Unicasts a ScratchEvent with given name and argument value to the currently
   selected Scratch object and answers a newly created process. This is done by
   finding its public script that responds to this event, and starting a new
   process if it's not already running."

   | event newProc scratchLibrary spriteThumbnails thumb |
   scratchServer ifNotNil: [scratchServer queueBroadcast: name].
   event _ ScratchEvent new name: name argument: value.
   scratchLibrary _ (self ownerThatIsA: ScratchFrameMorph) libraryPane.
   spriteThumbnails _ scratchLibrary spriteThumbnails.
   (thumb _ spriteThumbnails detect: [:tb | tb isSelected] ifNone: [nil])
       ifNotNil: [
            newProc _ thumb target eventReceived: event]
        ifNil: [ newProc _ self eventReceived: event].
     ^ newProc
```

Código 5.33: ScratchStageMorph»unicastEventNamed:with:

Este método começa por criar o evento Scratch e de seguida acede à `ScratchLibraryMorph`, onde estão situados os ícones do **Palco** e dos *Sprites*. Aí, acede-se à lista dos ícones dos *Sprites* e procura-se pelo que está seleccionado (que é o que representa o objecto para o qual se está a desenvolver/testar o *script*). Para determinar qual está seleccionado é necessário obter o valor booleano da variável de instância `isSelected` da classe `LibraryItemMorph`, e isto obriga a que se defina o seu método selector nessa classe, no protocolo `accessing`. Ao ser encontrado o *Sprite* seleccionado, acede-se à instância de `ScratchSpriteMorph` que o representa através do método `target`, e sobre esta invoca-se o método `eventReceived:`. Se não for encontrado, significa que é o **Palco** que está seleccionado, logo o método deverá ser invocado sobre este. Por fim devolve-se o único processo criado (contrariamente ao método de *broadcast*, que criava uma lista de processos).

Antes de passar à fase de adicionar o bloco à área de *scripting* da aba **Test** é preciso ainda ajustar uns métodos. No método `ScriptableScratchMorph»eventReceived:` estão a ser processados apenas os *scripts* existentes na tab **Scripts**, uma vez que está a ser invocado o método `scripts`, que devolve os *morphs* presentes no contentor de blocos desta aba. Como tal, é preciso realizar um teste para verificar qual a aba activa. Para a encontrar é preciso aceder ao componente da interface que a integra, ou seja, ao `ScratchScriptEditorMorph`. Para isso pode-se, a partir do `ScriptableScratchMorph`, percorrer a cadeia dos *morphs* que são seus

owners até encontrar um que seja do tipo `ScratchFrameMorph` (janela principal do Scratch). Isto é feito através do método `Morph»ownerThatIsA:`. Tendo esta referência, consegue-se aceder ao `ScratchScriptEditorMorph` através da variável de instância `scriptsPane`, que guarda a referência para o editor de *scripts*. Já na posse do editor de *scripts*, testa-se se a aba activa é a **Test**. Se for, então tem de se encontrar os *scripts* definidos na sua área de *scripting*. Caso contrário, encontram-se os *script* da aba **Scripts**. Para encontrar os *scripts* da aba **Test** é necessário definir o método `ScriptableScratchMorph»singleScript:`, dentro do protocolo `scripts`, por forma a aceder ao contentor de *scripts* desta aba e devolver todos os blocos do tipo *hat* (Código 5.34).

```
singleScript

    (testBlocksBin isKindOf: Morph) ifFalse: [^ testBlocksBin].
    ^ testBlocksBin submorphs select: [:m | m isKindOf: HatBlockMorph]
```

Código 5.34: ScriptableScratchMorph»singleScript:

Depois faz-se então o teste no `ScriptableScratchMorph»eventReceived:`, criando uma variável `scpts` que vai guardar os *scripts* e que vai substituir as anteriores chamadas ao método `scripts` (Código 5.35).

```
eventReceived: event
    | targetScripts newProcs ssem scpts |
    targetScripts _ #().
    ssem _ (self ownerThatIsA: ScratchFrameMorph) scriptsPane.
    ssem currentCategory = 'Test'
       ifTrue: [scpts _ self singleScript]
       ifFalse: [scpts _ self scripts]].
...
```

Código 5.35: ScriptableScratchMorph»eventReceived:

Tendo isto pronto, é altura de adicionar o novo bloco como sendo o primeiro a figurar na área de *scripting* da aba **Test**. Para isso, adiciona-se uma nova variável de instância à classe `ScriptableScratchMorph`, de nome `scriptBlock`, e define-se o seu método selector e modificador no protocolo `blocks` da mesma classe (neste protocolo estão agrupados vários métodos relacionados com *scripts* e blocos). Após a inicialização do contentor de *scripts* da aba **Test** no método `initialize` da classe `ScriptableScratchMorph`, cria-se um novo bloco tirando partido dos métodos de classe `blockFromSpec:color:` e `blockColorFor:`, já anteriormente apresentados. Ao primeiro método passam-se 2 parâmetros: a especificação do novo bloco e a cor, a qual, por sua vez, é determinada enviando ao segundo método a string "`control`". Por fim adiciona-se o bloco ao contentor e alinha-se o *script* verticalmente através do método `ScratchScriptsMorph»cleanUp` (Código 5.36).

```
initialize
...
    testBlocksBin _ ScratchScriptsMorph new.
    scriptBlock _ (self blockFromSpec: #('when %m clicked' X -)
                  color: (self class blockColorFor: 'control')).
    testBlocksBin addMorph: scriptBlock; cleanUp.
...
```

Código 5.36: ScriptableScratchMorph»initialize

A partir deste momento, quando se inicia o Scratch, já é possível visualizar o novo bloco na aba **Test** (Figura 5.18).

Figura 5.18: Novo bloco na aba **Test**.

No entanto, quando se tenta testar o bloco, ele não responde nem aos eventos de clique do rato sobre o bloco nem ao evento gerado pela bandeira azul. Isto deve-se ao facto do objecto ao qual o bloco (e, consequentemente, o *script* que nele pode ser encaixado) está associado não estar correctamente construído na composição de *morphs* da interface. Este objecto é o *Sprite* ou o **Palco**, dados pela variável `scriptOwner` do bloco. A solução passa por alterar os valores relativos ao `scriptOwner` aquando da detecção de qual a aba activa. Isso é feito no método `ScratchScriptEditorMorph»categoryChanged:`, o qual invoca o método `currentCategory:` da mesma classe (que é o mesmo método onde se associa o conteúdo da aba **Test** ao contentor de *scripts*), e é neste método que se tem de proceder à alteração. A solução passa por obter a referência do bloco existente na área de *scripting* da aba **Test** e, caso o evento associado a este bloco seja o `Scratch-StartScriptClicked`, então altera-se o seu `scriptOwner` (Código 5.37).

```
currentCategory: aString
...
    currentCategory = 'Test' ifTrue: [
        pageViewerMorph contents: self target testBlocksBin.
        eventHat _ self target scriptBlock.
        eventHat eventName = 'Scratch-StartScriptClicked'
            ifTrue: [eventHat newScriptOwner: self target].
        self target scriptBlock: (self target testBlocksBin submorphs detect: [:m
            | m isKindOf: EventHatMorph])].
...
```

Código 5.37: ScratchScriptEditorMorph»currentCategory:

A referência ao *Sprite* ou **Palco** é obtida através do método `target`, logo o `scriptOwner` é afectado sempre com o valor correcto, em função do contexto actual. A última instrução apresentada força a que o bloco associado à bandeira azul referenciado pelo *Sprite*/**Palco** seja o mesmo que surge na área de *scripting* da aba **Test** (uma vez que o processo de inicialização cria, na aba **Test**, um bloco diferente daquele que inicialmente fica associado ao *Sprite*/**Palco**). Assim, passa a ser possível criar um *script* e executá-lo na aba **Test** recorrendo à bandeira azul (Figura 5.19).

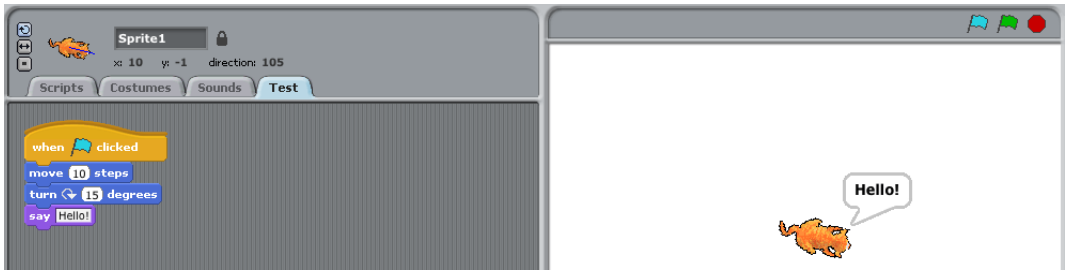


Figura 5.19: *Script* desenvolvido e executado na aba **Test**.

No entanto, ainda é preciso resolver uns pequenos pormenores. Uma vez que o novo bloco não deve figurar na palete, pois o seu único propósito é existir na área de *scripting* da aba **Test**, é preciso impedir que o mesmo seja arrastado para a palete ou para as categorias, pois caso isso aconteça, o bloco, por omissão, é apagado da área de *scripting*. Para efectuar esta tarefa tem de se alterar o método `justDroppedInto:event:`, presente no protocolo `dropping/grabbing` da classe `BlockMorph`. Nele é feito um teste onde se verifica se o *owner* do bloco é um `ScratchViewerMorph` (zona da interface onde reside a palete e as categorias) e que apaga o bloco quando largado sobre essa zona. Assim, tem de se alterar esta porção de código para impedir que o novo bloco siga esse comportamento. Deste modo, é preciso aceder ao editor de *scripts* para saber qual a aba activa, guardando-a na variável local criada para o efeito, `currentCategory`. Caso seja a aba **Test** e o bloco seja uma instância da classe `EventHatMorph`, testa-se se o evento associado ao mesmo é o `Scratch-StartScriptClicked`. Em caso afirmativo, impede-se que o bloco seja largado (Código 5.38).

```
justDroppedInto: newOwner event: evt
    "Handle being dropped into a new situation."
...
    ((self ownerThatIsA: ScratchViewerMorph) notNil) ifTrue: [
        currentCategory _ (self ownerThatIsA: ScratchFrameMorph)
                              scriptsPane currentCategory.
        (currentCategory = 'Test' & self isMemberOf: EventHatMorph) ifTrue: [
            self eventName = 'Scratch-StartScriptClicked'
               ifTrue: [^ self rejectDropEvent: evt]].
        "delete myself when dropped in the blocks palette area"
        self delete.
        self receiver blocksBin changed.
        ^ self].
```

Código 5.38: BlockMorph»justDroppedInto:event:

Aqui, usa-se o método `Object»isMemberOf:`, que verifica se o receptor da mensagem é instância da classe passada como argumento (exactamente desta classe e não da classe ou de uma das suas subclasses como no método `isKindOf:`), porque se sabe com certeza qual o tipo exacto do bloco.

Outra alteração que é preciso fazer no mesmo método consiste em impedir que se arrastem blocos do tipo *hat* para a aba **Test**, uma vez que se pretende que o único bloco do tipo *hat* que lá exista seja o bloco associado à bandeira azul. É, portanto, necessário criar um teste para o caso em que um bloco é largado na área de *scripting* (`ScratchScriptsMorph`). Mais uma vez, verifica-se se a aba activa é a **Test**. Se for, e caso o bloco seja instância da classe `HatBlockMorph` ou de uma das suas subclasses (necessário pois todos os blocos do tipo *hat* existentes verificam esta condição), então tem que se verificar se o mesmo não é um `EventHatMorph` com o evento `Scratch-StartScriptClicked` associado. Caso não seja, impede-se que o bloco seja largado (Código 5.39).

```
(owner isKindOf: ScratchScriptsMorph) ifTrue: [
        currentCategory _ (self ownerThatIsA: ScratchFrameMorph)
                              scriptsPane currentCategory.
        (currentCategory = 'Test' & self isKindOf: HatBlockMorph)
            ifTrue: [(((self isMemberOf: EventHatMorph) &
                       (self eventName = 'Scratch-StartScriptClicked')) not)
                          ifTrue: [^ self rejectDropEvent: evt]]].
```

Código 5.39: BlockMorph»justDroppedInto:event:

A última condição é necessária para prevenir o caso em que se faz a acção `delete/Undelete` sobre o bloco, pois sem esse teste, é impossível recolocar o bloco na área de *scripting*.

Outro pormenor a ajustar tem a ver com a forma como a bandeira azul sinaliza que está a ser executado um *script* na aba **Test**. O que acontece é que após o clique na bandeira azul, a bandeira verde fica activa e a azul inactiva, enquanto é executado o *script*. Fazendo uma pesquisa pela variável da classe `ScratchFrameMorph` associada à bandeira verde, `flagButton`, verifica-se que esta é activada e desactivada no método `step` da mesma classe, nomeadamente aquando da verificação da existência de processos para execução. Portanto, é neste ponto que é necessário fazer alterações. Se houver processos ainda para executar, mais uma vez se obtém a referência para a aba actual, e caso seja a aba **Test**, a bandeira azul (representada pela variável de instância `scriptFlagButton`) é activada. Se não for (ou seja, significa que se está numa das outras 3 abas, logo o processo a ser executado diz respeito a um *script* existente na aba **Scripts**), então a bandeira azul é desactivada e a bandeira verde é activada. Quando já não houver mais processos para executar, ambas são desactivadas (Código 5.40).

```
step
    "Run each process until it gives up control, then filter out any processes
    that have terminated."
...
   workPane processesToRun size > 0
        ifTrue: [
            currentCategory _ self scriptsPane currentCategory.
            currentCategory = 'Test' ifTrue: [scriptFlagButton on]
                                     ifFalse: [scriptFlagButton off. flagButton on
                                         ]]
         ifFalse: [scriptFlagButton off. flagButton off].
```

Código 5.40: ScratchFrameMorph»step

Falta também resolver um problema resultante da execução do projecto em modo de apresentação. Quando se tenta executar neste modo ocorre um erro derivado do facto de se estar a tentar aceder a um `ScratchFrameMorph` no método `ScriptableScratchMorph»eventReceived:`. Isto ocorre porque em modo de apresentação não existe nenhum `ScratchFrameMorph` no Scratch, mas antes um `ScratchPresenterMorph`. Para resolver este problema, tem de se fazer um teste no método assinalado para verificar se existe alguma instância de `ScratchPresenterMorph` activa. Caso exista (significa que se está em modo de apresentação), são carregados os *scripts* da aba **Scripts** invocando o método `scripts`. Caso contrário significa que se está em modo de desenvolvimento e teste dos *scripts*, logo sabe-se à partida que existe uma instância de `ScratchFrameMorph`, portanto pode utilizar-se o código já existente para carregar os *scripts* adequados, em função da aba activa (Código 5.41).

```
eventReceived: event

    | targetScripts newProcs ssem scpts |
    targetScripts _ #().
    (World findA: ScratchPresenterMorph)
        ifNotNil: [scpts _ self scripts]
        ifNil: [
            ssem _ (self ownerThatIsA: ScratchFrameMorph) scriptsPane.
            ssem currentCategory = 'Test'
                ifTrue: [scpts _ self singleScript]
                ifFalse: [scpts _ self scripts]].
...
```

Código 5.41: ScriptableScratchMorph»eventReceived:

Ainda outra particularidade: o menu contextual da área de *scripting* (Figura 5.20) não surge na aba **Test**.

Figura 5.20: Menu contextual da área de *scripting*.

Este menu encontra-se definido no método `ScratchScriptEditorMorph»scriptsMenu:`. Existem dois métodos que o invocam: `mouseDown:` e `mouseHold:`, da classe `ScratchScriptsMorph`. Ambos fazem um teste antes de invocar o menu onde verificam se a aba actual é a **Scripts**, portanto a solução consiste em adicionar, em ambos, uma condição que também verifica se a aba é a **Test** (Código 5.42).

```
(m target notNil and: [m currentCategory = 'Scripts' or: [m currentCategory = '
    Test']])
            ifTrue: [m scriptsMenu: evt hand position].
```

Código 5.42: ScratchScriptsMorph»mouseDown:

É preciso ainda impedir que o bloco da nova aba seja apagado via opção `delete` do menu contextual do bloco. Este menu está definido no método `BlockMorph»rightButtonMenu`. A condição que a adiciona ao menu faz com que a opção `delete` esteja apenas disponível para blocos que estão no topo do *script* ou sozinhos (o seu *owner* é o `ScratchScriptsMorph` que representa a área de *scripting*) e não para blocos que estão no meio ou fim (cujo *owner* é o bloco que se encontra imediatamente acima). Como se pretende alterar as situações em que esta opção está disponível para o bloco do tipo *hat* da aba **Test**, a melhor solução passa por redefinir este método na classe `HatBlockMorph`, seguindo o método a mesma estrutura do método da superclasse. À condição supracitada é preciso adicionar uma outra que verifica se o bloco sobre o qual se está a invocar o menu contextual é o bloco do tipo *hat* definido para existir na aba **Test**, e que é referenciado pela variável `scriptBlock` da classe `ScriptableScratchMorph` (cuja instância é obtida pela invocação do método `receiver`) (Código 5.43).

```
rightButtonMenu
...
    (owner isKindOf: ScratchBlockPaletteMorph) ifFalse: [
        ...
        (self owner isKindOf: BlockMorph) ifFalse: [ "we can't yet delete a
            blocks inside a script"
            (self receiver scriptBlock == self) ifFalse: [
                menu add: 'delete' action: #delete]].
...
```

Código 5.43: HatBlockMorph»rightButtonMenu

De notar o uso do método `==` que testa a identidade entre objectos, ou seja, para este caso, testa se o objecto que representa o bloco da aba **Test** é exactamente aquele sobre o qual se está a invocar o menu contextual.

Neste momento, a nova aba e o seu bloco já se encontram prontos para que, ao consultar o corpo de um SDU, os blocos seus constituintes sejam empilhados debaixo deste bloco, sendo possível executá-los.

5.4.5 Clonagem do conteúdo da nova aba

A clonagem de conteúdo da aba **Test** entre *Sprites*, ou entre um *Sprite* e o **Palco**, é uma funcionalidade que se pretende ver desactivada, uma vez que esta aba serve apenas para consulta,

teste e eventual alteração de SDUs. Há duas situações em que esta clonagem pode acontecer:

1. Quando se arrastam blocos da aba **Test** de um *Sprite* para cima do ícone de outro *Sprite* ou do **Palco**, localizado na zona da interface representada por um `ScratchLibraryMorph` (Figura 5.21);
2. Quando se duplica um *Sprite* através da opção **duplicate** disponível no menu contextual presente quer no ícone do *Sprite* na zona da interface supracitada, quer na imagem do *Sprite* presente no **Palco** (Figuras 5.22a e 5.22b).

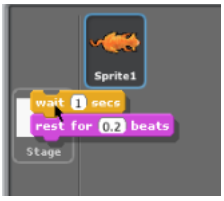
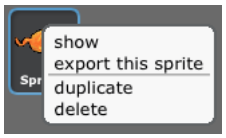
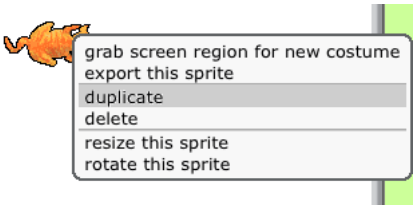


Figura 5.21: Blocos arrastados para cima do ícone do **Palco**.



(a) Menu contextual do ícone do *Sprite*.



(b) Menu contextual do *Sprite* presente no **Palco**.

Figura 5.22: Menus contextuais do *Sprite*, com opção de duplicação.

É preciso, portanto, alterar o comportamento existente para estas situações. Numa nota breve, é de referir que o restante código de suporte às explicações fornecidas neste capítulo passa a estar disponível no Anexo C.

Para a primeira situação, o comportamento definido por omissão faz com que, ao arrastar blocos da aba **Test** para cima do ícone de outro *Sprite* ou do **Palco**, estes sejam copiados para a aba **Scripts** deste. Para impedir qualquer cópia de *scripts* da aba **Test** realizada desta forma é necessário alterar o método `acceptDroppingMorph:event:` da classe `LibraryItemMorph` (que representa o ícone dos *Sprites* e do **Palco**). Este método apresenta comportamento diferente conforme o tipo do seu primeiro argumento (o *morph* que se está a largar sobre o ícone). Os tipos testados para o argumento são `MediaItemMorph`, `BlockMorph` e `ScratchCommentMorph`. Interessa, portanto, alterar o comportamento existente para os dois últimos (o primeiro diz respeito aos elementos *media*). Em ambos os casos, o comportamento definido é semelhante: através do método `target` acede-se ao **Palco** ou ao *Sprite* correspondente ao ícone e sobre este é invocado o método `addStack:` ou `addComment:`, conforme se trate de um `BlockMorph` ou um `ScratchCommentMorph`, sendo-lhe passado como argumento uma cópia do *morph* a clonar (neste caso, blocos ou comentários, respectivamente). Os métodos `addStack:` e `addComment:`

colocam os blocos/comentários na aba **Scripts**, alinhados verticalmente a seguir ao último *script* lá existente. Como tal, apenas é necessário impedir que isto ocorra quando os *morphs* arrastados para cima do ícone sejam provenientes da aba **Test**, ou seja, quando esta é a aba actual. O método `acceptDroppingMorph:event:` fica então com a configuração presente no Código C.1.

Na segunda situação, ao aplicar a opção **duplicate** sobre um *Sprite* é criada uma cópia deste, com *scripts*, sons, variáveis, etc, próprios (independentes do primeiro *Sprite*), excepto no que toca ao conteúdo da aba **Test** que é partilhado (e.g., apagar um bloco desta aba num *Sprite* repercute-se no outro *Sprite*). O que se pretende é que ao criar o clone de um *Sprite*, as suas abas **Test** sejam independentes e que o seu conteúdo não seja copiado aquando da criação do clone. A aba **Test** do novo *Sprite* deve possuir apenas o seu bloco de topo associado à bandeira azul.

Para tal, é preciso encontrar os métodos que fazem a duplicação dos *Sprites*. Para isso faz-se uma pesquisa pelo nome da opção **duplicate**, encontrando-se duas implementações do método `rightButtonMenu`: uma na classe `ScratchSpriteMorph` e outra na `LibraryItemMorph`. O primeiro mostra o menu contextual do *Sprite* que se encontra no **Palco**, enquanto que o segundo mostra o menu contextual do ícone do *Sprite*. Para o primeiro, a acção associada à opção **duplicate** é dada pelo método com o mesmo nome, enquanto que para o segundo é dada pelo `duplicateNoAttach:`, ambos da classe `ScriptableScratchMorph`. A diferença entre estes dois métodos é que o `duplicate`, após invocar o `duplicateNoAttach` para criar a cópia do *Sprite*, executa alguns passos extra relacionados com a interacção do novo *Sprite* com o **Palco**. Portanto, o foco de atenção encontra-se no método `duplicateNoAttach` da classe `ScriptableScratchMorph` (Código C.2). Olhando para o código do método, verifica-se que a parte que constrói a cópia é dada pela invocação do método `fullCopy`. A pseudo-variável `self` faz com que se inicie uma procura dinâmica pelo método começando na classe do receptor do método (`ScriptableScratchMorph`). O método é encontrado na classe `Morph` (Código C.3). Nele é invocado o método `copyRecordingIn:`. Mais uma vez a pseudo-variável `self` faz com que a procura pelo método se inicie na classe do receptor do primeiro método iniciado nesta cadeia de invocações (`duplicateNoAttach`), ou seja, na classe `ScriptableScratchMorph`. E é nessa classe que é encontrado o método `copyRecordingIn:`. Caso não existisse o `self`, o método seria encontrado na classe `Morph` e não se encontraria o código onde é necessário fazer alterações.

Uma vez no método correcto, `ScriptableScratchMorph»copyRecordingIn:`, este informa, através da sua documentação, que é responsável por copiar os campos e *scripts* de uma instância sua (Código C.4). Neste método, no entanto, já é invocado o `Morph»copyRecordingIn:` por via da pseudo-variável `super` que faz com que a procura comece na superclasse da classe onde se encontra a invocação com o `super`. Neste caso, seria a superclasse de `ScriptableScratchMorph`, que é a `Morph`.

A parte que interessa analisar diz respeito à criação da cópia da variável que representa a área de *scripting* da aba **Scripts**, `blocksBin`, a qual fica guardada na variável `newBlocksBin`. Este processo de cópia recorre, mais uma vez, ao método `fullCopy` da classe `Morph`. Após isso são processados os blocos já existentes, sendo feita uma acção diferente conforme se tratem de `HatBlockMorph`s ou `CommandBlockMorph`s. Por fim, a variável é atribuída à nova cópia do *Sprite* através da invocação do método `ScriptableScratchMorph»vars:lists:blocksBin:` (Código C.5). Para se atingir o objectivo pretendido para esta tarefa tem que se acrescentar no método `copyRecordingIn:` código que represente o comportamento pretendido para a cópia da aba **Test** e seu conteúdo. Como tal, adiciona-se uma variável local que represente a área de *scripting* dessa aba, `newTestBlocksBin`, guardando-se nesta uma cópia do conteúdo da aba **Test**, tal como acontece com a aba **Scripts**. Desta forma, as abas **Test** do *Sprite* original e do *Sprite* cópia passam a ser independentes. No entanto, no problema a resolver pretende-se que a aba **Test** do novo *Sprite* não seja uma cópia da do *Sprite* original, devendo ter apenas o bloco de topo

associado à bandeira azul. Por isso tem que se garantir que, após a cópia, se eliminam todos os blocos que estejam acoplados por baixo deste e ainda aqueles que se encontram livres na área de *scripting* da mesma aba. Para tal, acede-se aos *submorphs* da área de *scripting*, obtendo-se a colecção de todos os blocos que estão no topo dos *scripts* dessa aba. Desta forma obtém-se o bloco associado à bandeira azul e todos os primeiros blocos de quaisquer *scripts* existentes que não estejam ligados a este bloco. Tendo essa colecção, itera-se sobre os seus elementos testando-se se são do tipo `EventHatMorph` (a classe representativa do bloco associado à bandeira azul). Caso sejam - só existe um elemento que verifica esta condição -, então é necessário efectuar 2 passos:

- Indicar que o *owner* desse bloco (e do *script* por si encabeçado) é o novo *Sprite*, para que não ocorra novamente o problema do bloco não ter comportamento;
- Eliminar todos os blocos que estão acoplados por baixo deste.

O primeiro passo consiste em replicar o código que já existia para a aba **Scripts**. No segundo é necessário aceder ao bloco que se encontra encaixado por baixo do bloco do tipo `EventHatMorph`. Isto é feito através do método `nextBlock`. É necessário testar se o resultado deste método é `nil` pois poderá não existir nenhum. Caso exista, invoca-se sobre esse bloco o método `delete` por forma a eliminá-lo. O bloco que se encontra encaixado imediatamente abaixo é um *submorph* deste bloco e assim sucessivamente para os restantes blocos de um *script*. Eliminar um bloco consiste em eliminá-lo do conjunto de *submorphs* do seu *owner*, ficando a referência para o seu *owner* com o valor `nil`, como se ficasse órfão. Desta forma não é colocado na interface aquando da sua construção, e os blocos que estavam encaixados por baixo dele (os seus *submorphs*) também não são colocados. Assim, apagam-se facilmente os blocos pretendidos. Caso hajam blocos que não são do tipo `EventHatMorph` então devem ser eliminados num processo semelhante. Para associar a nova cópia da aba **Test** ao novo *Sprite*, altera-se o método `ScriptableScratchMorph»vars:lists:blocksBin:` para que receba mais um parâmetro, que é dado pela variável `newTestBlocksBin` (Código C.6). O método `copyRecordingIn:` fica então com a configuração visível no Código C.7.

Neste momento, a clonagem de um *Sprite* mantém as suas abas **Test** independentes, ficando a aba do clone apenas com o bloco de topo associado à bandeira azul. As duas situações referidas no início da secção ficam assim resolvidas.

5.4.6 Criação de um SDU

Nesta secção reporta-se todo o trabalho desenvolvido por forma a criar uma das principais funcionalidades desta extensão do Scratch: a criação de um SDU. Recorde-se que, para efeitos de terminologia, considera-se um SDU um bloco cujo comportamento é dado por uma sequência de blocos que formam o seu corpo.

O processo de criação de um SDU consiste em, dado um empilhamento de blocos existente na área de *scripting*, seja ele encabeçado ou não por um bloco do tipo *hat*, invocar sobre ele a opção **save script** disponível através do menu contextual de um dos seus blocos. Aquando da sua criação, e tal como acontece nas variáveis e listas, o utilizador pode escolher entre criar um SDU local ou global. Se o SDU for definido para um *Sprite*, o utilizador pode escolher entre defini-lo como local a esse *Sprite* ou global a todos os *Sprites* (cada *Sprite* armazena a definição desse SDU). Se for definido para o **Palco** então é sempre local ao **Palco**. A diferença para as variáveis e listas é que, quando estas são globais, são-no para os *Sprites* e para o **Palco**, ficando armazenadas neste último. Para os SDUs não se pode seguir essa abordagem porque existem categorias de blocos, como a **Motion**, que contêm blocos para os *Sprites* mas não contêm para o

Palco, ou que têm diferentes blocos conforme se trate de um *Sprite* ou do **Palco**. A utilização, num *script* de um *Sprite*, de um bloco que só exista para o **Palco**, ou vice-versa, geraria erros.

O utilizador tem de fornecer um nome para o SDU, que não pode ser repetido (independentemente do SDU ser local ou global), à semelhança do que acontece nas variáveis e listas. Os SDUs, após serem criados, ficam acessíveis na categoria de blocos **MyScripts**. A partir daqui, podem ser usados na construção de comportamentos via bloco **import**.

Portanto, como referido, a criação de um SDU desencadeia-se pela selecção da opção **save script**, acrescentada ao menu contextual dos `HatBlockMorph`s (classe que representa os blocos do tipo *hat*) e `CommandBlockMorph`s (superclasse comum a todos os restantes tipos de blocos). Essa opção é definida no método `rightButtonMenu` destas classes. Na classe `HatBlockMorph` define-se então que esta opção está disponível apenas quando o bloco (do tipo *hat*, representado por esta classe) se encontra na aba **Scripts** (Código C.8). Isto porque o único bloco deste tipo existente na aba **Test** é o bloco associado à bandeira azul, e esse tem outro tipo de funções (ainda que seja possível criar um SDU a partir dele, como abordado na Secção 5.4.11). Já na classe `CommandBlockMorph` a opção está disponível quer os blocos representados por esta classe se encontrem na área de *scripting* da aba **Scripts** quer se encontrem na aba **Test**, desde que, nesta última, não estejam encaixados debaixo do bloco do tipo *hat* associado à bandeira azul (Código C.9). A ideia é que, também na aba **Test**, o utilizador possa criar SDUs, desde que o seu corpo não esteja encaixado debaixo do bloco associado à bandeira azul, pois este tem funções específicas que afectam o empilhamento do qual faz parte. Ou seja, qualquer empilhamento desta aba que não esteja encaixado neste bloco pode ser usado para criar um SDU. Assim, a opção **save script** fica visível no menu contextual dos blocos (Figura 5.23).

Figura 5.23: Opção **save script** disponível através do menu contextual dos blocos.

Posteriormente é preciso definir o método `saveScript`, associado a esta nova opção. Revisitando a hierarquia de classes que descende da classe `BlockMorph` (superclasse de todas as classes representativas dos blocos do Scratch), verifica-se que esta tem 3 subclasses directas: `CommandBlockMorph`, `HatBlockMorph` e `CommentBlockMorph`. Como estas três classes devem apresentar comportamentos distintos aquando da invocação da opção **save script**, têm, cada uma delas, de fornecer uma implementação do método `saveScript`. Assim, define-se o método como abstracto na classe `BlockMorph` (Código C.10), ficando a implementação concreta a cargo das 3 subclasses. Em Squeak, uma classe é considerada abstracta quando tem pelo menos um método abstracto. No entanto, continua a ser possível criar instâncias dessa classe. O único cuidado a ter consiste em não invocar métodos definidos como abstractos, pois estes irão lançar excepções. Na classe `CommentBlockMorph`, que é uma classe obsoleta que já não é usada, define-se que o corpo do método é vazio, deixando apenas um comentário elucidativo (Código C.11).

As outras duas classes fornecem então a implementação que interessa analisar. Em ambas, o algoritmo consiste em recolher a sequência de blocos que forma o corpo do SDU e, de seguida, invocar sobre o *Sprite* ou **Palco** para o qual se está a definir o SDU, o método `addBlockWithBody:`. De notar que, caso o empilhamento a partir do qual se quer criar o SDU tiver como primeiro bloco um bloco do tipo *hat*, então o corpo do SDU a ser guardado é formado por toda a sequência de blocos do empilhamento excepto esse primeiro bloco. Isto porque este tipo de blocos apenas define qual o evento que inicia a execução do empilhamento e, quando usados, são sempre os pri-

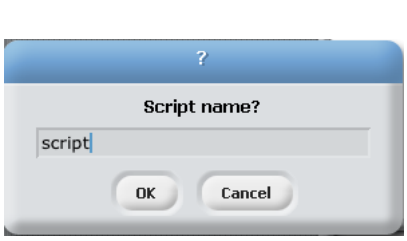
meiros blocos. Como não interessa guardar isso num SDU, visto que este será encaixado noutros blocos (e.g., pode até vir a ser encaixado num bloco do tipo *hat*), opta-se por descartar esses blocos. Do mesmo modo, não é possível criar um SDU constituído apenas por um bloco do tipo *hat*. Assim, na classe `HatBlockMorph`, a implementação tem a configuração definida no Código C.12, enquanto que a versão da classe `CommandBlockMorph` pode ser consultada no Código C.13.

Antes de apresentar o método `addBlockWithBody:` (invocado no `saveScript`), é preciso tomar atenção num tipo de blocos que também precisa de ter uma implementação do método `saveScript`, os `ReporterBlockMorph`, que representam os blocos do tipo *reporter*. Esta classe é superclasse das classes `VariableBlockMorph` e `ListContentsBlockMorph`, que devolvem o valor de uma variável e de uma lista, respectivamente. Como este tipo de blocos se limita a devolver um valor guardado, sem executar qualquer tipo de comportamento, não deve ser possível, pelo menos numa fase inicial, criar um SDU constituído apenas por um bloco deste género (note-se que é possível ter uma composição de blocos do tipo *reporter*, no entanto sobressai que a estrutura geral dessa composição é um bloco do tipo *reporter*). Para além disto, a invocação da opção **save script** sobre um bloco deste tipo tem um pormenor que é necessário ter em atenção. Se o bloco for relativo a uma variável local, então é assumido que se está a definir o SDU para um *Sprite* (uma variável definida para o **Palco** é sempre global ao **Palco** e aos *Sprites*; uma variável local é sempre relativa a um *Sprite*). Se for relativo a uma variável global então o comportamento por omissão assume que se está a definir o SDU para o **Palco**, o que pode não ser verdade (caso em que o bloco em questão está afecto a um *Sprite*). Como tal, o método `saveScript` também tem de ser implementado na classe `ReporterBlockMorph`, prevendo os dois casos referidos. Para o caso em que o utilizador tenta criar um SDU contendo apenas um único bloco desse tipo, surge uma mensagem de aviso. Para o caso em que a opção **save script** é desencadeada sobre um bloco deste tipo que está encaixado num outro bloco, então o método homólogo da classe do bloco no qual está encaixado é invocado. Este algoritmo garante que, para o caso de uma composição de blocos do tipo *reporter*, se chega a uma de duas situações: ou surge a mensagem de aviso caso o bloco de topo dessa composição seja do tipo *reporter*, ou o método homólogo da classe `CommandBlockMorph` acaba por ser invocado quando se atingir um bloco de um outro tipo (o método da classe `HatBlockMorph` não é executado porque para se chegar a um bloco do tipo *hat* tem que se passar primeiro por um bloco que o ligue ao bloco do tipo reporter em questão). O método `ReporterBlockMorph»saveScript` fica com a configuração visível no Código C.14.

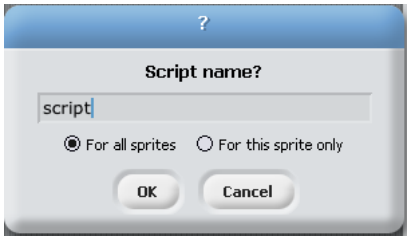
O método `addBlockWithBody:`, definido na classe `ScriptableScratchMorph`, é responsável por apresentar ao utilizador a janela de diálogo que lhe permite dar um nome ao SDU e, a partir daí, dar início ao processo de armazenamento interno do mesmo (Código C.15). Esta janela de diálogo tem duas modalidades: quando o SDU está a ser definido para o **Palco**, a janela apenas tem um campo de texto para colocar lá o nome do SDU; quando está a ser definido para um *Sprite*, tem também um botão que permite definir se o SDU a criar é global ou local (Figura 5.24).

Após dar o nome ao SDU e definir a sua abrangência, inicia-se o seu processo de armazenamento interno. Dada a necessidade de ter uma estrutura que guarde a informação relativa a um dado SDU, com vista a poder ser aumentada no futuro, define-se a classe `UserBlockDefinition`, com 3 variáveis de instância: `spec`, para guardar o nome do SDU; `body`, para guardar o seu corpo; `isGlobal`, que indica se o SDU é global ou local. Ao mesmo tempo definem-se os seus métodos selectores e modificadores. O método de criação de instâncias desta classe apresenta a configuração presente no Código C.16.

Como os SDUs podem ser definidos quer para os *Sprites* quer para o **Palco**, faz sentido que se adicione na superclasse comum das classes que os representam (`ScriptableScratchMorph`) uma nova variável de instância de nome `userBlocks`, que fica responsável por armazenar os SDUs. Esta variável é um dicionário que associa o nome do SDU à instância da sua definição



(a) Janela de diálogo de criação de um SDU para o **Palco**.



(b) Janela de diálogo de criação de um SDU para um *Sprite*.

Figura 5.24: Janelas de diálogo de criação de SDUs.

(`UserBlockDefinition`). Uma vez definida esta variável e o seu método selector, é necessário inicializá-la no método `ScriptableScratchMorph»initialize` (Código C.17). A inicialização desta variável nas subclasses que representam os *Sprites* e o **Palco**, `ScratchSpriteMorph` e `ScratchStageMorph`, respectivamente, é feita por via da invocação do método de inicialização da superclasse.

Tendo este suporte para armazenamento dos SDUs montado, no método `addBlockWithBody:`, o próximo passo consiste na criação de uma instância da nova classe definida, que vai guardar os parâmetros relevantes acerca do SDU a ser criado (Código C.18). Após este passo, e antes de colocar no dicionário a nova definição criada, é necessário verificar se já existe um SDU com o nome escolhido. Para isso é invocado sobre o *Sprite*/**Palco** para o qual se está a criar o SDU o método `existsBlock:global:`. Este método, declarado como abstracto na classe `ScriptableScratchMorph`, tem duas implementações distintas: na classe `ScratchSpriteMorph` e na `ScratchStageMorph`. A primeira verifica, em função do segundo argumento que indica se o SDU é local ou global, se o SDU existe no *Sprite* onde se pretende criá-lo ou em qualquer um dos *Sprites* existentes (Código C.19). A segunda faz uma verificação simples (Código C.20). Em ambos os casos, esta verificação consiste na invocação do método `existsBlock:` da classe `ScriptableScratchMorph` que verifica, para o *Sprite* ou **Palco** em questão, se o dicionário representado pela variável `userBlocks`, possui uma entrada cuja chave é o nome do SDU que se pretende criar (Código C.21). Se por acaso já existir um SDU com o nome escolhido, o utilizador é avisado desse facto e o processo de criação é terminado. Feita esta validação, chega-se à fase de guardar a definição previamente criada, através da invocação do método `createUserBlockWithDefinition:global:` (Código C.22). Este método segue o mesmo esquema de implementação do método `existsBlock:global:`, sendo abstracto na classe `ScriptableScratchMorph` e concreto nas suas duas subclasses. Na classe `ScratchStageMorph` este método limita-se a invocar o método `createUserBlockWithDefinition:` que vai adicionar a entrada do novo SDU ao dicionário. Na classe `ScratchSpriteMorph` faz o mesmo, sendo a sua acção limitada a um *Sprite*, caso o SDU seja local, ou pode afectar todos os *Sprites*, caso seja global. No entanto, nesta classe, o método tem de fazer uns procedimentos extra relacionados com a utilização de blocos de variáveis ou listas locais em SDUs globais. Um SDU global fica disponível para todos os *Sprites*. Mas se na sua definição utilizar um bloco relativo a uma variável ou lista que seja local ao *Sprite* onde foi definido, então essa variável/lista tem de ser adicionada a todos os *Sprites* que ainda não a possuam definida, para que o SDU, ao executar nesses *Sprites*, tenha comportamento válido. Para as variáveis é necessário verificar se o SDU contém blocos do tipo `SetterBlockMorph` (alteram o valor de uma variável) ou `VariableBlockMorph`. Se possuir, então é necessário garantir que todos os *Sprites* possuem a variável a que cada um

desses blocos se refere, através do método `ScriptableScratchMorph»addVariable:` (Código C.23). Para as listas, o mecanismo é semelhante sendo que neste caso os blocos a ter em atenção são os `ListContentsBlockMorph`, e as novas listas são adicionadas aos vários *Sprites*. Assim, na classe `ScratchSpriteMorph`, o método `createUserBlockWithDefinition:global:` fica de acordo com o Código C.24. Na classe `ScratchStageMorph`, tal como referido anteriormente, o método é bastante mais simples (Código C.25). O método `createUserBlockWithDefinition:` da classe `ScriptableScratchMorph` vai adicionar ao dicionário de SDUs uma nova entrada onde, ao nome do SDU, associa uma cópia da sua definição. Antes de o fazer, afecta o corpo do SDU ao *Sprite* onde vai ser armazenado, garantindo deste modo que cada cópia dum SDU global guardada num dado *Sprite* fica afecta a esse *Sprite* (Código C.26). Concluído este processo, resta apenas actualizar a categoria de blocos **MyScripts** para que passe a listar o novo SDU (Código C.27).

Todo o fluxo de execução de métodos para a criação de um SDU pode ser observado no esquema da Figura 5.25.

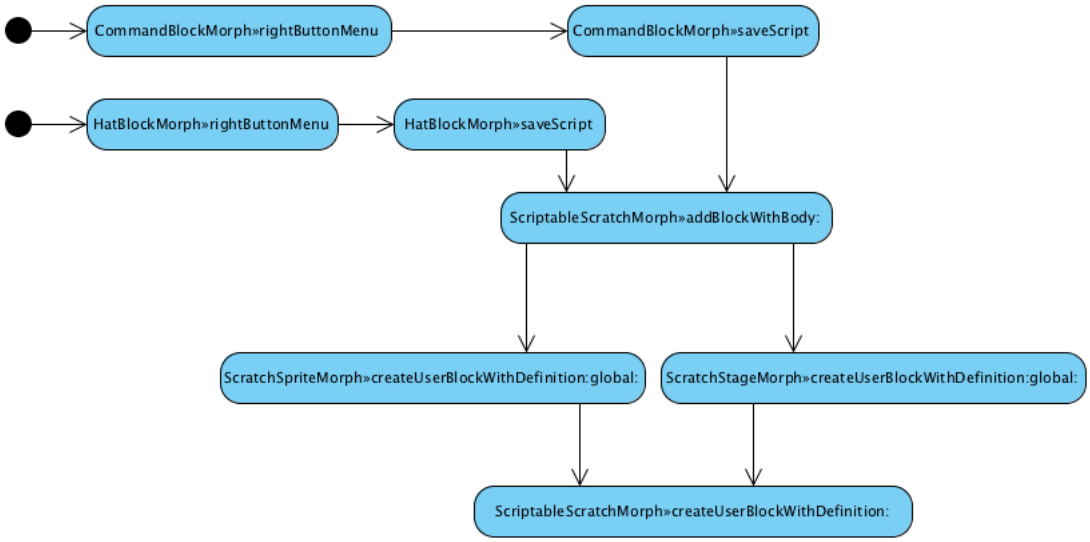


Figura 5.25: Esquema de execução de métodos do processo de criação de um SDU.

Para apresentar o SDU como um bloco na categoria **MyScripts**, define-se, dentro do método `ScriptableScratchMorph»viewerPageForCategory:`, uma condição que indica que a disposição dos SDUs nesta categoria de blocos é dada pelo método `viewerPageForMyScripts` da mesma classe (Código C.28). Neste método indica-se que os SDUs são apresentados separando os globais dos locais com um conector (linha horizontal), ficando os globais localizados acima do conector e os locais abaixo. Cada um destes grupos de SDUs é ordenado alfabeticamente pelo nome para permitir ao utilizador encontrar mais facilmente o SDU que pretende (Código C.29). A criação de cada bloco (elemento da interface) é feita através do método `newUserBlockFor:`, que vai criar a instância da classe que representa visualmente os SDUs e atribuir-lhe valores específicos. Essa classe, criada para o efeito, é a classe `UserCommandBlockMorph`, subclasse de `CommandBlockMorph`. À instância então criada desta classe atribuem-se valores como a cor do bloco que a representa ou o nome do SDU (Código C.30). Neste passo atribui-se como selector o método `userBlockWarn` da classe `UserCommandBlockMorph`. Este método mostra uma janela de diálogo onde informa o utilizador acerca da forma de utilização deste tipo de blocos (peças

definidas pelo utilizador) (Código C.31). Apesar de ser este o método associado ao bloco representativo do SDU, não é ele que é executado quando se utiliza o SDU. Quem trata de executar o SDU é o bloco **import**. Nesta classe também se define o método `rightButtonMenu`, o qual, inicialmente, tem uma definição igual ao método homólogo da superclasse `CommandBlockMorph`.

Finalizados estes pormenores, passa a ser possível visualizar os SDUs criados, com a disposição referida, na categoria **MyScripts** (Figura 5.26).

Figura 5.26: SDUs criados e dispostos na categoria **MyScripts**.

5.4.7 Alteração do bloco import

O bloco **import** criado em fase anterior (Secção 5.4.3) é uma instância da classe `CBlockMorph`, a qual cria uma bloco em forma de 'C', tal como está definido na sua especificação, no método de classe `blockSpecs` de `ScriptableScratchMorph`. Este tipo de blocos permite que o utilizador coloque no seu interior vários blocos empilhados (Figura 5.27).

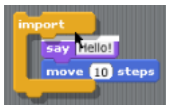


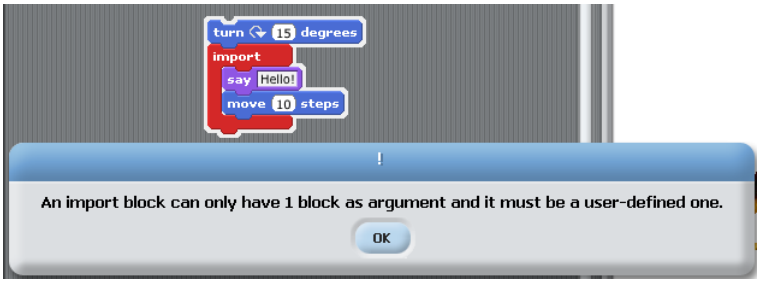
Figura 5.27: Bloco **import** pronto a englobar um empilhamento de blocos.

No entanto, agora pretende-se que este bloco aceite apenas um único bloco no seu interior (o SDU que se pretende executar). Para diferenciar este bloco dos restantes em forma de 'C' cria-se uma subclasse de `CBlockMorph`. Assim, define-se a classe `ImportCBlockMorph`, a qual não acrescenta estrutura, mas define novo comportamento (Código C.32). Antes de se analisar as diferenças que esta classe introduz, pode-se aproveitar para alterar o tipo associado ao bloco **import**. Assim, na especificação do bloco, altera-se o tipo de 'c' para 'i' (para mais facilmente ser associado à funcionalidade do bloco e ao nome da sua classe) (Código C.33). Depois é preciso alterar o método `ScriptableScratchMorph»blockFromSpec:color:` que cria um bloco a partir da sua especificação. Neste introduz-se um novo teste que verifica se o tipo do bloco é dado pela letra 'i' e, caso seja, cria-se uma nova instância de `ImportCBlockMorph`. Analogamente às instâncias de `CBlockMorph` criadas neste método, também para este novo tipo de bloco se indica que tem uma regra de avaliação própria, através do método `isSpecialForm:` da classe `BlockMorph` (Código C.34). A indicação de que tem uma regra de avaliação própria é depois aproveitada, durante a execução de um processo, pelo método `evaluateSpecialForm` da classe `ScratchProcess` para invocar o método associado ao bloco (Código C.35). Com estas alterações, o bloco **import** já passa a constar na palete como uma instância da nova classe criada.

Seguidamente tem de se definir comportamento para indicar que o bloco **import** apenas suporta um único bloco (definido pelo utilizador) no seu interior. Em primeiro lugar, define-se dentro do protocolo `accessing` da classe `ImportCBlockMorph` o método `nestedBlock` (Código C.36), que é um simples método de acesso à variável de instância com o mesmo nome existente na superclasse `CBlockMorph` e que contém a referência para o bloco contido dentro do bloco **import**. De seguida é preciso introduzir um caso especial no método associado ao bloco **import**, `ScratchProcess»importScript`, para indicar que, quando for executado um *script* onde exista um bloco **import** que tenha no seu interior algo que não um bloco definido pelo utilizador, nele seja assinalado um erro, parando de imediato a execução do *script* onde se encontra. Para isso tira-se partido do método `BlockMorph»showError` que muda a cor do bloco para vermelho (Código C.37). Outro método a usar é o `BlockMorph»stop` que pára a execução do processo (Código C.38). Neste, o método de mesmo nome da classe `ScratchProcess` é também invocado (Código C.39).

Após a sinalização do bloco com a cor vermelha e a paragem do processo, invoca-se o método `importBlockWarn`, definido no protocolo `private` da classe `ImportCBlockMorph`, para mostrar a mensagem de aviso de como se deve utilizar o bloco **import** (Código C.40). Posteriormente executa-se o método `ScratchProcess»errorFlag:` (Código C.41), o qual vai indicar ao sistema que ocorreu um erro, através da variável `errorFlag`. Esta variável é depois usada, durante a actualização da interface, pelo método `BlockMorph»updateCachedFeedbackForm` (Código C.42) que vai contornar de cor vermelha todo o *script* onde se encontra o bloco que originou o erro. Após a execução do método `errorFlag:`, executa-se o método `ScratchProcess»doReturn`, o qual está associado ao bloco que é usado para terminar um *script*, fazendo assim com que a execução do *script* em questão termine aquando do erro. Portanto, o método `importScript` fica de acordo com o Código C.43.

As Figuras 5.28a e 5.28b espelham o resultado de todas estas alterações, mostrando como o utilizador é avisado de um erro derivado de utilizar o bloco **import** de forma errada. Primeiro surge o bloco **import**, que gerou o erro, assinalado a vermelho, juntamente com a mensagem de aviso. Após o utilizador fechar a janela da mensagem de aviso, o *script* é contornado a vermelho, continuando o bloco de erro assinalado.



(a) Bloco **import** assinalado a vermelho e mensagem de aviso.

(b) Bloco **import** e *script* assinalados a vermelho.

Figura 5.28: Sinalização de erros na utilização do bloco **import**.

Falta ainda ajustar o método associado ao bloco **import** para que passe a executar o único SDU que contém. A primeira alteração que se tem de fazer consiste em, após se obter a referência para o bloco **import**, verificar se este não possui nenhum bloco no seu interior (pode acontecer que o utilizador encaixe apenas o bloco **import** num *script* e o mande executar). Caso não

possua, então o bloco não executa. Para executar o único SDU é necessário aceder ao seu corpo (conjunto de blocos que o definem). Como as estruturas que armazenam estes blocos estão guardadas nos objectos `ScriptableScratchMorph` existentes, é necessário aceder ao objecto onde o SDU em causa está definido, devolvendo o seu corpo para que possa ser executado (Código C.44). É necessário invocar o método `blockSequence` sobre o corpo do bloco, pois o corpo está definido como sendo apenas o primeiro bloco que o constitui (sendo os restantes blocos seus *submorphs*). Desta forma obtém-se, a partir do primeiro bloco, uma lista ordenada de todos os blocos, os quais são executados de seguida. O método `definitionOfUserBlock:`, definido na classe `ScriptableScratchMorph`, devolve a instância de `UserBlockDefinition` associada ao bloco a executar (Código C.45).

Feitas estas alterações, o bloco **import** fica pronto a executar o seu único bloco (Figura 5.29).

Figura 5.29: Bloco **import** com SDU.

5.4.8 Manipulação dos SDUs

No que toca aos SDUs há duas situações a tratar relativamente à forma como estes são manipulados pelo utilizador. A primeira diz respeito à execução deste tipo de blocos quando o utilizador faz um clique simples ou duplo sobre o bloco, por forma a ver qual o seu efeito. Uma vez que estes blocos só podem ser executados dentro do bloco **import**, é necessário fornecer essa informação ao utilizador quando ele tenta executar as acções referidas. A solução passa por redefinir os métodos `click:` (Código C.46) e `doubleClick:` (Código C.47) da classe `BlockMorph` na nova classe `UserCommandBlockMorph`. Originalmente, estes métodos, e para o caso em que o clique simples ou duplo é feito sobre um bloco, iniciam a execução do *script* onde o bloco se encontra. Caso o bloco não esteja em nenhum *script* (quando está na palete ou livre na área de *scripting*), então executam o bloco. Ao serem redefinidos tem que se indicar que só é executado o *script* caso o bloco definido pelo utilizador esteja encaixado no bloco **import**. Caso esteja então invoca-se o mesmo método da superclasse. Caso contrário invoca-se o método `userBlockWarn`, que mostra uma janela de diálogo (Figura 5.30) onde informa o utilizador acerca da forma de utilização deste tipo de bloco (Código C.48 e C.49).

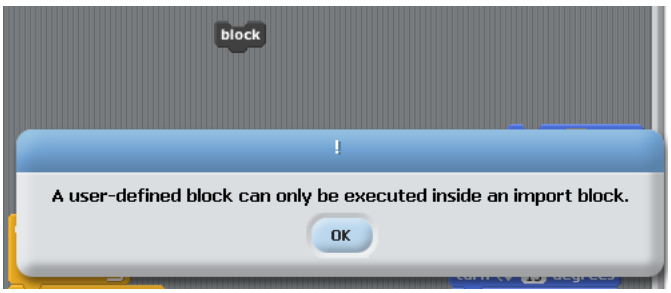
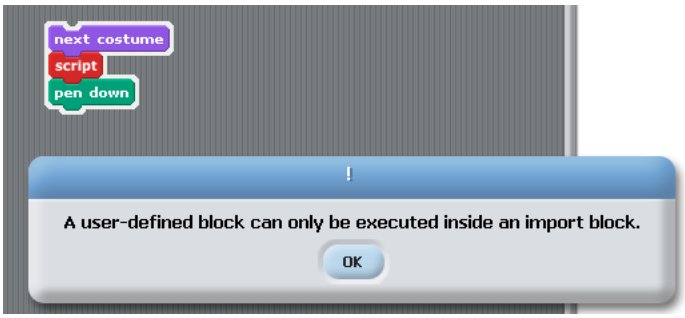


Figura 5.30: Mensagem de aviso que indica como se deve usar um SDU.

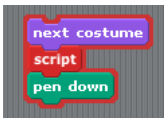
A segunda situação tem a ver com a execução de um *script* onde esteja encaixado um SDU sem usar o bloco **import**. É necessário prever este caso para que a utilização deste tipo de

blocos como blocos normais não faça surgir a típica janela de *debugging* com a mensagem de erro, mas antes uma indicação visual do mesmo. Seguindo a cadeia de invocações de métodos da classe `ScratchProcess` que ocorre durante a execução de um *script*, a certa altura, é invocado o método `applyPrimitive`, o qual, por sua vez, invoca o método `evaluateWithArgs:` sobre o `CommandBlockMorph` que está a processar, para que um bloco desse tipo seja executado com os seus argumentos (Código C.50). O método `CommandBlockMorph≫evaluateWithArgs:`, para a maioria dos casos, executa o método associado ao bloco em questão, com a sua lista de argumentos (Código C.51). Sendo a classe que representa os SDUs, `UserCommandBlockMorph`, uma subclasse de `CommandBlockMorph`, os blocos por ela representados são avaliados por este método quando se encontram encaixados num empilhamento sem ser dentro dum bloco **import**. Como os SDUs têm um método associado (`userBlockWarn`) que não é aquele que executa o seu corpo, tem de se introduzir aqui um caso especial para indicar que, quando este tipo de blocos estiver a ser executado fora do contexto dum bloco **import**, neles seja assinalado um erro, parando de imediato a execução do *script* onde se encontram. Para isso segue-se uma abordagem semelhante à utilizada para a sinalização do erro no bloco **import**, invocando primeiramente os métodos `BlockMorph≫showError` e `BlockMorph≫stop`. Após a sinalização do bloco com a cor vermelha e a paragem do processo, invoca-se o método associado ao bloco para mostrar a mensagem de aviso de como se deve utilizar um SDU. Posteriormente, devolve-se para o método `applyPrimitive` o símbolo `#stop` (Código C.52). Este símbolo `#stop` é então usado para executar os métodos `ScratchProcess≫errorFlag:` e `ScratchProcess≫doReturn`, fazendo assim com que a execução do *script* em questão termine aquando do erro (Código C.53).

As Figuras 5.31a e 5.31b espelham o resultado de todas estas alterações, mostrando como o utilizador é avisado de um erro derivado de utilizar um SDU num *script*, fora do bloco **import**. Primeiro surge o bloco que gerou o erro assinalado a vermelho, juntamente com a mensagem de aviso. Após o utilizador fechar a janela da mensagem de aviso, o empilhamento é contornado a vermelho, continuando o bloco de erro assinalado.



(a) Bloco de erro assinalado a vermelho e mensagem de aviso.



(b) Bloco de erro e respectivo empilhamento assinalados a vermelho.

Figura 5.31: Sinalização de erros na utilização dos SDUs.

5.4.9 Adição de novo *Sprite*

Nesta secção aborda-se a questão relacionada com a adição de um *Sprite* a um projecto e a necessidade de adicionar a esse novo *Sprite* os SDUs globais já existentes. Isto permite manter a coerência na nomenclatura de "global", pois um SDU dito global deve existir em todos os *Sprites* e não apenas em alguns.

O Scratch permite que o utilizador adicione um novo *Sprite* a um projecto usando uma de 6 opções. Três delas estão localizadas por baixo do **Palco** (Figura 5.32):

- Pintar um novo *Sprite*;
- Adicionar um *Sprite* a partir de um ficheiro;
- Adicionar um *Sprite* aleatório.

Figura 5.32: Opções de adição de novo *Sprite* disponíveis por baixo do **Palco**.

A quarta opção consiste em arrastar para o ambiente Scratch um ficheiro de extensão `.sprite`, que contenha um *Sprite* gravado. A quinta opção consiste em accionar a opção **turn into new sprite** presente no menu contextual de um traje, que cria um novo *Sprite* com esse traje. A sexta opção consiste em accionar a opção **grab screen region for new sprite** do menu contextual do **Palco** para seleccionar uma área do mesmo e transformá-la no traje do novo *Sprite*.

No método `ScratchLibraryMorph»makeNewSpriteButtons:` estão definidas as 3 primeiras opções (Código C.54). A primeira opção permite fazer um desenho no editor de imagens e a partir dele gerar um *Sprite*, sendo o desenho a aparência do *Sprite*. Essa opção tem a si associado o método `paintSpriteMorph` da classe `ScratchFrameMorph`. Neste método é invocado o método `ScratchFrameMorph»addAndView:` que recebe um *Sprite* como argumento e o adiciona ao conjunto de *Sprites* do projecto. A segunda opção adiciona um *Sprite* a partir de um ficheiro, através do método `addSpriteMorph` da classe `ScratchFrameMorph`. Aqui existem dois modos distintos de tratamento de ficheiros: se o ficheiro seleccionado tiver a extensão `.sprite`, é invocado o método `importSpriteOrProject:` da mesma classe. Se tiver uma extensão de um ficheiro de imagem, é invocado o `addAndView:`. A terceira opção adiciona um novo *Sprite* escolhendo um ficheiro de imagem aleatório e invocando o método `addAndView:`, à semelhança do que acontece num dos casos da segunda opção. Quanto à opção de arrastar e largar sobre o ambiente Scratch um ficheiro representativo de um *Sprite*, o método `ScratchFrameMorph»processDroppedFiles` define que esse caso é tratado pelo método `importSpriteOrProject:`, já referido para a segunda opção. A quinta opção tem a si associado o método `MediaItemMorph»turnIntoNewSprite` que, mais uma vez, invoca o método `addAndView:`. A sexta e última opção tem a si associado o método `ScratchStageMorph»grabSpriteFromScreen`, o qual também acaba por invocar o método `addAndView:`.

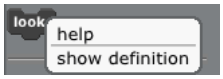
Portanto, encontram-se aqui dois pontos comuns a vários processos de adição de *Sprites*: o método `addAndView:` e o `importSpriteOrProject:`. Consultando o segundo método verifica-se que, na sua fase final, invoca o método `addAndView:` para cada um dos *Sprites* importados (Código C.55). Portanto o único método em que se tem de efectuar alterações é o `addAndView:` (Código C.56). Este método posiciona o *Sprite* recebido como argumento, depois adiciona-o à lista de *Sprites* e, por fim, mostra-o, ficando este seleccionado. As alterações têm, portanto, de ser introduzidas antes do *Sprite* ser adicionado à lista de *Sprites*. Em primeiro lugar, é necessário indicar que os `SetterBlockMorph`s do *Sprite* importado devem ficar afectos ao mesmo, pois

verifica-se que esse tipo de blocos que alteram o valor de variáveis globais, ao serem importados com o *Sprite*, ficam afectos ao **Palco**. De seguida, adicionam-se os SDUs globais já existentes ao novo *Sprite*. Por último faz-se o processo inverso, isto é, adiciona-se aos *Sprites* já existentes os SDUs globais importados com o *Sprite*. Esta última alteração ainda não surte efeito, pois ainda não é possível exportar *Sprites* que possuam SDUs. No entanto, o sistema fica já preparado para os suportar. O método `addAndView:` fica então com a configuração presente no Código C.57.

Feitas estas alterações, conclui-se o objectivo de propagar os SDUs globais pelos novos *Sprites* adicionados ao projecto.

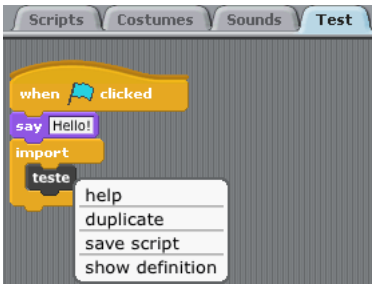
5.4.10 Consulta da definição de um SDU

Caso deseje, o utilizador deve poder consultar o corpo de um bloco definido por si ou por outra pessoa. Como tal, impõe-se a existência de uma opção **show definition**. Esta opção deve estar disponível através de um menu contextual existente para os blocos definidos pelo utilizador nas seguintes situações: quando o bloco está na palete (Figura 5.33a) e quando está na área de *scripting* da aba **Scripts** (Figura 5.33b) ou da aba **Test** (Figura 5.33c). Faz sentido que nesta última aba também esteja disponível pois ao consultar a definição de um bloco pode acontecer que esta, por sua vez, contenha outros blocos definidos por um utilizador, pelo que pode haver necessidade de consultar a definição destes sem ter de os encontrar na palete.



(a) Menu contextual existente quando o bloco definido pelo utilizador está na palete.

(b) Menu contextual existente quando o bloco definido pelo utilizador está na área de *scripting* da aba **Scripts**.



(c) Menu contextual existente quando o bloco definido pelo utilizador está na área de *scripting* da aba **Test**.

Figura 5.33: Menus contextuais para consulta da definição de um SDU.

Por forma a cumprir estas restrições, é necessário alterar o método `rightButtonMenu` da classe `UserCommandBlockMorph`, que define este menu contextual, adicionando-lhe a nova opção (Código C.58). Posteriormente, define-se no protocolo `private` da mesma classe o método `showDefinition`, que está associado a essa opção do menu contextual (Código C.59). Este método carrega o *script* que representa o corpo do bloco para a área de *scripting* da aba **Test**, encaixando-o debaixo do bloco associado à bandeira azul. Neste método, em primeiro lugar, obtém-se uma cópia do corpo bem como a referência para o editor. A cópia, realizada pelo método `deepCopy`, permite que o *script* que é colocado na aba seja independente do que está definido na estrutura de dados que o armazena. Desta forma evita-se que, ao alterar o *script*

após este estar carregado na aba **Test**, a definição armazenada seja automaticamente alterada (tal aconteceria se houvesse partilha). Depois altera-se a aba actual para passar a ser a aba **Test** e obtém-se a referência para o bloco associado à bandeira azul. De seguida, é necessário limpar todo o conteúdo da aba **Test**, deixando apenas o bloco associado à bandeira azul. Esta tarefa implica remover qualquer *script*, inclusivé algum que esteja encaixado debaixo deste bloco. Depois coloca-se debaixo do bloco associado à bandeira azul o *script* que se pretende carregar. Antes de terminar, é preciso indicar que o objecto `ScriptableScratchMorph` associado ao bloco o está a consultar. Isto tem utilidade nas funcionalidades de actualização de um SDU, eliminação, etc. Para tal, define-se uma nova variável de instância nessa classe, de nome `inspectedBlock`, que guarda o nome do SDU que se está a consultar. Define-se também o seu método acessor e modificador, e define-se no construtor que o seu valor inicial é `nil` (pois inicialmente não se está a consultar nenhum SDU). Após isso, no método `showDefinition`, é preciso afectar esta variável com o nome do SDU que se pretende consultar. Para dar uma dica visual ao utilizador, altera-se também o texto do corpo do bloco associado à bandeira azul, para que acrescente ao seu texto inicial, **when [blue flag] clicked**, o texto "**run**" seguido do nome do bloco. Para tal, recorre-se ao método `addBlockName:` (Código C.60). Este método, definido em `EventHatMorph`, apaga qualquer texto que exista para além do "**when**" e "**clicked**", e só acrescenta o novo texto caso o nome do SDU seja diferente de `nil` (este caso é usado na situação em que se apaga um SDU que se está a consultar, para repôr o estado original). Neste método, o nome do bloco tem de ser codificado no conjunto de caracteres MacRoman para que, ao ser apresentado no texto do corpo do bloco, mostre correctamente os caracteres latinos.

O resultado final fica de acordo com a Figura 5.34.

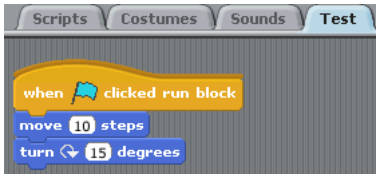


Figura 5.34: Corpo do bloco de nome **block** definido pelo utilizador, carregado na aba **Test**.

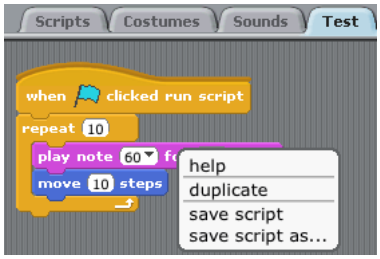
Esta funcionalidade permite também realçar que um SDU constituído por um único bloco do tipo *reporter* não faria sentido pois, ao consultá-lo, o bloco não encaixaria no bloco associado à bandeira azul.

5.4.11 "Gravar" e "Gravar como"

Após a consulta de um SDU, estando a aba **Test** aberta e com o *script* consultado encaixado debaixo do bloco associado à bandeira azul, faz sentido que estejam disponíveis ao utilizador 2 opções: uma que permita actualizar esse *script* após se alterar a sua estrutura e outra que permita criar um novo *script* a partir dessa estrutura. Estas opções devem estar disponíveis apenas para o empilhamento que se encontra debaixo do bloco associado à bandeira azul, pois é ele que representa a estrutura do SDU que está a ser consultado. Outros empilhamentos que porventura existam na área de *scripting* têm as opções normais existentes para empilhamentos da aba **Scripts** (incluindo a opção **save script** normal).

Estas duas novas opções estão então disponíveis através do menu contextual dos blocos que fazem parte do empilhamento encabeçado pelo bloco associado à bandeira azul (este incluído). Essas opções são a **save script** que, apesar de ter o mesmo nome da opção existente para blocos da aba **Scripts** e para blocos livres da aba **Test**, não faz o mesmo, pois esta actualiza o corpo

do SDU consultado, e a opção **save script as** que cria um novo SDU a partir do *script* consultado (podendo este ter sofrido alterações ou não). Para definir estas novas opções é preciso alterar o método `rightButtonMenu` das classes `HatBlockMorph` (para abranger o bloco associado à bandeira azul), `CommandBlockMorph` e `UserCommandBlockMorph` (para abranger todos os outros tipos de blocos). Para a classe `HatBlockMorph` apenas se tem de testar se a aba actual é a **Test**, para adicionar estas duas opções (Código C.61). Antes de se adicionar a opção de actualização do SDU é preciso verificar se se está a consultar algum SDU, pois inicialmente nenhum está a ser consultado. Nas classes `CommandBlockMorph` e `UserCommandBlockMorph`, tem que se indicar que as duas novas opções só estão disponíveis caso o bloco em questão pertença ao *script* que está encaixado debaixo do bloco associado à bandeira azul, existente na aba **Test** (Código C.62) (Figura 5.35a). A opção **save script** normal, que cria um novo SDU dado um empilhamento de blocos, está presente nos `CommandBlockMorph`s (e em particular, na sua subclasse `ReporterBlockMorph`) e na classe dos `UserCommandBlockMorph`s, quer estes se encontrem na aba **Scripts**, quer se encontrem nalgum empilhamento livre na aba **Test** (Figura 5.35b). Para os `HatBlockMorph`s, apenas está presente na aba **Scripts**, pois na aba **Test**, o único bloco desse tipo é o que está associado à bandeira azul.



(a) Opções disponíveis para um bloco do empilhamento encabeçado pelo bloco associado à bandeira azul, da aba **Test**.

(b) Opções disponíveis para um empilhamento livre na aba **Test**.

Figura 5.35: Opções do menu contextual para blocos da aba **Test**.

Quanto às duas novas opções, a primeira, **save script**, tem a si associado o método de instância `updateBlockDefinition`, o qual está definido como abstracto na classe `BlockMorph`, indicando no seu corpo, que a responsabilidade de fornecer uma implementação é delegada nas suas subclasses (Código C.63).

As suas subclasses `HatBlockMorph` e `CommandBlockMorph` (e `ReporterBlockMorph`, sua subclasse) fornecem implementações concretas para este método. A subclasse `CommentBlockMorph` (obsoleta), por herança, fica também com essa responsabilidade. No entanto, para esta, fornece-se apenas uma implementação vazia e um comentário elucidativo, tal como havia sido feito no método `saveScript` usado na criação de novos SDUs. Este método `updateBlockDefinition` é responsável por recolher o *script* que se encontra encaixado debaixo do bloco associado à bandeira azul e actualizar a definição do bloco que se está a consultar, passando esse *script* a ser o novo corpo do bloco. Para o caso em que o bloco que se está a consultar é global a todos os *Sprites*, essa definição tem de ser actualizada em todos eles. Caso o bloco que está a ser consultado seja relativo ao **Palco**, então é local, pois apenas existe um **Palco**, logo, apenas é necessário actualizar a definição que o **Palco** contém desse bloco. Para isso recorre-se ao método `createUserBlockWithDefinition:global:`, já usado quando se cria um novo SDU, e que, em função do bloco ser local ou global, trata de fazer a tal actualização. Assim, na classe

`HatBlockMorph`, o método tem a configuração presente no Código C.64. A definição presente na classe `CommandBlockMorph` segue o mesmo padrão, diferindo na forma como recolhe o *script* e na determinação do objecto (*Sprite*/**Stage**) afecto ao *script* (Código C.65). No fim, ainda é preciso actualizar a aba **Test** dos *Sprites* que estejam a consultar o bloco, no caso dele ser global, para que, debaixo do bloco associado à bandeira azul indicando qual o bloco que está a ser consultado, surja o novo corpo desse bloco, mantendo assim coerência entre a informação apresentada ao utilizador e a definição internamente guardada acerca do bloco em causa. Para isso invoca-se o método `ScriptableScratchMorph≫updateTestTabScript:`, o qual percorre cada *Sprite* que esteja a consultar o bloco que foi actualizado, e substitui o *script* presente na aba **Test** pelo *script* correspondente ao novo corpo do bloco (Código C.66). O método `Object≫~~` utilizado verifica se o receptor e o argumento do método não são o mesmo objecto.

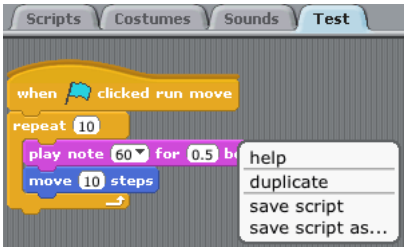
Quanto à classe `ReporterBlockMorph`, subclasse de `CommandBlockMorph`, também tem de fornecer uma implementação do método `updateBlockDefinition` pelos mesmos motivos que teve de implementar o método de criação de um SDU, `saveScript`. Assim, a implementação deste método segue o mesmo esquema da implementação do `saveScript`, sendo invocado o método homólogo do *owner* do bloco (Código C.67).

Feito isto, a actualização decorre como esperado. A definição interna do bloco é actualizada, local ou globalmente, e as abas **Test** dos *Sprites* que o estejam a consultar são igualmente actualizadas.

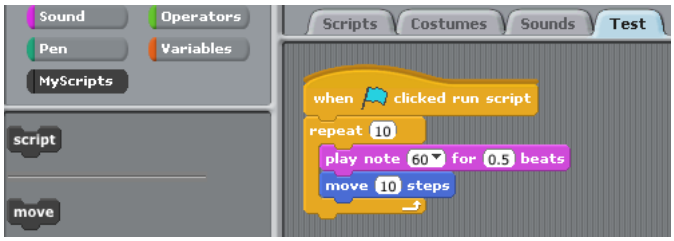
A opção **save script as** é semelhante à opção normal **save script** existente na aba **Scripts**, que permite criar um novo SDU a partir de um empilhamento de blocos. A diferença é que agora é necessário actualizar o texto do corpo do bloco associado à bandeira azul para que nele passe a constar o nome do novo bloco: "**when [blue flag] clicked run *nomeBloco***". Isto é necessário pois sendo esta opção executada sobre um bloco pertencente ao empilhamento encabeçado pelo bloco associado à bandeira azul, criando um novo SDU com base nesse empilhamento, o mesmo passa a representar o corpo desse novo SDU, e como tal faz sentido que o utilizador possua essa informação visual. Esta nova opção tem a si associado o método `saveScriptAs`. Mais uma vez, este método segue o mesmo esquema de implementação dos métodos `saveScript` e `updateBlockDefinition`, sendo abstracto na classe `BlockMorph` e concreto nas subclasses `HatBlockMorph`, `CommandBlockMorph` e `ReporterBlockMorph`, ficando a `CommentBlockMorph` com a implementação vazia. As implementações concretas das classes `HatBlockMorph` (Código C.68) e `CommandBlockMorph` (Código C.69), após obterem o corpo do novo bloco, invocam o método `ScriptableScratchMorph≫saveBlockWithBody:`. O método `saveBlockWithBody:` segue o procedimento normal de criação de um novo SDU, já demonstrado na Secção 5.4.6. Apenas tem duas instruções extra: uma para actualizar a variável que refere qual o bloco que está a ser consultado e outra que actualiza o texto do corpo do bloco associado à bandeira azul com o nome do novo bloco (Código C.70). A classe `ReporterBlockMorph` segue o mesmo esquema apresentado anteriormente para a actualização do SDU (Código C.71).

Após a execução desta opção, o novo SDU aparece na categoria **MyScripts** com o nome que lhe foi atribuído. O seu corpo é constituído pelo *script* presente na aba **Test**, debaixo do bloco associado à bandeira azul, e este último altera o texto do seu corpo para indicar esse facto (Figura 5.36).

É de interesse ter uma opção do menu contextual da área de *scripting* da aba **Test** que permita limpá-la. Esta opção remove qualquer *script* que se encontre encaixado debaixo do bloco associado à bandeira azul, bem como outros que se encontrem livres na área de *scripting*. Para além disso, coloca a `nil` a variável `inspectedBlock` do *Sprite*/**Stage** que referencia essa área de *scripting*, indicando que não se está a consultar nenhum SDU para esse *Sprite*/**Stage**, e altera o texto do corpo do bloco associado à bandeira azul para o seu valor inicial: "**when [blue flag] clicked**". Isto permite voltar ao contexto inicial da aba. Para



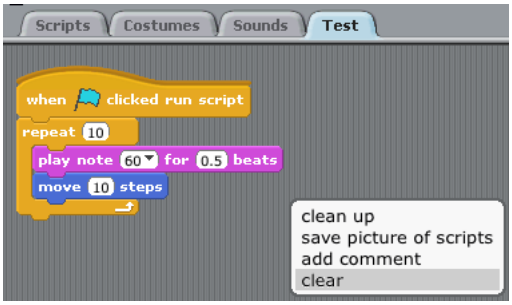
(a) Opção **save script as** disponível no *script* da aba **Test**.



(b) Bloco **script** criado e texto do bloco associado à bandeira azul alterado, como resultado da execução da opção **save script as**.

Figura 5.36: Criação de um SDU a partir do corpo de outro.

obter este efeito, adiciona-se a nova opção **clear** ao menu contextual supracitado, alterando o método `ScratchScriptEditorMorph»scriptsMenu:` (Código C.72). Na mesma classe, no protocolo `menu/button ops`, cria-se o método `clear` (Código C.73), obtendo-se o efeito final mostrado na Figura 5.37.



(a) Opção **clear** disponível no menu contextual da área de *scripting* da aba **Test**.

(b) Resultado da execução da opção **clear**.

Figura 5.37: Efeito da opção **clear**.

5.4.12 Eliminação de um SDU

Tal como existe a possibilidade de criar um SDU, faz sentido que a funcionalidade de eliminar um também esteja disponível ao utilizador. Mais uma vez se recorre ao menu contextual deste tipo de blocos, definido em `UserCommandBlockMorph»rightButtonMenu` (Código C.74), para adicionar a opção **delete script**, à qual se associa o método `deleteScript`, obtendo-se o resultado mostrado na Figura 5.38.

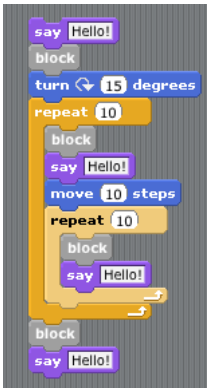
Figura 5.38: Menu contextual para um SDU presente na palete, com a opção **delete script**.

Esta opção, no entanto, apenas está disponível para os SDUs que se encontrem na palete, isto é, se estiverem numa área de *scripting*, esta opção não aparece. Isto deve-se ao facto de que este processo de eliminação deve eliminar toda e qualquer instância do bloco em questão. Como já existe a opção **delete** no mesmo menu contextual quando o bloco está numa área de *scripting*, e porque a palete apresenta os blocos existentes no sistema, opta-se por esta solução por forma a tornar mais perceptível qual o efeito gerado.

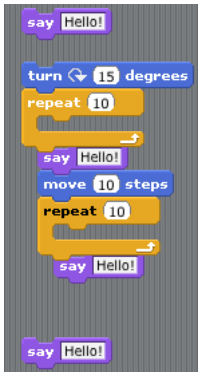
Associado a esta opção está o método `UserCommandBlockMorph»deleteScript` (Código C.75). Este método é responsável por remover a entrada relativa ao SDU em questão do dicionário onde está armazenado e por limpar das abas **Scripts** e **Test** qualquer instância desse SDU. Se o SDU está definido para o **Palco**, então é este o comportamento definido. Para os *Sprites*, há que distinguir o caso em que o SDU é local ao *Sprite* e o caso em que é global a todos os *Sprites*. Para o caso de ser local a um *Sprite*, é executado o comportamento definido apenas para esse *Sprite*. Se for global, então é para todos os *Sprites*.

Em primeiro lugar, interroga-se o utilizador acerca da sua certeza de efectuar a eliminação do SDU, para evitar acidentes na utilização do menu contextual. Em caso afirmativo, obtém-se o nome do SDU e o valor booleano a si associado que indica se é global ou não. Posteriormente, remove-se a entrada relativa ao SDU do dicionário do objecto `ScriptableScratchMorph` no qual o bloco está definido. Isto é feito através do método `removeUserBlock:` da classe `ScriptableScratchMorph` (Código C.76). Seguidamente, apaga-se das abas **Scripts** e **Test** do objecto `ScriptableScratchMorph` ao qual está associado o SDU, toda e qualquer instância deste, através do método `ScriptableScratchMorph»deleteUserBlock:`. Este trata de invocar o método de mesmo nome da classe `ScratchScriptsMorph` sobre as variáveis que representam as duas abas (Código C.77). Em cada uma das abas, as instâncias existentes do bloco a remover são apagadas. A abordagem seguida pelo BYOB para esta funcionalidade passa por simplesmente remover os blocos, deixando os blocos que se encontravam encaixados debaixo destes exactamente na mesma posição onde estavam, passando o seu *owner* a ser a instância de `ScratchScriptsMorph` que representa a área de *scripting* da aba. Isto faz com que, por exemplo, ao eliminar um bloco que está encaixado num empilhamento dentro dum bloco em forma de 'C' ou num bloco **if-else**, os blocos desse empilhamento que se encontravam por baixo do bloco a remover deixem de pertencer ao interior do bloco em forma de 'C' ou **if-else** que os continha. Para um empilhamento de maiores dimensões, esta operação deixa-o desfigurado, cabendo ao utilizador a responsabilidade de reorganizar os blocos (Figura 5.39).

Tendo em vista deixar o *script* num estado mais amigável para o utilizador, para que este tenha o menor esforço a recompô-lo, o método `ScratchScriptsMorph»deleteUserBlock:` faz



(a) *Script* desenvolvido no BYOB, contendo o SDU **block**.



(b) O mesmo *script* anterior, após a remoção do bloco **block**.

Figura 5.39: Eliminação de um SDU no BYOB.

uma série de testes para verificar os mais diversos casos particulares em que um bloco se pode encontrar. Entre os casos testados encontram-se os seguintes:

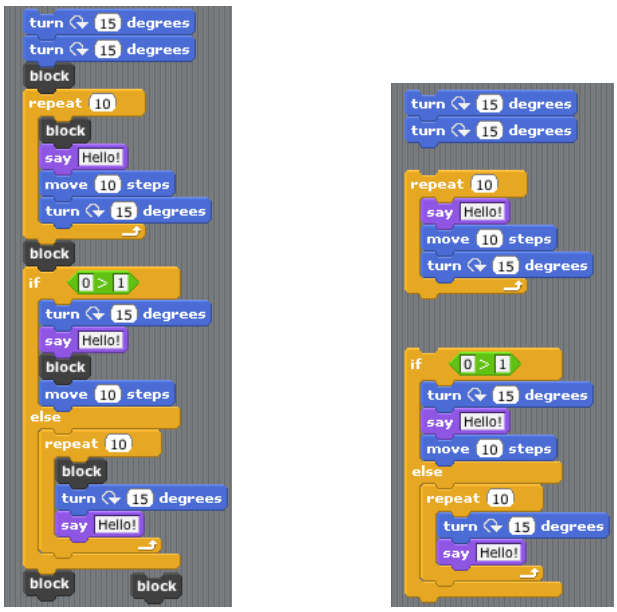
1. Quando o bloco está livre na área de *scripting*;
2. Quando o bloco está no topo de um *script*;
3. Quando o bloco é o último bloco de um *script*;
4. Quando o bloco está no meio de um *script*;
5. Quando o bloco é o último bloco de um *script* inserido num bloco em forma de 'C' ou num dos ramos do bloco **if-else**;
6. Quando o bloco é o primeiro bloco de um *script* inserido num bloco em forma de 'C';
7. Quando o bloco está no meio de um *script* inserido num bloco em forma de 'C';
8. Quando o bloco é o primeiro bloco do ramo `true` dum bloco **if-else**;
9. Quando o bloco está no meio de um *script* inserido no ramo `true` dum bloco **if-else**;
10. Quando o bloco é o primeiro bloco do ramo `false` dum bloco **if-else**;
11. Quando o bloco está no meio de um *script* inserido no ramo `false` dum bloco **if-else**.

Para os casos 1, 2, 3 e 4, a solução passa por remover o bloco da área de *scripting*, mantendo nesta os blocos aos quais o bloco estava ligado, isto é, o bloco imediatamente acima e abaixo deste. O espaço ocupado pelo bloco fica vazio, o que significa que, para o caso 4, o *script* fica dividido em dois na zona onde se encontrava o bloco a remover. Para os restantes casos, remove-se o bloco do *script* onde ele se encontra (esteja em que posição estiver nesse *script*: topo, fim ou meio), e o espaço que ficaria vazio é compensado por uma fusão de blocos: para o caso em que o bloco a remover é o primeiro ou último do *script*, o bloco em forma de 'C' ou **if-else** que o

contém adapta-se para passar a conter o *script* resultante da eliminação do bloco; para o caso em que o bloco está no meio, os dois *scripts* resultantes são unidos no ponto de separação e o bloco que o contém adapta-se ao novo tamanho do *script*. Para o desenvolvimento destas funcionalidades tira-se partido dos métodos `firstBlockList` e `firstBlockList:` da classe `CBlockMorph`, respectivamente acessor e modificador do *script* de blocos contido dentro dum bloco em forma de 'C', bem como os métodos `trueBlockList` e `trueBlock:` da classe `IfElseBlockMorph`, respectivamente acessor e modificador do *script* de blocos do ramo `true` dum bloco **if-else**, e os métodos `falseBlockList` e `falseBlock:`, que fazem o mesmo para o ramo `false`.

Este é o procedimento que é realizado para um bloco do **Palco** e para um bloco local de um *Sprite*. Para um bloco global aos *Sprites*, para além de ser executado todo este procedimento para o *Sprite* ao qual o bloco está associado, é também executado para cada um dos restantes *Sprites*. Para além disso, todos os *Sprites* que estivessem a consultar o bloco vêm a variável `inspectedBlock` tomar o valor `nil`, para indicar que já não estão a consultar nenhum bloco, e o texto do corpo do bloco associado à bandeira azul, relativo a cada um desses *Sprites*, toma o seu valor por omissão. Caso o bloco seja local, estes últimos passos são tomados apenas para o *Sprite* associado ao bloco. Por fim actualiza-se a categoria de blocos **MyScripts**.

As Figuras 5.40a e 5.40b ilustram o efeito obtido.



(a) *Script* contendo o SDU **block**.

(b) O mesmo *script* anterior, após a remoção do bloco **block**.

Figura 5.40: Eliminação de um SDU.

5.4.13 Escrita e leitura de projectos

A tarefa descrita nesta secção refere-se à possibilidade de gravar e ler de um ficheiro um projecto Scratch, sendo também gravada e carregada a informação relativa aos novos elementos

adicionados, nomeadamente, o estado da aba **Test** e os SDUs. Como esta problemática ainda envolve alguma complexidade, aborda-se este processo para a aba **Test** e para os SDUs de forma separada.

Em primeiro lugar, aborda-se a questão de gravar. Pretende-se que, ao gravar em ficheiro um projecto Scratch, o estado da aba **Test** seja também gravado (isto é, os seus conteúdos) por forma a ser recuperado mais tarde. Para gravar um projecto pode-se recorrer às opções **Save** e **Save As** do menu **File**. Este menu está definido no método `ScratchFrameMorph»fileMenu:` (Código C.78). À opção **Save** está associado o método `saveScratchProjectNoDialog` (Código C.79) e à **Save As** o método `saveScratchProject` (Código C.80), ambos da classe `ScratchFrameMorph`. Ambos os métodos, após realizarem algumas operações relacionadas com o pedido do nome do ficheiro ao utilizador, a localização do mesmo, etc, invocam o método `writeScratchProject` da mesma classe. Este método começa por guardar alguns parâmetros relativos ao projecto tais como qual o objecto `ScriptableScracthMorph` a que pertence o *script* que se está a desenvolver, qual a aba activa, a categoria de blocos activa, posição do **Palco**, etc. De seguida, é invocado o método de classe `buildBlockSpecDictionary` da classe `ScriptableScracthMorph` que constrói dois dicionários: um em que associa o método selector de um bloco à sua especificação (`BlockSpecDict`), e outro em que associa o método selector à cor do bloco (`BlockColorDict`). Posteriormente, entra numa fase em que pára qualquer processo de execução de *scripts* que esteja a decorrer e converte os empilhamentos de blocos existentes na aba **Scripts** em tuplos, através do método `ScriptableScratchMorph»convertStacksToTuples`. Depois, guarda o estado do projecto para um ficheiro temporário através de uma *stream* aberta em modo binário. De seguida, o estado do projecto antes de ser gravado é reposto, convertendo os empilhamentos de tuplos para blocos através do método `ScriptableScratchMorph»convertTuplesToStacks`, sendo também repostos os valores dos parâmetros inicialmente guardados. Por fim, o ficheiro temporário é renomeado para o nome que o utilizador havia escolhido.

A primeira situação de conversão consiste na conversão dos empilhamentos de blocos da aba **Scripts** em tuplos (Código C.81). Para cada objecto do tipo `ScriptableScratchMorph` existente, isto é, para todos os *Sprites* e o **Palco**, é invocado o método `stop` sobre os seus blocos presentes na área de *scripting* da aba **Scripts**, por forma a parar algum processo que esteja em execução. De seguida é então feita a conversão dos empilhamentos de blocos em tuplos no método `convertStacksToTuples` (Código C.82). Tal como indicado na documentação, este método converte os empilhamentos de blocos numa colecção de pares (<ponto>,<tuplo>), que neste caso são representados como um `Array`. O elemento <ponto> indica a posição geométrica onde se encontra o primeiro bloco do empilhamento, relativamente à posição da área de *scripting*. O elemento <tuplo> provém da invocação do método `tupleSequence` que pega na sequência de blocos que formam um empilhamento e devolve-os na forma de tuplos, invocando sobre cada um o método `asBlockTuple`, cujo comportamento varia em função do tipo de bloco, mas que, essencialmente, devolve uma descrição do bloco na forma de um tuplo (`Array`). Portanto, o método em análise começa por testar se a variável de instância `blocksBin`, que representa a área de *scripting* da aba **Scripts**, já se encontra na forma de um `Array`, caso em que a conversão já ocorreu. Caso ainda não esteja, então são recolhidos os seus *submorphs* que sabem responder à mensagem `tupleSequence`, que tanto podem ser `BlockMorph`s (que neste caso são os blocos de topo dos empilhamentos) como `ScratchCommentMorph`s (comentários), sendo ambos separados em duas colecções distintas. De seguida ambas as colecções são processadas do modo já indicado (para os comentários é devolvido um `Array` descritivo, visto que estes não possuem blocos encaixados), sendo devolvida a colecção resultante da concatenação das suas respectivas conversões em tuplos, a qual é atribuída à variável `blocksBin`.

Para perceber melhor o esquema de conversão, verifique-se o empilhamento de blocos da Figura 5.41 e o resultado da sua conversão em tuplos.

Figura 5.41: Empilhamento de blocos.

Resultado:

```
((31@40.0 ((#EventHatMorph 'Scratch-StartClicked') (#forward: 10) (#say: UTF8['
    Hello!'])))).
```

Tendo este conjunto de procedimentos aplicados à aba **Scripts**, é necessário replicar um comportamento semelhante para a aba **Test**, quer invocando o método `stop` sobre os seus blocos relativos aos *Sprites* e ao **Palco**, quer convertendo esses mesmos blocos em tuplos, através do novo método `convertTestStacksToTuples` que é em tudo semelhante ao `convertStacksToTuples`, alterando apenas a variável de instância para aquela que representa a área de *scripting* da aba **Test** (Código C.83). O método `ScratchFrameMorph»writeScratchProject` vê assim a sua primeira fase de conversão alterada (Código C.84).

A situação inversa de conversão faz a conversão de tuplos para empilhamentos de blocos (Código C.85), através do método `convertTuplesToStacks` (Código C.86). Primeiro verifica se a variável `blocksBin` já foi convertida. Após verificar que tal ainda não ocorreu, guarda o conteúdo desta (a colecção de tuplos anteriormente gerada), sendo-lhe atribuída uma nova instância de `ScratchScriptsMorph` (classe que representa a área de *scripting*). De seguida, a colecção de tuplos é processada, sendo recriados os empilhamentos existentes (com o posicionamento prévio) e estes são colocados na área de *scripting*. Para recriar os empilhamentos é invocado o método `stackFromTupleList:receiver:` que devolve um empilhamento a partir de uma lista de tuplos. Para isso, itera sobre a lista de tuplos e invoca o método `blockFromTuple:receiver:` para transformar cada tuplo num bloco (Código C.87). O método `blockFromTuple:receiver:` possui diferente comportamento para os diferentes tipos de blocos. É neste método que são utilizados os dois dicionários criados anteriormente no método `buildBlockSpecDictionary`. Como na Secção 5.4.4 se criou o bloco associado à bandeira azul como sendo do tipo *hat* e com um novo evento associado, é necessário atentar na parte onde são processados esses tipos de blocos (Código C.88). Para este tipo de blocos é invocado um outro método chamado `hatBlockFromTuple:receiver:`. Neste é necessário adicionar uma condição para indicar que o novo bloco associado à bandeira azul deve ser construído usando o método `EventHatMorph»forStartScriptEvent`, desenvolvido para criar o novo bloco com o novo evento que lhe está associado (Código C.89).

Feita esta alteração e sabendo como se obtêm os empilhamentos da aba **Scripts** a partir de tuplos, tem de se adicionar um novo método, `convertTestTuplesToStacks`, que converte os tuplos relativos à aba **Test** (Código C.90). Este método invoca o `testStackFromTupleList:receiver:` da mesma classe, o qual é semelhante ao `stackFromTupleList:receiver:`, excepto na invocação do novo método `testBlockFromTuple:receiver:` (Código C.91). Este novo método é também ele semelhante ao que é usado na conversão relativa à aba **Scripts**, `blockFromTuple:receiver:`, apenas diferindo no método que é invocado no processamento de um tuplo representativo de um comentário, `testBlockWithID:` (Código C.92). Quando um comentário, representado pela classe `ScratchCommentMorph`, é convertido em tuplo, esse tuplo guarda 4 parâmetros fixos e um quinto parâmetro variável. Este parâmetro variável só existe no caso em que o comentário está associado a um dado bloco (Figura 5.42).

Para o caso em que existe, esse parâmetro guarda a referência para o bloco a que está associado. Durante a conversão de tuplo para blocos da aba **Scripts**, ao ser processado um comentário

Figura 5.42: Comentário associado ao bloco **turn <x> degrees**.

que possui esse parâmetro, vai ser executado o método `ScriptableScratchMorph»blockWithID:`, que devolve essa referência (Código C.93 e C.94). No entanto, este método apenas está a processar comentários existentes na aba **Scripts**, pois utiliza a variável que a representa, `blocksBin`. Daí surgir a necessidade de, no método `testBlockFromTuple:receiver:`, se ter a invocação de um novo método `testBlockWithID:`, que faz o mesmo que o `blockWithID:`, mas para a aba **Test** (Código C.95). Desta forma, comentários que estejam associados a blocos na aba **Test** são recuperados dos seus tuplos correctamente, ficando a sua associação ao bloco bem definida. Após a definição de todos estes métodos dos quais depende o `convertTestTuplesToStacks`, basta alterar a porção de código do método `writeScratchProject`, adicionando este novo comportamento à segunda fase de conversões (Código C.96).

O carregamento de um ficheiro é feito através da opção **Open** do menu **File**, dado pelo método `ScratchFrameMorph»fileMenu:`, já apresentado. A esta opção está associado o método `openScratchProject` da mesma classe. Este método, inicialmente, verifica se o projecto actual está vazio, isto é, se não tem *scripts*, variáveis, trajes especiais, sons e se tem apenas 1 *Sprite*, invocando, para tal, o método `ScratchFrameMorph»projectIsEmpty`. No último bloco de instruções deste método é feita a verificação relativa aos *scripts*. Como agora existe uma nova aba cujo conteúdo não deve ser menosprezado, convém fazer um teste semelhante para essa aba (Código C.97). Após fornecer ao utilizador a opção de gravar o projecto actual e abrir o explorador do sistema de ficheiros para que se seleccione o ficheiro desejado, este método invoca o método `openScratchProjectNamed:`. Por sua vez, este método abre uma *stream* em modo binário sobre o ficheiro escolhido, lê o conteúdo do ficheiro para um *array* de bytes (representado pela classe `ByteArray`) e invoca o método `ScratchFrameMorph»extractProjectFrom:`, passando-lhe como argumento esse *array* de bytes. O método termina fazendo inicializações de vários parâmetros relativos ao projecto (Código C.98). Tal como indica a documentação, o método `extractProjectFrom:` devolve um **Palco** (que poderá ter *Sprites* ou não). Após efectuar a leitura a partir da *stream* aberta, a qual difere em função da versão do Scratch associada ao projecto a ler, faz-se um despiste de erros testando se o resultado da leitura é um **Palco**. Depois é invocado o método `buildBlockSpecDictionary` e fazem-se as duas conversões já abordadas. Uma vez conhecido como este processo de conversão funciona, apenas se tem de adicionar os dois novos métodos construídos para a conversão associada à aba **Test**, na ordem pela qual as conversões entre tipos de representações são feitas (primeiro de empilhamentos para tuplos e só depois no sentido inverso) (Código C.99).

Apesar das alterações efectuadas no processo de leitura e escrita de projectos, este ainda não está terminado. É preciso ter atenção para que as alterações efectuadas não originem quebra de compatibilidade para com os projectos desenvolvidos no Scratch original, isto é, para que não deixe de ser possível ler esses projectos pelo motivo de neles não existir a aba **Test** e, por conseguinte, o algoritmo de desserialização não conseguir extrair do conjunto de bytes desse projecto informação relativa a essa aba. A retrocompatibilidade é um ponto extremamente importante, pois permite que o utilizador aproveite um dos inúmeros projectos criados no Scratch original pela comunidade e lhe acrescente componentes existentes apenas nesta modificação do Scratch. É por isso necessário que o mecanismo de serialização saiba como guardar/ler o estado

dos novos componentes existentes nos projectos.

Analisando a implementação do BYOB para este problema, verifica-se que a solução utilizada passa por introduzir uma variável de classe na classe `ScriptableScratchMorph` responsável por indicar qual o tipo de projecto a ler (se é um desenvolvido no Scratch original ou se é um desenvolvido numa modificação deste, neste caso o BYOB), sendo essa variável alterada em determinadas situações. A variável de classe é afectada com um símbolo que indica qual a versão do projecto que se está a manipular: `#scratch` para um projecto do Scratch original, e `#byob` para um projecto realizado com o BYOB. No Scratch original todos os projectos são gravados com um cabeçalho onde consta a *string* "ScratchVXX", sendo XX o número da versão do Scratch em que o projecto é criado. Na versão mais recente, XX tem o valor 02, resultando na *string* "ScratchV02". No BYOB essa *string* foi alterada para "BloxExpV01". Com isto, os projectos criados com o BYOB são gravados com esta *string*, o que significa que, tal como esperado, deixam de ser compatíveis com o Scratch original. No processo de leitura, para que se possam ler projectos quer do Scratch original quer do BYOB, é feito um teste por forma a identificar qual a *string* presente no cabeçalho do ficheiro, activando-se opções diferentes de leitura conforme o tipo de projecto a ler (através da tal variável de classe). Com isto, consegue-se tirar partido da grande quantidade de projectos Scratch já existentes, podendo estes ser carregados e modificados. Apenas se perde compatibilidade na gravação dos projectos, mas faz todo o sentido que tal aconteça pois tratam-se de projectos que possuem novas funcionalidades que o Scratch original desconhece.

No entanto, esta solução não é a melhor. Para além de seguir uma prática não muito de acordo com a forma como o sistema está implementado, obriga a fazer alterações que não são muito claras, perdendo-se um pouco de legibilidade do código assim como vantagens do desenvolvimento orientado a objectos. Analisando a solução implementada pelo Panther, e confrontando as duas, conclui-se que a do Panther é bastante mais simples e vai de encontro ao modo como o sistema de serialização está implementado, até mesmo para permitir futuras extensões das classes do Scratch. De seguida analisa-se essa mesma solução à medida que é implementada para este caso de estudo.

O segredo para a simplicidade da implementação desta funcionalidade no Panther reside na versão da classe a serializar. Durante o processo de escrita do projecto em ficheiro é invocado o método `ObjStream≫writeObject:objEntry:` (Código C.100). Neste método é obtido um símbolo de um método, `putUserObj:id:`, o qual é associado, no método de classe `initialize` de `ObjStream`, às classes definidas pelo utilizador (Código C.101). No conjunto dessas classes definidas pelo utilizador encontram-se as classes `ScratchStageMorph` e `ScratchSpriteMorph` (Código C.102). Portanto, ao escrever em ficheiro instâncias dessas duas classes, vai ser executado o método `fieldsVersion` de cada uma dessas classes. Este método devolve um valor inteiro que corresponde à versão da classe, isto é, um número que, sempre que é criada uma nova versão da classe (isto é, com mais variáveis de instância que serão serializadas, alterando, portanto, o formato de serialização das suas instâncias) deve ser incrementado pelo programador. Isto permite saber qual a estrutura dessas classes em determinado projecto (e.g., um projecto criado nas primeiras versões do Scratch tem uma versão destas classes inferior à versão actual, porque entretanto foram acrescentadas novas variáveis). Para a classe `ScratchStageMorph` é devolvido o valor 5, e para a classe `ScratchSpriteMorph` o valor 3. Estes valores são então guardados em ficheiro. Seguidamente, o método `writeObject:objEntry:` faz com que o método `putUserObj:id:` seja invocado sobre a instância que está a ser gravada, o que leva a que o método `storeFieldsOn:` da classe da instância seja executado (Código C.103). Em ambas as classes, o método `storeFieldsOn:` invoca o método homólogo da superclasse `ScriptableScratchMorph`, que trata de gravar algumas das suas variáveis de instância, através do método `Object≫storeFieldsNamed:on:`. Em seguida, este último método é invocado em

ambas as subclasses para gravar as variáveis que lhes são específicas. De notar aqui um pormenor, que será clarificado mais à frente, que consiste no facto de haver variáveis de instância da classe `ScriptableScratchMorph`, como `sceneStates` e `lists`, que são gravadas, não nesta classe, como seria de esperar, mas em ambas as suas subclasses.

No processo inverso, isto é, de leitura do ficheiro, durante a execução do método de instância `ObjStream»readObjFrom:showProgress:`, é invocado o `ObjStream»readObjectRecord` (Código C.104). Neste método, a versão da classe é lida da *stream* e guardada. Após a execução deste método, no método `readObjFrom:showProgress:` é invocado o `initializeUserDefinedFields:` da mesma classe, o qual inicializa a instância que está a ler através da invocação do método `initFieldsFrom:version:`, ao qual passa como parâmetro a versão da classe lida (Código C.105). A partir daqui sabe-se que esse método é invocado para as classes `ScratchStageMorph` (**Palco**) e `ScratchSpriteMorph` (*Sprites*). Para a primeira, cuja versão tem o valor 5, a inicialização segue o esquema do Código C.106.

O primeiro passo consiste, analogamente ao processo de escrita, na invocação do mesmo método da superclasse que inicializa as variáveis comuns às duas classes em questão. No entanto, e aqui é que reside o ponto principal da solução, variáveis comuns como a `sceneStates` e `lists` são incializadas no método da subclasse (analogamente, e como se tinha visto anteriormente, são também gravadas pelos métodos da subclasse). Mais, olhando para o código apresentado, verifica-se que estas só são inicializadas caso a versão da classe seja a 5. Uma situação semelhante ocorre no mesmo método da classe `ScratchSpriteMorph` (Código C.107). Uma primeira solução passa por inicializar a variável que referencia a aba **Test**, `testBlocksBin`, na superclasse `ScriptableScratchMorph`, pois ela é comum às duas subclasses. No entanto, dado que esta classe não define o método `fieldsVersion` e dado que não se criam suas instâncias durante a execução, mas antes instâncias das suas duas subclasses, ao ser invocado o seu método `initFieldsFrom:version:`, este vai receber como segundo parâmetro o valor da versão da classe relativo à subclasse que está a ser processada, o qual é diferente entre as duas subclasses. Isto resulta que, por exemplo, num caso receba o valor 3 e no outro o valor 5. Como tal, não é possível implementar nesta classe uma lógica semelhante de inicialização das variáveis em função da versão da classe. Poder-se-ia utilizar um número maior mas não se resolveria definitivamente o problema, apenas se deixava um remendo que poderia dar problemas mais tarde. Como tal, opta-se pela solução de inicializar esta variável em ambas as subclasses, tal como as variávels `sceneStates` e `lists`. Assim, altera-se o método `ScratchSpriteMorph»fieldsVersion` para que passe a devolver o valor 4 (Código C.108) e altera-se o método `initFieldsFrom:version:` da mesma classe para que passe a diferenciar as versões 3 e 4 (Código C.109). Um processo semelhante efectua-se na classe `ScratchStageMorph`, alterando a sua classe para a versão 6 (Código C.110 e C.111).

Assim como a variável `testBlocksBin` passa a ser lida nas subclasses, também passa a ser escrita nelas, seguindo o exemplo das variáveis `sceneStates` e `lists`. Para isso, tem que se alterar o método `storeFieldsOn:` das classes `ScratchStageMorph` e `ScratchSpriteMorph`, adicionando a variável à lista a ser gravada (Código C.112 e C.113). Na classe `ScriptableScratchMorph` apenas se tem de modificar o método de escrita, `storeFieldsOn:`, para que, analogamente ao que é feito para a aba **Scripts**, o *owner* da aba **Test** seja guardado no início e recuperado no fim da serialização (Código C.114).

Desta forma é mantida a compatibilidade, podendo ser lidos projectos desenvolvidos no Scratch original e nesta nova extensão, sem necessidade de alterar o cabeçalho dos ficheiros nem fazer grandes alterações. Os projectos criados ficam a ser apenas compatíveis com esta modificação do Scratch. Fica também claro como uma futura extensão deste ambiente deve proceder para escrever e ler os seus projectos. Na verdade, todo este processo está documentado, de forma genérica e mais abstracta, na documentação da classe `ObjStream` que, resumidamente,

diz:

> "Eu posso serializar uma colecção de objectos interligados para uma *stream* ou reconstruir a rede de objectos original a partir da sua forma serializada. Isto permite que as estruturas dos objectos possam ser gravadas em ficheiros, transmitidas através de uma rede, etc. Eu suporto evolução das classes definidas pelo utilizador através dos números de versão das classes, tornando possível detectar e converter formatos antigos de objectos.
>
> Os formatos de armazenamento dos objectos dividem-se em 3 categorias:
>
> - Valores imediatos: as constantes `nil`, `true`, `false`, inteiros e valores em vírgula flutuante;
> - Objectos de formato fixo cuja serialização é tratada por esta classe (`ObjStream`);
> - Objectos que tratam da sua própria serialização e cujo formato pode alterar-se ao longo do tempo.
>
> Assume-se que os objectos da segunda categoria têm formatos de armazenamento estáveis, logo não precisam de um número de versão da classe. Objectos da última categoria suportam serialização definida pelo utilizador. Devem implementar 3 métodos de instância: `storeFieldsOn:`, `initFieldsFrom:version:` e `fieldsVersion`. De um modo geral, qualquer alteração ao formato de serialização de um objecto implica adicionar novas versões dos métodos `storeFieldsOn:` e `initFieldsFrom:version:`, incrementando o número de versão devolvido pelo `fieldsVersion`.
>
> A classe de cada objecto desta categoria deve constar na tabela de IDs das classes. Para assegurar que objectos antigos podem ser lidos, assim que um ID é atribuído a um objecto, esse ID não pode ser modificado, e mesmo que a classe seja eliminada, o seu ID não deve ser usado para outra classe.
>
> Máximos:
>
> - Tipos de objectos (classes): 255
> - Atributos das classes definidas pelo utilizador: 255
> - Número de objectos: $2^{24} - 1$
> - Tamanho do objecto indexável: $2^{32} - 1$"

O último parágrafo toma especial importância no processo de serialização dos SDUs, uma vez que no âmbito de desenvolvimento das funcionalidades a eles associadas foram criadas novas classes.

Todo o processo de escrita de um projecto em ficheiro e sua posterior leitura, com a aba **Test** incluída, já foi abordado, pelo que agora apenas se esclarece a parte relativa aos SDUs.

Os SDUs são guardados na classe `ScriptableScratchMorph` através da variável de instância `userBlocks`. Desse modo, para serem serializados e desserializados, é necessário acrescentar essa variável à lista de variáveis que são usadas no processo de escrita e leitura, nos métodos `storeFieldsOn:` e `initFieldsFrom:version:` das suas duas subclasses, à semelhança do que foi feito para a variável da aba **Test**. O mesmo acontece para a variável `inspectedBlock`, que contém o nome do SDU que está a ser consultado para um dado `ScriptableScratchMorph`. Pode-se manter a mesma versão das classes, adicionando as variáveis à `testBlocksBin`, passando estas três a representar as novas variáveis introduzidas nas versões mais recentes das classes. Assim, na classe `ScratchStageMorph` os métodos passam a apresentar a configuração presente

no Código C.115 e C.116. Na classe `ScratchSpriteMorph` ficam de acordo com o Código C.117 e C.118.

Como a variável `userBlocks` representa um dicionário que associa uma *string* a uma instância da classe `UserBlockDefinition`, para que a serialização seja feita de forma correcta, é necessário adicionar 3 métodos no protocolo `object i/o` desta classe, por forma a indicar como as instâncias desta classe devem ser serializadas/desserializadas. O primeiro método a adicionar é o `fieldsVersion`, o qual deve devolver o valor 1 (Código C.119). Os outros dois métodos são o `initFieldsFrom:version:` e `storeFieldsOn:`, cuja função já é conhecida. No entanto, como esta classe não é subclasse de `Morph` mas de `Object`, não deve invocar os métodos homólogos da superclasse (Código C.120 e C.121).

É preciso também adicionar esta classe ao conjunto de classes definidas pelo utilizador que podem ser serializadas. Este conjunto de classes está definido no método de classe `userClasses` de `ObjStream`, o qual é usado para preencher os dicionários `IDToClassEntry` e `NameToClassEntry` da mesma classe, durante a execução do método de classe `initialize` (Código C.122). Estes dicionários são úteis durante a serialização. Este conjunto de classes está definido como um *array* que guarda pares, em que o primeiro elemento é o ID da classe e o segundo é o nome da classe. Os IDs para as classes definidas pelo utilizador podem variar entre 100 e 255 (Código C.123), informação esta dada no método de classe `fixedFormatClasses` da mesma classe, que por sua vez devolve um *array* de registos de classes de formato fixo (Código C.124). Este *array* é também usado para preencher os dicionários supracitados. Sabendo isto, adiciona-se a classe `UserBlockDefinition` com um ID ainda não utilizado no método `userClasses` (Código C.125). Há ainda outras classes que precisam de ser registadas, pois o facto de não estarem provoca erros no processo de leitura e escrita. Essas classes são a `ChoiceOrExpressionArgMorph`, `StringFieldMorph`, `ImportCBlockMorph`, `IfElseBlockMorph`, `UserCommandBlockMorph`, `AttributeArgMorph` e `UTF32`. Todas excepto a `UTF32` são adicionadas no método `userClasses` (Código C.126). A classe `UTF32`, por sua vez, é registada no método de classe `fixedFormatClasses`, por analogia com a classe `UTF8` que lá se encontra, sendo-lhe associado o mesmo método de leitura e escrita desta última (Código C.127). Para o método de leitura, `ObjStream»getBytes:id:`, adiciona-se um caso extra para a nova classe, de modo a criar uma instância de `UTF32` (Código C.128).

Tendo isto feito, seria expectável que as classes fossem registadas ao criar a instância de `ObjStream` usada para escrever/ler, pois a criação de uma instância através da invocação do método `new` leva à execução do método de instância `initialize` da classe da instância. No entanto, aqui é que reside o problema. O método `initialize` que preenche os dicionários anteriormente referidos com os valores fornecidos pelos métodos `userClasses` e `fixedFormatClasses`, é um método de classe e não de instância. Como tal, não é executado implicitamente no processo de criação da instância de `ObjStream`. "Embora os métodos de classe de inicialização sejam executados automaticamente quando o código é carregado para a memória, não são executados automaticamente da primeira vez que são escritos no *browser* e compilados nem quando são editados e recompilados" [BDNP07]. Portanto, é necessário executar o método explicitamente num `Workspace` (Código C.129). Deste modo os dicionários já ficam actualizados com as novas classes.

Também é preciso indicar como os SDUs são processados na conversão de blocos para tuplos e de tuplos para blocos. Como estes novos blocos são representados pela nova classe `UserCommandBlockMorph`, é necessário redefinir o método `asBlockTuple` da sua superclasse `CommandBlockMorph`, dentro do protocolo `stack/tuple conversion`, para indicar como estes blocos devem ser convertidos para tuplos (Código C.130). Neste método converte-se o corpo do bloco definido pelo utilizador num *array* de tuplos, em que cada um é a representação de cada bloco do corpo sob a forma de tuplo (método `BlockMorph»tupleSequence`). O método

devolve um *array* com 3 elementos: o selector associado ao bloco, `#userBlockWarn`, a *string* de especificação do bloco (o seu nome), e o *array* de tuplos previamente criado. Para recuperar um bloco definido pelo utilizador a partir de um tuplo tem de se acrescentar uma condição aos métodos `blockFromTuple:receiver:` e `testBlockFromTuple:receiver:` da classe `ScriptableScratchMorph`, indicando o que deve ser feito ao ler o selector `#userBlockWarn` (Código C.131 e C.132). Quando o selector corresponde ao símbolo `#userBlockWarn` é construída uma nova instância de `UserCommandBlockMorph` e são-lhe atribuídos vários parâmetros, onde se inclui a *string* de especificação guardada anteriormente no tuplo. Por fim, o corpo do bloco é recuperado processando cada tuplo do *array* de tuplos previamente gerado, sendo criado um empilhamento com os blocos resultantes da conversão desses tuplos para blocos.

Feitas estas alterações, já é possível ler e escrever projectos que possuam blocos definidos pelo utilizador.

Apenas resta resolver uns pormenores. O primeiro diz respeito à conservação do texto do corpo do bloco associado à bandeira azul entre escritas e leituras de um projecto. Verifica-se que, quer na escrita quer na leitura, esse texto volta à sua forma inicial. Tal não é pretendido pois, supondo que o utilizador grava o projecto enquanto está a consultar um SDU, é de bom senso esperar que, ao ler o projecto, tal informação seja mantida e mostrada ao utilizador. Este problema ocorre, mais uma vez, devido à diferença de identidade entre o bloco associado à bandeira azul que surge na aba **Test** e aquele que o Sprite/**Palco** referencia. Como tal, é necessário recorrer à solução que passa por forçar a que esses dois blocos sejam o mesmo. Para isso, no método `ScratchFrameMorph≫writeScratchProject`, após ser feita a conversão dos tuplos para empilhamentos para um dado *Sprite*/**Palco**, aplica-se a referida solução, e de seguida invoca-se o método `addBlockName:` sobre o bloco associado à bandeira azul referenciado por esse *Sprite*/**Palco**, passando-lhe como argumento o valor da variável `inspectedBlock` que, em função de se estar a consultar ou não um dado SDU, altera ou não o texto do corpo do bloco associado à bandeira azul (Código C.133). Uma solução semelhante tem de ser adoptada para o processo de leitura, após realizado o processo de conversão no método `ScratchFrameMorph≫extractProjectFrom:` (Código C.134). Com estas alterações, o texto do bloco já se comporta da forma correcta. Para resolver o último pormenor tem de se alterar o método `ScratchFrameMorph≫projectIsEmpty`, para que, ao verificar se um projecto está vazio, seja feito um teste que indique se existem blocos definidos pelo utilizador (Código C.135).

Após realizadas todas estas modificações, passa a ser possível gravar e, posteriormente, ler projectos desenvolvidos nesta extensão do Scratch, sendo guardada/recuperada a informação relativa aos novos elementos (aba **Test** e SDUs).

Os projectos criados com o Scratch são gravados em ficheiro com a extensão `.sb`. Para identificar os ficheiros criados com esta modificação do Scratch, define-se uma nova extensão para os ficheiros: `.extsb`. É preciso indicar que a nova extensão `.extsb` deve ser reconhecida como uma extensão de ficheiros válida, para que estes possam ser mostrados nas janelas de diálogo que exploram o sistema de ficheiros, entre outros, bem como é preciso alterar a extensão com que os ficheiros passam a ser guardados. Fazendo uma pesquisa no código fonte pela *string* "sb" são encontrados 10 métodos que têm de ser alterados. O método `createScratchFileChooserFor:saving:` da classe `ScratchFileChooserDialog` abre um explorador do sistema de ficheiros para que possam ser escolhidos ficheiros do Scratch. Para reconhecer a nova extensão e, por conseguinte, os ficheiros que a usam, é necessário adicionar a sua designação (Código C.136). A mesma coisa tem de ser feita no método `ScratchFrameMorph≫importScratchProject` (Código C.137). O método de classe `ScratchFileChooserDialog≫confirmFileOverwriteIfExisting:`, que permite escrever sobre um ficheiro já existente ou escolher outro nome para o ficheiro, tem de ser alterado para que o ficheiro seja gravado com a nova extensão (Código C.138). O

mesmo tem de ser feito no método de classe `saveScratchFileFor:` da mesma classe (Código C.139). O método `ScratchFrameMorph»nameFromFileName:`, que retira a extensão ao nome do ficheiro, também tem de ser alterado (Código C.140). O método `processDroppedFiles` da classe `ScratchFrameMorph` também deve reconhecer ficheiros com a nova extensão (Código C.141). Os métodos `saveScratchProject` e `saveScratchProjectNoDialog` devem substituir a extensão antiga pela nova (Código C.142 e C.143). O método `ScratchFrameMorph»startup`, invocado quando se abre o ambiente do Scratch, também deve reconhecer algum ficheiro com a nova extensão que deva abrir ao iniciar o ambiente (Código C.144). Por último, a mesma extensão também deve ser reconhecida ao escrever os sumários de vários ficheiros (Código C.145).

Fica assim terminada esta remodelação das extensões dos ficheiros.

5.4.14 Outras operações relacionadas com ficheiros

Ainda relacionadas com a leitura e escrita de ficheiros surgem as opções de exportar um *Sprite*, importar um projecto e largar um ficheiro sobre o ambiente Scratch por forma a abri-lo.

Exportar um *Sprite* pode ser feito de 3 formas diferentes: 1) através da opção **Export Sprite** do menu **File**; 2) através do menu contextual do *Sprite* que se encontra no **Palco**; 3) através do menu contextual do ícone do *Sprite*. A primeira opção faz executar o método `exportSprite` da classe `ScratchFrameMorph`. De seguida, este invoca o `exportObject` da classe `ScriptableScratchMorph`. Este último método é aquele que é invocado pelas opções 2) e 3) atrás referidas, de modo que a cadeia de métodos subsequente é comum às 3 opções. Seguidamente, o `exportObject` invoca o método `copyForExport` da classe `ScratchSpriteMorph`. Este acaba por invocar o mesmo método da superclasse, `ScriptableScratchMorph` (Código C.146). Mais uma vez se tem de replicar o processamento que é feito aos blocos da aba **Scripts** para a aba **Test**, quer invocando os métodos `stop` e `clearMorphReference`, quer convertendo também os seus blocos (Código C.147). Feito isto, o *Sprite* exportado passa a carregar consigo os *scripts* da aba **Test**.

A opção de importar um projecto consiste em fundir os componentes de outro projecto (variáveis, *Sprites*, trajes, *scripts*, etc) com os do projecto actual. Esta opção existe no menu **File** com o nome de **Import Project**, tendo a si associado o método `importScratchProject` da classe `ScratchFrameMorph`. Este, após solicitar a escolha do ficheiro onde está guardado o projecto, invoca o método de instância `importSpriteOrProject:` da mesma classe. Este método lê o conteúdo do ficheiro escolhido e, de seguida, invoca o método `extractProjectFrom:` já anteriormente analisado, passando-lhe o conteúdo do ficheiro como parâmetro (funciona numa lógica semelhante à abertura de um ficheiro). Posteriormente, têm de ser corrigidas todas as referências ao **Palco** devolvido na extracção do projecto, por forma a que os blocos da aba **Scripts** associados a esse **Palco** e aos seus *Sprites* passem a estar associados aos do projecto actual. O mesmo tem de ser feito para a nova aba (Código C.148). Mais tarde são adicionados ao projecto actual os *scripts* da aba **Scripts** existentes no **Palco** importado (Código C.149). Aqui o método `ScriptableScratchMorph»addStack:` é responsável por criar no **Palco** actual cópias dos *scripts* a importar (Código C.150). Este método coloca o *script* recebido como parâmetro imediatamente abaixo do último existente na aba **Scripts**. Mais uma vez se tem de adicionar código ao método `importSpriteOrProject:` para importar os *scripts* da aba **Test**. No entanto, para impedir que o bloco de topo associado à bandeira azul seja importado, originando a existência de dois blocos desse tipo na aba **Test** do projecto actual, é necessário fazer um teste para que, para o *script* encabeçado por esse bloco, apenas sejam copiados os blocos que se encontram encaixados por baixo dele (Código C.151). Para esta situação é necessário criar um método novo de nome `addTestStack:`, que é semelhante ao `addStack:`, mas que trata de colocar o *script* passado como parâmetro dentro da aba **Test** (Código C.152). A importação de

um projecto fica assim concluída.

O Scratch permite ainda a possibilidade de se arrastar um ficheiro correspondente a um projecto Scratch ou a um elemento de *media* (som ou imagem) para cima da sua interface, sendo o respectivo conteúdo carregado e apresentado. Este evento é tratado pelo método `processDroppedFiles` da classe `ScratchFrameMorph`. Caso o ficheiro corresponda a um projecto Scratch é invocado o método `openScratchDroppedProjectNamed:`, o qual acaba por invocar o método de instância `openScratchProjectNamed:`, que já foi alvo de análise. Para os elementos de *media*, a responsabilidade é delegada no método `ScriptableScratchMorph»importMedia:`, mas como este não interfere com a aba **Test** não é alvo de análise.

Após estas alterações, juntamente com as da Secção 5.4.13, verifica-se que os métodos que fazem uma dada conversão (empilhamentos para tuplos e vice-versa) têm de ser sempre invocados em conjunto, ou seja, é necessário invocar o método que faz a conversão relativa à aba **Scripts** e o que faz a conversão para a aba **Test**. Desta forma, efectua-se uma pesquisa no código pela invocação dos métodos `convertStacksToTuples` e `convertTuplesToStacks` para adicionar, logo após estes e para o caso de conversão adequado, os novos métodos criados, `convertTestStacksToTuples` e `convertTestTuplesToStacks`. Assim, altera-se o método `PaintCanvas»extractProjectFrom:`, semelhante ao método do mesmo nome presente na classe `ScratchFrameMorph` (Código C.153). O método `ScratchFrameMorph»setLanguage:`, que altera o idioma em que o Scratch está localizado e actualiza os blocos, também sofre alterações (Código C.154). Para sanar ainda qualquer funcionalidade que tivesse passado despercebida, realiza-se uma pesquisa no código pela variável que representa a área de *scripting* da aba **Scripts** por forma a verificar se existe algum método em que é necessário adicionar comportamento para suportar a aba **Test**. Deste modo, encontra-se o método `ScratchStageMorph»stopAllProcesses` que pára todos os processos em execução e no qual tem de ser adicionado o suporte à aba **Test** (Código C.155).

O utilizador pode gerar ficheiros de texto contendo um sumário de um ou vários projectos Scratch (da mesma pasta) através das opções **Write Project Summary** e **Write Multiple Project Summaries**, que se encontram no menu **File**, mas apenas são visíveis caso se pressione a tecla `Shift`. Existem dois métodos que intervêm neste processo de escrita: o `ScratchFrameMorph»writeSummaryTotalsOn:` e `ScriptableScratchMorph»printSummaryOn:`. Estes dois métodos acabam por ser executados após a invocação do método `writeSummaryFile`, definido como o método associado à opção de escrever um sumário do projecto (Código C.156). O método `writeMultipleSummaries` invoca o `writeSummaryFile` para vários projectos. O primeiro método, `writeSummaryTotalsOn:`, escreve no ficheiro de texto o número total de elementos do projecto (*Sprites*, *scripts*, trajes únicos e sons únicos). Para os *scripts*, este método tem apenas em conta a existência de uma única aba de *scripting*, mas como agora existem duas é necessário introduzir uma alteração no código para que a contagem dos *scripts* seja feita para as duas abas, separadamente. Para isso, introduz-se a variável auxiliar `testStackCount` que guarda o número de *scripts* da aba **Test** e a contagem dos *scripts* passa a ser escrita em linhas distintas para as duas abas (Código C.157). O segundo método, `printSummaryOn:`, é usado para escrever a informação individual de cada *Sprite* e do **Palco** em ficheiro. Mais uma vez, a parte dedicada à escrita de informação sobre os *scripts* é replicada para a aba **Test** (Código C.158). Feitas estas mudanças, o ficheiro de texto gerado com o sumário de um projecto que tem 1 *Sprite* e 1 **Palco**, com *scripts* em ambas as abas, tem o seguinte aspecto:

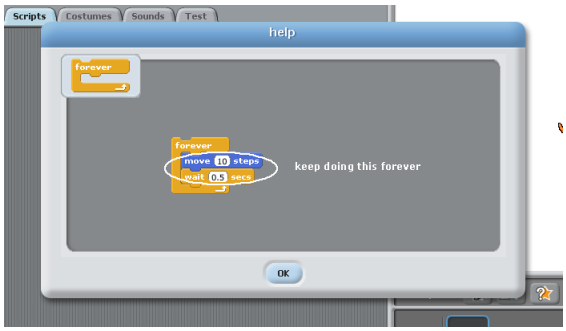
```
Project: bl
Author:
Scratch: 1.4 (source code of 23-Sep-09)
History:
    2011-8-25 00:11:05  save    bl
    2011-8-25 00:11:33  save    bl
    2011-8-25 00:11:45  save    bl
    2011-8-25 00:11:58  save    bl
    2011-8-25 00:12:52  save    bl
    2011-8-25 00:13:29  save    bl
    2011-8-25 00:13:55  save    bl
    2011-8-25 00:14:13  save    bl
    2011-8-25 00:15:12  save    bl

Totals:
    Sprites: 1
    Scripts Stacks: 5
    Test Stacks: 2
    Unique costumes: 2
    Unique sounds: 2
--------
Sprite: Stage
  Costumes (1):
    background1 (480x360)
  Sounds (1):
    pop (0:00:00)
  Scripts Stacks (2):
    change "color" effect by 25

    import
        stage2

  Test Stacks (1):
    when I receive "Scratch-StartScriptClicked"
        change "color" effect by 25
        play note 60 for 0.5 beats
    end
--------
Sprite: Sprite1
  Costumes (1):
    costume1 (73x36)
  Sounds (1):
    meow (0:00:01)
  Scripts Stacks (3):
    move 10 steps
    import
        global

    play drum 48 for 0.2 beats
    turn  15 degrees

    cat

  Test Stacks (1):
    when I receive "Scratch-StartScriptClicked"
        repeat 10
            play drum 48 for 0.2 beats
            turn  15 degrees
    end

--------
```

As funcionalidades de exportação de *Sprite* e importação de projecto não abarcam os SDUs devido a problemas de implementação para os quais não se encontrou solução dentro do tempo disponível.

5.4.15 Ecrã de ajuda

Associado a cada bloco do Scratch existe um menu contextual com a opção **help**, a qual faz surgir no ambiente uma janela descritiva da funcionalidade do bloco em questão, como se pode ver na Figura 5.43.

(a) Menu contextual dum bloco, com a opção **help**.



(b) Ecrã de ajuda do bloco **forever**.

Figura 5.43: Ecrã de ajuda activado via menu contextual do bloco.

Dado que agora existem novos blocos que ainda não têm esta janela de ajuda, faz todo o sentido que elas sejam feitas, para se manter este comportamento e também para que o utilizador se possa sentir auxiliado em caso de dúvida da utilização dos novos blocos. Mas primeiro é preciso entender como são criadas estas janelas de ajuda.

A opção **help** está definida em `BlockMorph`, `HatBlockMorph` e `CommandBlockMorph` através do método `rightButtonMenu` (Código C.159). As classes que representam os novos tipos de blocos, `ImportCBlockMorph` e `UserCommandBlockMorph`, são subclasses de `CommandBlockMorph`, logo já possuem a si associado o menu contextual, pelo que não é necessário defini-lo explicitamente. A opção **help** faz executar o método `BlockMorph»presentHelpScreen` (Código C.160). Este, por sua vez, invoca o método com o mesmo nome da classe `ScratchFrameMorph`, passando-lhe como argumento o resultado da invocação do método `helpScreenName` sobre a instância do bloco em questão. Este último método está definido para cada tipo de bloco e devolve uma *string* diferente para cada um deles. O método da classe `ScratchFrameMorph` faz uma pesquisa na directoria do Scratch, dentro da pasta onde se encontram todas as imagens de ajuda dos blocos, `Help/en` (para o idioma Inglês, ou noutra subpasta de `Help` que exista para o idioma que se está a utilizar). Nessa pesquisa procura por um ficheiro de extensão HLP, GIF, PNG, JPG ou BMP, cujo nome é dado pela *string* recebida como argumento, apresentando a imagem correspondente ao ficheiro, quando encontrada, numa janela de diálogo. Portanto, a solução reside em criar as 2 novas imagens (uma para o bloco **import** e outra, genérica, para os blocos definidos pelo utilizador, visto que estes apresentam comportamentos diferentes que são especificados pelo utilizador em

tempo de execução) com um nome específico, extensão válida, e colocá-las na pasta `Help/en`. Depois, é necessário redefinir, dentro do protocolo `accessing`, o método `helpScreenName` para as classes que representam o bloco **import** e os blocos definidos pelo utilizador, `ImportCBlockMorph` e `UserCommandBlockMorph`, respectivamente.

Tendo por base imagens já existentes para outros blocos, utiliza-se um programa de edição de imagem para criar as ditas imagens, dando-se o nome de `importBlock.gif` à imagem associada ao bloco **import**, e `scriptBlock.gif` à imagem associada aos blocos definidos pelo utilizador. Estas imagens são então colocadas na pasta `Help/en`. De seguida, passa-se à redefinição dos métodos. Na classe `ImportCBlockMorph` define-se que a *string* devolvida é "importBlock" (para ser concordante com o nome do ficheiro da imagem) (Código C.161). Na classe `UserCommandBlockMorph`, o método devolve a *string* "scriptBlock" (Código C.162). Imediatamente as imagens passam a ser reconhecidas e apresentadas para cada bloco (Figuras 5.44 e 5.45).

Figura 5.44: Ecrã de ajuda do bloco **import**.

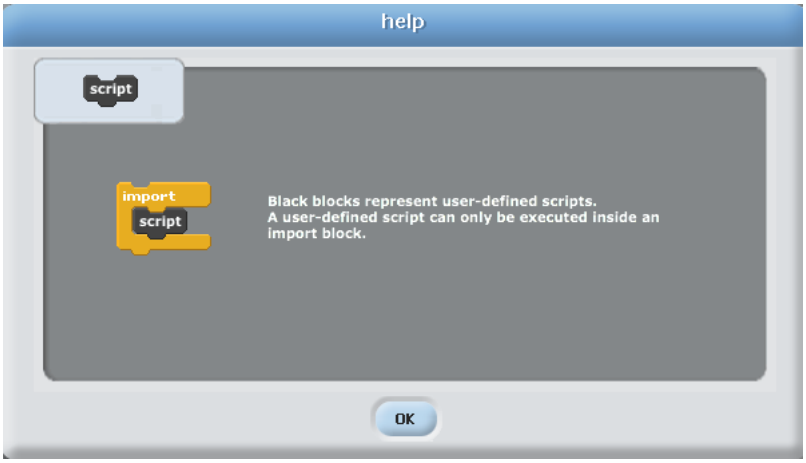


Figura 5.45: Ecrã de ajuda genérico dos SDUs.

O bloco associado à bandeira azul também não possui um ecrã de ajuda personalizado.

Quando sobre este é executada a opção **help**, é mostrada a imagem do bloco associado à bandeira verde. Portanto, efectua-se um procedimento semelhante. Cria-se o ficheiro `startScriptHat.gif`, coloca-se na directoria correcta, e altera-se o método `helpScreenName` da classe `EventHatMorph`, acrescentando-lhe mais um caso de teste do evento que está associado ao bloco (Código C.163). O novo teste verifica se o evento é o `Scratch-StartScriptClicked`, e se for carrega a nova imagem (Figura 5.46).

Figura 5.46: Ecrã de ajuda do bloco associado à bandeira azul.

Aproveita-se também esta tarefa para replicar a estrutura da pasta `Media` do Scratch, com as suas subpastas `Backgrounds`, `Costumes` e `Sounds`, copiando-se para estas os conteúdos existentes nas suas homólogas distribuídas com o Scratch original. Apenas se apagam as duas imagens relativas ao gato do Scratch da subpasta `Costumes/Animals`, por motivos relacionados com a licença do código fonte do Scratch. Ainda relativamente à estrutura de pastas, é preciso adicionar uma pasta de nome `Projects` à estrutura existente nesta modificação do Scratch. É nesta pasta que são procurados ficheiros quando se escolhe a opção **Examples** nas janelas de diálogo de escolha de ficheiros (Figura 5.47).

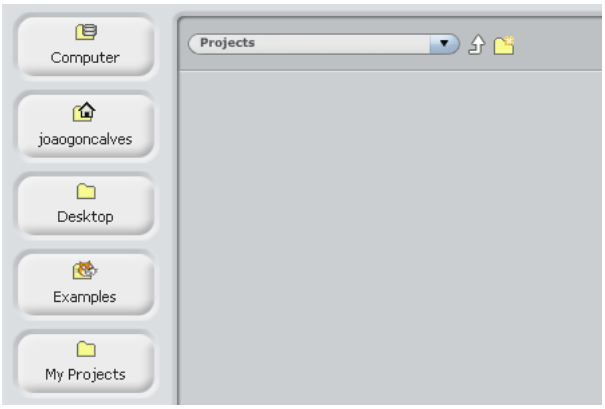


Figura 5.47: Janela de diálogo de escolha de ficheiros.

5.4.16 Exportação de um SDU

Nesta secção descreve-se todo o trabalho efectuado para que um dado SDU num dado projecto possa ser exportado para ficheiro, podendo ser importado num outro projecto por forma a ser usado.

O processo de exportação de um SDU consiste na criação de um ficheiro de extensão `.script` que armazena a sua definição. A definição é uma instância da classe `UserBlockDefinition` que é serializada para um ficheiro, em formato binário. Esta nova funcionalidade passa a estar disponível através do menu contextual do SDU, sendo visível apenas quando este se encontra na palete. Como tal, o método `rightButtonMenu` da classe `UserCommandBlockMorph` volta a ser alterado (Código C.164), ficando de acordo com a Figura 5.48.

Figura 5.48: Menu contextual de um SDU, com a opção de exportação.

À nova opção **export script** associa-se o método `exportScript` da mesma classe. Este método está dividido em 4 passos:

1. Escolha do nome do ficheiro e directoria para o gravar;
2. Caso já exista um ficheiro com o mesmo nome na directoria, escolhe-se se se quer gravar por cima ou dar novo nome/mudar de directoria;
3. Construção do nome do ficheiro;
4. Escrita da definição em ficheiro.

No passo 1 invoca-se o método de classe `ScratchFileChooserDialog»saveScriptFile:`, que recebe como argumento o nome do SDU e abre uma janela de exploração do sistema de ficheiros. Esta janela tem o título **Export script** e apresenta como nome por omissão para o ficheiro o nome do SDU recebido como argumento.

Antes da apresentação do código que constrói essa janela é necessário atentar num pormenor existente nas janelas de exploração do sistema de ficheiros. Todas elas apresentam do lado esquerdo uma coluna com botões, que são atalhos que permitem aceder a determinadas pastas do computador do utilizador (Figura 5.49).

A pasta da raiz do disco, a pasta da *home* do utilizador e a do ambiente de trabalho são as que aparecem por omissão, sendo as restantes dependentes da funcionalidade para a qual a janela de diálogo é solicitada (na Figura 5.49, como se trata de exportação de trajes, aparece uma outra pasta, de nome `Costumes`). Para o caso da exportação de SDUs, e fazendo uma analogia com a exportação de um projecto Scratch, é necessário ter 2 atalhos para pastas extra

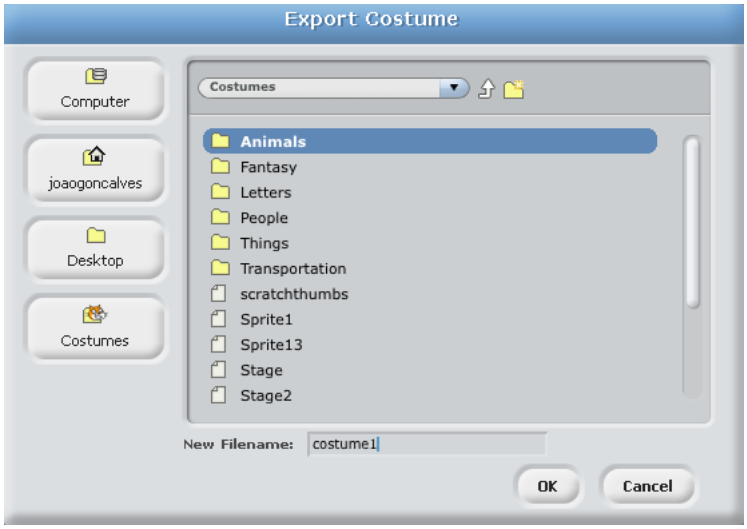


Figura 5.49: Janela de diálogo de exploração do sistema de ficheiros.

para além das 3 referidas: um, de nome `Scripts`, que aponte para a pasta de igual nome presente dentro da directoria de instalação desta extensão do Scratch, e o segundo, de nome `My Scripts`, que aponte para a pasta de igual nome presente dentro da directoria dos documentos do utilizador. Para conseguir fazer isto, é necessário indicar que o tipo da janela de diálogo é dado pelo símbolo `#scripts` (novo tipo definido para esta tarefa) (Código C.165). Olhando para o método modificador `type:` da classe `ScratchFileChooserDialog`, percebe-se que a atribuição de um tipo a uma janela de diálogo faz acrescentar os botões de atalho para esse tipo (Código C.166). Antes de se abordar os botões adicionados, note-se que este método, na última instrução, invoca o método de classe `getLastFolderForType:` que devolve a última pasta utilizada para o tipo dado por parâmetro, ou a pasta por omissão para esse tipo. Para o caso em que devolve a pasta por omissão para o tipo, é invocado o método de classe `getDefaultFolderForType:`, e neste é necessário indicar qual é essa pasta para o tipo `#scripts` (Código C.167). A pasta é então dada pela invocação do método de classe `userScriptsDir`, que é em tudo semelhante ao `userScratchProjectsDir`, excepto no nome da pasta que é `My Scripts`. O método devolve o caminho para a pasta `My Scripts` e, caso não exista uma pasta com esse nome dentro da pasta `Documents` do utilizador, ele tenta criá-la (Código C.168).

Passando então à colocação dos botões de atalho na janela de diálogo pelo método de instância `addShortcutButtons`, este começa por adicionar os 3 atalhos que existem em todas as janelas de diálogo. Depois, em função do tipo da janela, adiciona atalhos extra. Para o tipo `#scripts`, adiciona-se então os 2 atalhos extra para as pastas `Scripts` e `My Scripts` (Código C.169). O método `sampleScripts` permite aceder à pasta de nome `Scripts` presente na instalação desta extensão do Scratch (Código C.170). O método `userScripts` lista o conteúdo da directoria dada pela invocação do método `userScriptsDir`, já atrás referido, e que devolve a pasta `My Scripts` (Código C.171).

Estando então preparado o ambiente para criar uma janela para exportação de SDUs, volta-se ao método `saveScriptFile:` (Código C.172), invocado no `exportScript`, responsável por criar a dita janela (Figura 5.50).

Este método devolve o nome do ficheiro para o método `exportScript`. Feitas as validações desta operação, entra-se na fase 2 deste método.

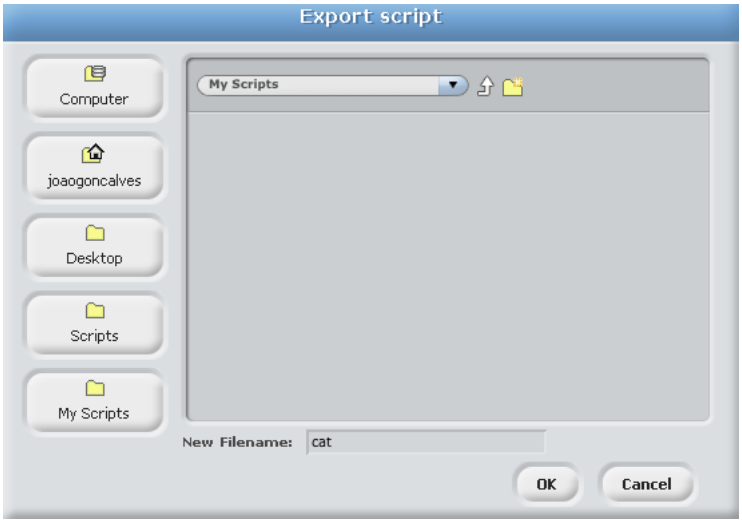


Figura 5.50: Janela de diálogo de exportação de um SDU.

Caso o nome escolhido para o ficheiro coincida com algum já existente na directoria em causa, é perguntado ao utilizador se pretende gravar por cima. Caso o utilizador não queira, então é novamente aberta uma janela de exportação de SDUs, mas desta vez o nome que ela apresenta para o ficheiro não é o nome do SDU mas sim "script1", sendo então realizado o mesmo procedimento atrás explicado, através da invocação do método `saveScriptFile:`. Apenas há que salientar o método que confirma a reescrita do ficheiro em caso de já existir, `confirmScriptFileOverwriteIfExisting:` (Código C.173).

Na fase seguinte define-se o formato final do nome do ficheiro, através da invocação do método `nameFromFileName:` (Código C.174).

Finalmente, o método `exportScript` termina com a invocação do método `writeScriptFile:`, que vai criar o ficheiro com o nome escolhido (Código C.175).

O método que trata de todo o processo de escrita em ficheiro, `writeScriptFile:`, segue o mesmo esquema de criação de ficheiro temporário e tratamento de erros do método que faz a escrita em ficheiro de um projecto Scratch. O esquema normal de escrita em ficheiro da definição do SDU recorreria à abertura de uma *stream* de objectos, `ObjStream`, que serializaria a definição, mantendo a estrutura com que esta foi guardada aquando da criação do SDU. No entanto, verifica-se que este procedimento faz com que a definição, ao ser recuperada numa importação posterior, apresente problemas, nomeadamente no que toca aos argumentos dos blocos (deixam de ser editáveis). O problema reside no campo `body` da definição, que guarda o empilhamento de blocos que constitui o *script*. Para dar a volta a este problema converte-se este empilhamento numa sequência de tuplos e serializa-se a definição com esta alteração (Código C.176). Dada esta alteração, é preciso ter em conta que, ao importar o SDU, é necessário reconstruir o empilhamento, afectando-o ao **Palco** ou *Sprite* para o qual o SDU é importado. As restrições de importação de SDUs (apresentadas na Secção 5.4.17) obrigam a recorrer a um artifício. Esse artifício é um campo extra na classe `UserBlockDefinition`, de nome `receiver`, que contém o nome da classe do objecto `ScriptableScratchMorph` para o qual o SDU foi definido. Este campo tem apenas dois valores possíveis: "`ScratchSpriteMorph`", para o caso dos *Sprites*, e "`ScratchStageMorph`" para o caso do **Palco**. Assim, é necessário alterar os métodos `initialize` (Código C.177), `storeFieldsOn:` (Código C.178) e `initFieldsFrom:version:` (Código C.179)

da classe que representa a definição, bem como adicionar o método selector e modificador desta nova variável de instância. Esta variável é então ser alterada nos métodos `addBlockWithBody:` e `saveBlockWithBody:` da classe `ScriptableScratchMorph`, já analisados anteriormente no processo de criação de um SDU e de criação de um novo SDU após consulta da definição de um outro, respectivamente. Em ambos, a única modificação a fazer é na instrução que cria a instância de `UserBlockDefinition` que guarda a definição, que agora passa a guardar um valor extra correspondente ao campo `receiver` (Código C.180). Com a solução adoptada para este problema perde-se um pouco a legibilidade da modelação da definição de um SDU, no entanto esta apresenta-se como a única solução encontrada dentro do tempo disponível.

Posteriormente, no método `writeScriptFile:`, abre-se então a *stream* de objectos para nela escrever a instância da definição (Código C.181). Antes de terminar este processo, há ainda que registar no método `userClasses` mais classes a serem serializadas, por força do funcionamento do mecanismo de serialização (Código C.182). Concluído todo este processo é então gerado um ficheiro com o nome escolhido, de extensão `.script`, que armazena a definição do SDU.

5.4.17 Importação de um SDU

O processo de importação de um SDU consiste em, dada a definição do SDU previamente armazenado num ficheiro de extensão `.script`, recuperar a definição e armazená-la no **Palco** ou *Sprite(s)* para o qual é importado.

Para atingir este objectivo, começa-se por definir no método `ScratchFrameMorph»fileMenu:` uma opção de nome **Import Script** no menu **File**, junto das opções de importação de um projecto e exportação de um *Sprite* (Código C.183), obtendo-se o resultado da Figura 5.51.

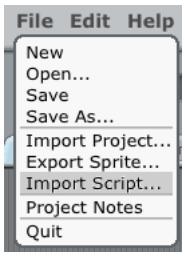


Figura 5.51: Menu **File** com a opção **Import Script**.

A esta nova opção associa-se o método `importScript`, definido na mesma classe, no protocolo `menu/button actions`. Este invoca o método homólogo da classe `ScriptableScratchMorph`, sendo o mesmo executado sobre o *Sprite* ou o **Palco** seleccionado no momento (Código C.184). Já na classe `ScriptableScratchMorph`, o método `importScript` está dividido em 5 fases:

1. Escolha do ficheiro do SDU;
2. Leitura do conteúdo do ficheiro;
3. Validação do SDU a importar;
4. Criação do SDU;
5. Actualização da categoria **MyScripts**.

Na fase 1 cria-se uma janela de diálogo para o tipo de ficheiros dado pelo símbolo `#scripts`, cuja extensão deve ser `.script`, tendo essa janela o título **Import Script** (Figura 5.52).

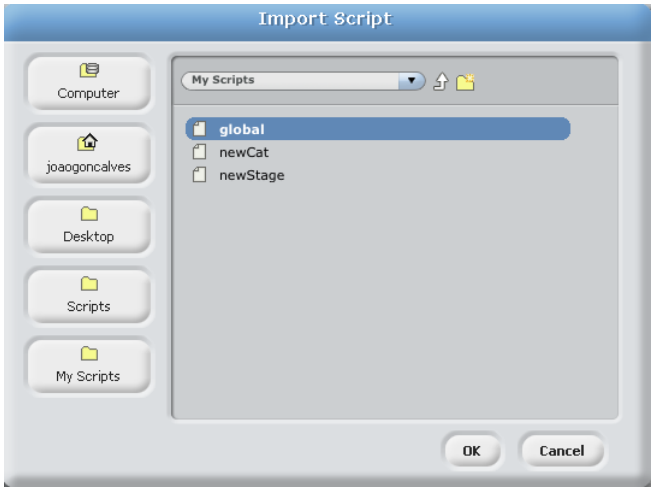


Figura 5.52: Janela de diálogo de escolha de ficheiros de SDUs.

A fase 2 consiste na leitura do conteúdo do ficheiro seleccionado. Esta leitura é feita no método `ScriptableScratchMorph»readScriptFile:` (Código C.185), que recebe como parâmetro o nome do ficheiro a ler. Começa-se por abrir o ficheiro para efectuar uma leitura em modo binário, lê-se todo o conteúdo do ficheiro e, de seguida, cria-se uma `ObjStream` sobre esse conteúdo, que vai construir o objecto nele armazenado. O despiste de erros na leitura do ficheiro segue o mesmo padrão existente para o método que faz a leitura de um projecto Scratch.

Seguidamente, na fase 3, é necessário verificar se a importação do SDU pode ser realizada para o contexto no qual se quer inseri-lo. Portanto, dada a definição do SDU lida do ficheiro é preciso efectuar 4 testes:

1. Se se está a importar um SDU global para o **Palco**;
2. Se se está a importar um SDU local a um *Sprite* para o **Palco**;
3. Se se está a importar um SDU local ao **Palco** para um *Sprite*;
4. Se o SDU a importar já existe.

O primeiro teste é necessário porque os SDUs definidos para o **Palco** são locais, nunca globais. Os SDUs globais apenas podem ser definidos para os *Sprites*. O segundo e terceiro testes são necessários devido ao facto de existirem blocos que apenas existem para o **Palco** ou para os *Sprites*, e a importação de um SDU que contivesse um bloco não existente num dado contexto levaria à ocorrência de erros. Por exemplo, o **Palco** não possui blocos na categoria **Motion**, logo, importar para o **Palco** um SDU que contenha blocos dessa categoria provoca erros. É nestes dois testes que é importante a variável `receiver` adicionada à classe `UserBlockDefinition`, pois indica se o SDU a importar é proveniente de um *Sprite* ou do **Palco**. O quarto teste evita a situação de existirem SDUs repetidos e é utilizado também no processo de criação de um SDU, já anteriormente abordado.

A fase seguinte consiste na criação do SDU, dada a definição lida do ficheiro e já validada. Este processo é efectuado no método `ScriptableScratchMorph»createScriptWithDefinition:` e consiste em adicionar a definição do SDU ao dicionário `userBlocks` do *Sprite*/**Palco** para o qual o SDU é importado (Código C.186). No entanto, como o corpo do *script* presente nessa definição

está representado sob a forma de uma sequência de tuplos, é necessário criar o empilhamento que lhe corresponde. Depois disso, o método `createUserBlockWithDefinition:global:` trata de afectar o corpo da definição ao objecto para o qual a definição é importada. Tal como no processo de criação de um SDU, este procedimento é realizado quer para SDUs locais quer para globais.

A última fase consiste na actualização da categoria de blocos **MyScripts**, que irá apresentar o novo SDU importado sob a forma de um bloco. Portanto, o método `importScript` fica com a configuração presente no Código C.187.

Apenas há que fazer uma ressalva relativamente à importação de SDUs que contenham no seu corpo blocos que lidam com variáveis ou listas. Quando o *script* a importar para um *Sprite* possui blocos relativos a variáveis, fossem elas locais ou globais aquando da criação do *script*, o processo de importação adiciona sempre as variáveis ao *Sprite* como se fossem globais. No que toca a listas há 2 casos. Caso o *script* que contenha blocos relativos à lista seja global, ao importar o *script* para um *Sprite* é criada uma lista local em todos os *Sprites*. Se o *script* que contiver blocos relativos a uma lista for local, é criada uma lista local no *Sprite* onde é importado. A importação para o **Palco** resulta sempre na criação de variáveis ou listas globais, seguindo o comportamento normal para este caso. Apesar de estes pormenores introduzirem pequenas diferenças entre o SDU exportado e o resultante da importação, dado o tempo disponível para efectuar alterações que as resolvessem e a profundidade que essas alterações teriam, optou-se por manter este esquema de funcionamento. Com isto se conclui a importação de um SDU.

5.5 Conclusão

Neste capítulo foi descrito todo o processo de análise de baixo nível do ambiente Scratch, bem como o desenvolvimento das novas funcionalidades adicionadas a este ambiente. A análise da estrutura interna focou-se na documentação do conjunto de classes que suportam o ambiente, com maior foco nas classes que representam os blocos, os objectos programáveis e os elementos da interface. O processo de análise prosseguiu ao longo de todo o desenvolvimento das extensões, permitindo fundamentar todas as decisões tomadas e explicar, em detalhe, a sequência de passos levada a cabo na implementação das funcionalidades. As funcionalidades criadas vêm suportar o novo conceito de "*scripts* definidos pelo utilizador". Estes são encapsulamentos de comportamentos na forma de novos blocos, que são listados numa nova categoria. Uma nova aba permite a sua consulta, teste, modificação e criação de novos blocos a partir destes. Em conjunto com o novo bloco **import**, podem ser utilizados no contexto dos programas que o utilizador desenvolva. A qualquer momento, o utilizador pode eliminar estes blocos. Foi adicionada a possibilidade de gravar e ler projectos que utilizem estas novas entidades assim como a possibilidade de as exportar para um formato persistente, podendo ser importadas noutros projectos. As questões de usabilidade também foram alvo de atenção, por forma a tentar manter a simplicidade e facilidade de utilização do ambiente. Ainda foram tidas em conta outras funcionalidades não tão visíveis à primeira vista, mas que no seu todo ajudam a manter a coerência da informação gerada ao longo da utilização desta modificação do Scratch. O ambiente vê assim adicionada uma nova camada de funcionalidade que atinge quase todo o espectro de funcionalidades preexistentes. Ficou por implementar a exportação de *Sprites* e importação de projectos com SDUs, duplicar correctamente os SDUs quando se duplica um *Sprite* e copiar correctamente um SDU local de um *Sprite* para outro (largando o bloco sobre o ícone do *Sprite*). Para além dos pontos anteriores, aos quais se adicionam as questões relacionadas com a utilização de variáveis e listas na importação de SDUs, todo o pacote está funcional.

Capítulo 6

Exemplo de aplicação

Neste capítulo faz-se uma apresentação das potencialidades das novas extensões desenvolvidas, através da demonstração da sua aplicação num caso prático.

6.1 Rosáceas e relógio

Aproveitando o espírito de partilha do Scratch e da sua comunidade, utiliza-se como caso prático o projecto "Rosáceas" partilhado no portal Sapo Kids [Sap12]. Este projecto tem por objectivo permitir o desenho de rosáceas, estampando o traje de um *Sprite* seleccionado em torno de um centro de rotação, com um espaçamento variável. Através da variação do valor da amplitude do ângulo ou do número de repetições, serão desenhadas diferentes rosáceas (Figura 6.1).

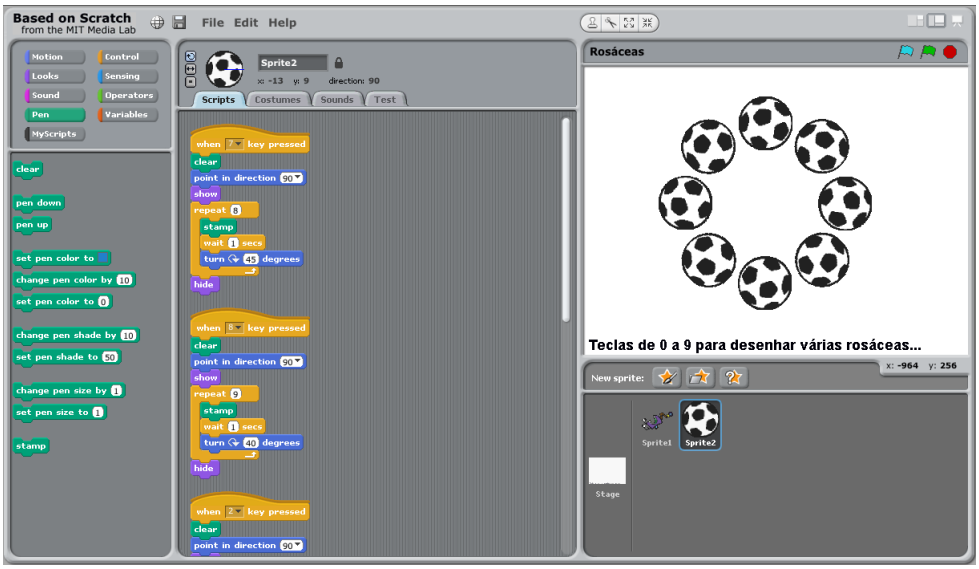


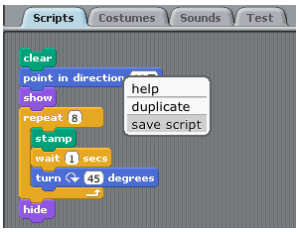
Figura 6.1: Projecto "Rosáceas".

Para desenhar as diferentes rosáceas existem diferentes *scripts* que possuem a mesma estrutura, variando apenas os dois valores referidos e o bloco do tipo *hat* que os activa (Figura 6.2).

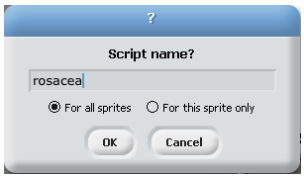
Figura 6.2: *Script* que desenha uma rosácea, com valores do ângulo e das repetições assinalados.

Para efeitos de demonstração, vamos admitir que existe apenas um *Sprite* que possui um único *script* que desenha uma rosácea, utilizando um valor fixo para a amplitude do ângulo e para o número de repetições. O **Palco** mantém-se tal como está, pois a sua utilidade neste projecto resume-se ao seu pano de fundo que indica ao utilizador como interagir com o projecto.

Uma vez que se pretende chegar a um estado em que existem múltiplos *Sprites* capazes de desenhar múltiplas rosáceas, e dado que todos os *scripts* necessários têm a mesma estrutura, faz sentido que se encapsule o comportamento definido por essa estrutura num SDU. Para tal, invoca-se sobre um dos blocos constituintes do empilhamento a opção **save script** (Figura 6.3a). De seguida, dá-se o nome ao SDU. Como este encapsula o desenho de uma rosácea, toma o nome **rosacea**. Para além disso, fica definido como sendo global para que a definição de uma rosácea fique acessível a todos os *Sprites* (Figura 6.3b). Confirmando estas escolhas, o SDU surge na categoria **MyScripts**, posicionado na zona dedicada aos SDUs globais (Figura 6.3c).



(a) Menu contextual dum bloco, com a opção **save script**.



(b) Janela de diálogo de criação do novo bloco.

(c) Novo bloco **rosacea** na categoria **MyScripts**.

Figura 6.3: Processo de criação de um SDU que desenha uma rosácea.

O utilizador pode confirmar que o corpo do SDU foi bem guardado, consultando-o, usando para isso o menu contextual do novo bloco (Figura 6.4).

Imediatamente, o painel central altera a aba para a **Test**, encaixando o corpo do SDU no bloco associado à bandeira azul, o qual vê acrescentado ao seu corpo o texto que informa qual o SDU que está a ser consultado (Figura 6.5).

Figura 6.4: Menu contextual do SDU, com a opção **show definition**.

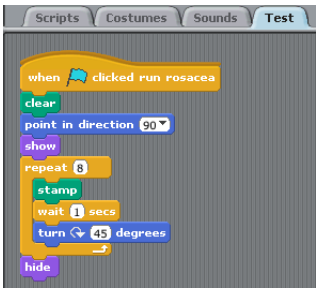


Figura 6.5: Aba **Test** mostrando o corpo do bloco que desenha uma rosácea.

Este novo bloco criado, que desenha uma rosácea com um dado valor de amplitude do ângulo e número de repetições, pode ser usado como modelo para construir novos blocos idênticos, mas que possuam valores diferentes destes dois parâmetros. Para tal, o utilizador pode alterar estes valores (Figura 6.6) e testar o seu efeito através da utilização da bandeira azul (Figura 6.7). Durante a execução do teste, o *script* e a bandeira azul ficam destacados. De notar que os valores alterados estão sujeitos à restrição de que o seu produto deve ser igual a 360, para que se obtenha o efeito rosácea.

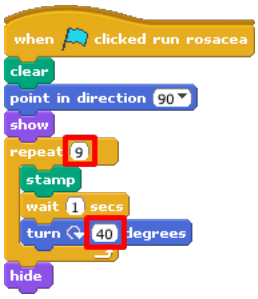
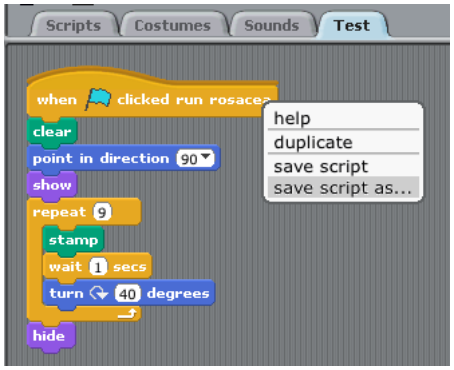


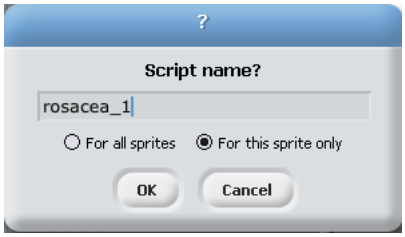
Figura 6.6: *Script* que desenha uma rosácea com novos valores.

Posto isto, o utilizador pode gravar este *script* modificado num novo bloco, desta vez local ao *Sprite*. Para isso recorre ao menu contextual de qualquer um dos blocos do empilhamento, activando a opção **save script as** (Figura 6.8a). Ao novo bloco local atribui-se o nome **rosacea_1**, porque será associado ao bloco do tipo *hat* activado pela tecla 1 (Figura 6.8b). Após a criação, o novo bloco é mostrado na categoria **MyScripts**, na zona dedicada aos SDUs que sejam locais (Figura 6.8c). Ao mesmo tempo, o bloco associado à bandeira azul actualiza o texto do seu

Figura 6.7: Teste do *script* que desenha uma rosácea com novos valores.



(a) Opção **save script as** do menu contextual do bloco asssociado à bandeira azul.



(b) Janela de diálogo de criação do novo bloco local.

(c) Novo bloco **rosacea_1** na categoria **MyScripts**.

(d) Bloco associado à bandeira azul actualizado.

Figura 6.8: Processo de criação de um SDU a partir de outro.

corpo para indicar ao utilizador que o empilhamento que se encontra encaixado por baixo deste corresponde ao corpo do bloco **rosacea_1** (Figura 6.8d).

De seguida o utilizador pode tentar utilizar o novo bloco criado. Se não conhecer o modo de funcionamento deste novo tipo de blocos, o utilizador é tentado a colocar o novo bloco na área de *scripting* da aba **Scripts** e executá-lo da mesma forma que faz com os restantes blocos. No entanto, como estes blocos têm um modo de funcionamento diferente, é assinalado o erro no novo bloco bem como é apresentada ao utilizador uma mensagem informativa (Figura 6.9). Em caso de dúvida, o utilizador pode accionar a opção **help** do menu contextual do novo bloco por forma a obter mais ajuda acerca do seu funcionamento (Figura 6.10).

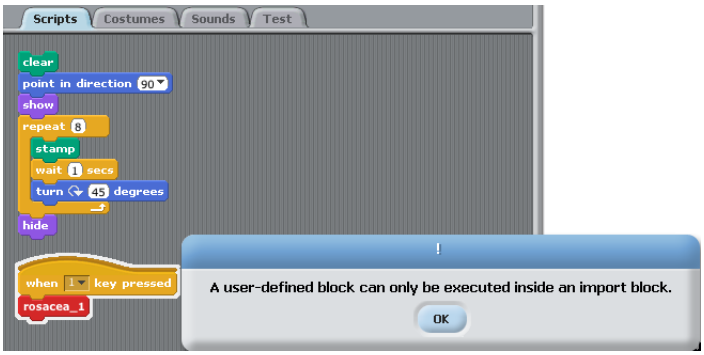


Figura 6.9: Sinalização de erro e mensagem informativa.

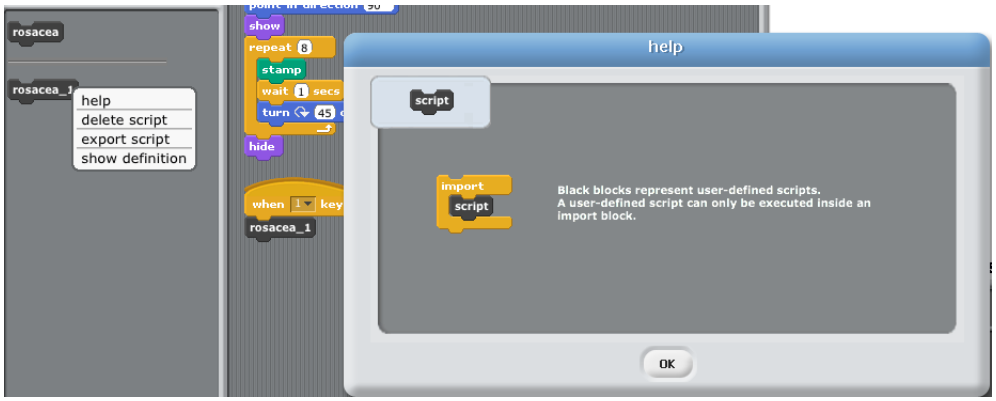


Figura 6.10: Ecrã de ajuda do SDU.

Uma vez apresentado o modo de utilização destes blocos, o utilizador pode então usar o bloco **import** para dar vida ao bloco **rosacea_1** (Figura 6.11).

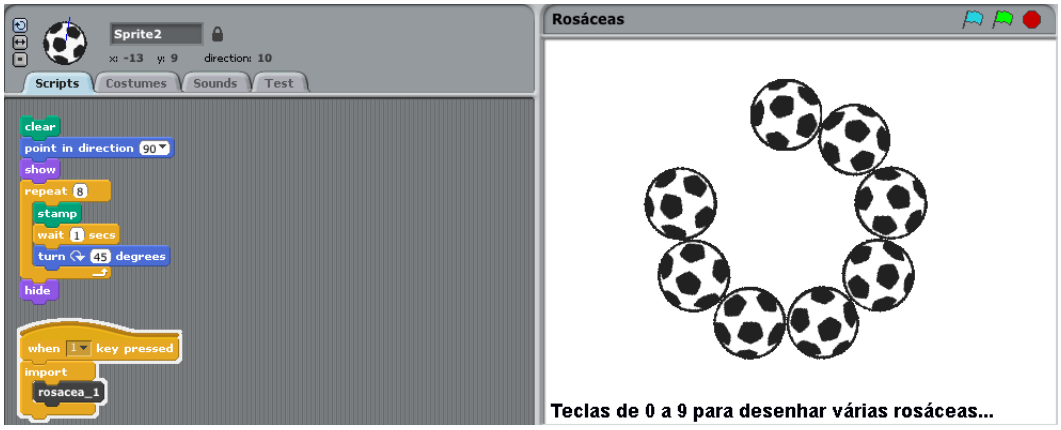


Figura 6.11: Utilização do bloco **import** para executar o SDU.

A partir deste momento, o utilizador pode recorrer a esta sequência de passos para criar outros blocos que utilizem outras combinações dos valores da amplitude do ângulo e número de repetições (Figura 6.12).

Figura 6.12: Vários blocos que desenham diferentes rosáceas.

Pretende-se agora adicionar ao projecto um novo *Sprite*, com um traje diferente, que permita desenhar rosáceas com motivos diferentes. Como o SDU que encapsula o desenho de uma rosácea está definido como sendo global, este fica automaticamente disponível para o novo *Sprite* (Figura 6.13).

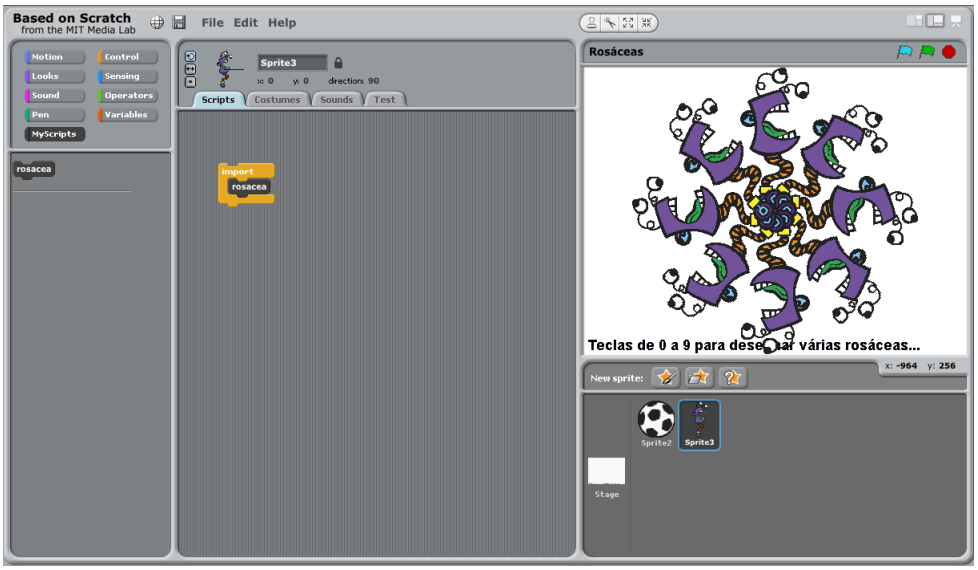


Figura 6.13: Bloco global **rosacea** disponível para o novo *Sprite*.

Também aqui se podem experimentar outras combinações dos valores que definem as rosáceas, tal como no primeiro *Sprite* (Figura 6.14).

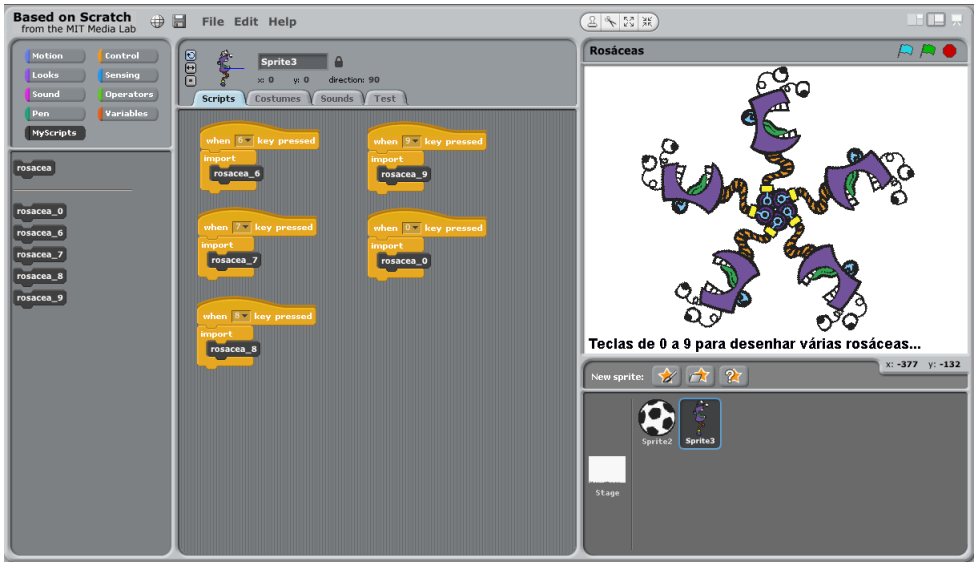
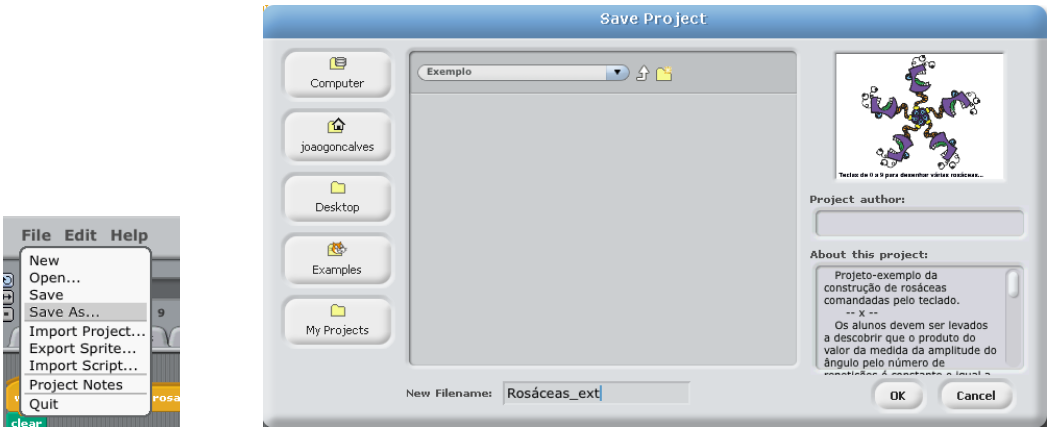


Figura 6.14: Vários blocos que desenham diferentes rosáceas com o novo *Sprite*.

Com isto, atinge-se o estado final do projecto, que contém múltiplos *Sprites* capazes de desenhar múltiplas rosáceas. Assim, o utilizador pode agora gravar o projecto em ficheiro. Para comprovar que a informação relativa aos SDUs e à aba **Test** é guardada e recuperada correctamente, consulta-se a definição de qualquer um dos SDUs em ambos os *Sprites*, previamente

à opção de escrita do projecto em ficheiro. De seguida, através da opção **Save As** do menu **File**, o utilizador pode gravar o projecto desenvolvido (Figura 6.15a). Selecciona-se uma pasta e define-se um nome para o ficheiro, e.g., `Rosáceas_ext` (Figura 6.15b). É assim gerado um ficheiro de nome `Rosáceas_ext.extsb` na pasta escolhida (Figura 6.16).



(a) Opção **Save As** do menu **File**.

(b) Janela de diálogo de gravação do projecto.

Figura 6.15: Processo de gravação do estado do projecto em ficheiro.

Figura 6.16: Ícone do ficheiro do projecto no sistema de ficheiros.

De imediato, a interface é actualizada com o novo nome dado ao projecto (apresentado por cima do **Palco**). Quer os SDUs quer o estado da aba **Test** são mantidos correctamente (Figura 6.17). Este seria o resultado obtido caso o utilizador abrisse explicitamente o projecto via opção **Open** do menu **File**, pelo que não é aqui apresentado esse procedimento.

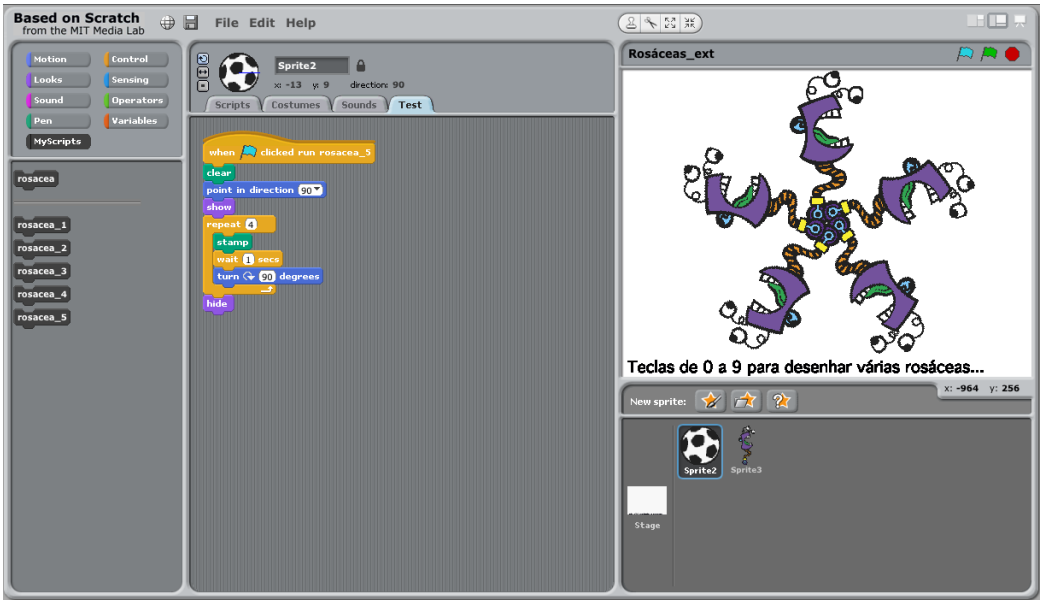


Figura 6.17: Projecto "Rosáceas_ext".

Dado que a única diferença entre todos os *scripts* criados é o valor da amplitude do ângulo e do número de repetições, pode-se aproveitar esse facto para construir um único *script* em que esses dois valores estão encapsulados em duas variáveis: `Ângulo` e `Repetições` (Figura 6.18).

Figura 6.18: Variáveis para guardar o valor do ângulo e número de repetições.

Com isto, o utilizador pode criar um novo *script* que utilize as variáveis no lugar dos referidos valores. Cria-se assim o SDU local **rosacea_vars**, cujo corpo é apresentado na Figura 6.19.

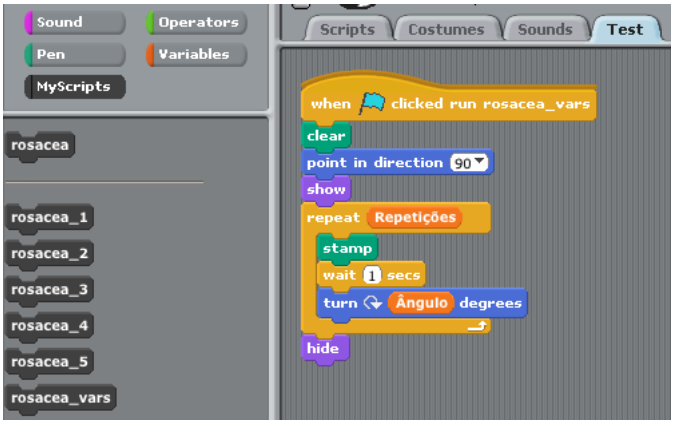
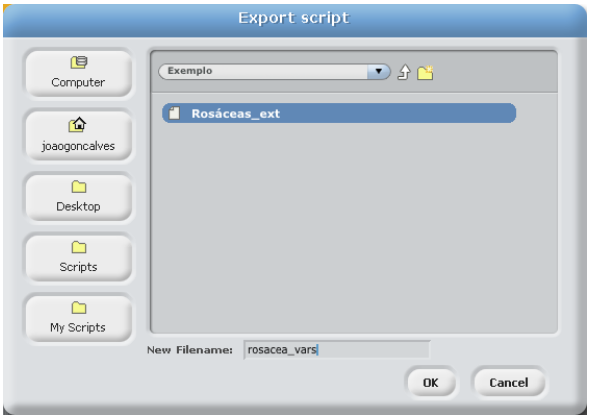


Figura 6.19: Novo SDU **rosacea_vars** e respectiva definição.

Este novo SDU genérico pode ser exportado para ficheiro, podendo ser usado como base de trabalho noutro projecto, via mecanismo de importação de SDUs. A exportação é activada através do menu contextual do SDU quando este se encontra na palete (Figura 6.20a). Seguidamente, o utilizador escolhe a directoria e nome do ficheiro. Neste caso, aceita-se o mesmo nome do SDU (Figura 6.20b). Finalmente, é criado o ficheiro de extensão `.script` (Figura 6.20c).

(a) Opção **export script** do menu contextual do SDU.



(b) Janela de diálogo de exportação do SDU.

(c) Ícone do ficheiro do SDU no sistema de ficheiros.

Figura 6.20: Processo de exportação de um SDU para ficheiro.

Dada a funcionalidade encapsulada pelo SDU, podem-se imaginar outros contextos em que a mesma pode ser útil, nomeadamente na simulação de um ponteiro de um relógio. Para comprovar esta situação, cria-se um novo projecto constituído pelo **Palco**, que tem como pano de fundo um mostrador de um relógio, e por um *Sprite* que tem a forma de um ponteiro que, para este exemplo, toma a função do ponteiro dos segundos (Figura 6.21).

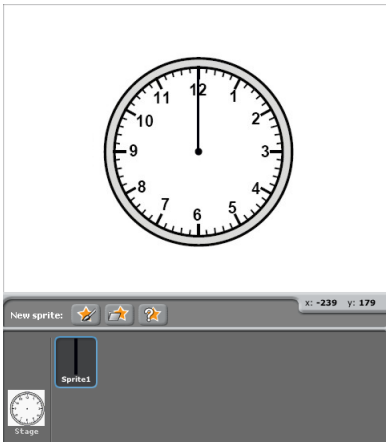
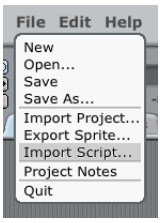


Figura 6.21: Mostrador de relógio com ponteiro dos segundos.

A rotação do ponteiro dos segundos pode ser simulada através do mesmo SDU que desenha as rosáceas e que foi previamente exportado. Como tal, o primeiro passo a tomar consiste na importação desse SDU via opção **Import Script** do menu **File** (Figura 6.22a e 6.22b).



(a) Opção **Import Script** do menu **File**.

(b) Janela de diálogo de importação do SDU.

Figura 6.22: Processo de importação de um SDU a partir de um ficheiro.

Assim que o utilizador executa a acção de importação, o SDU **rosacea_vars** surge na categoria **MyScripts**, na zona dedicada aos SDUs locais (isto porque este SDU havia sido definido como local ao *Sprite* do projecto das rosáceas) (Figura 6.23a). Como o SDU importado contém na sua definição blocos relativos às duas variáveis existentes, estas são adicionadas ao projecto (Figura 6.23b).

Para simular o ponteiro dos segundos, o *script* tem de ser ligeiramente modificado para que cada estampagem do traje do *Sprite* que representa o ponteiro seja apagada no fim de cada

(a) Bloco representativo do SDU importado.

(b) Variáveis adicionadas aquando da importação do SDU.

Figura 6.23: Novos blocos adicionados ao projecto após a importação do SDU.

movimento. Remove-se então o bloco **hide** da última instrução do *script* e adiciona-se o bloco **clear** como última instrução do ciclo (Figura 6.24).

Figura 6.24: *Script* alterado.

Seguidamente, o utilizador pode, via opção **save script** do menu contextual de qualquer um dos blocos do *script*, actualizar a definição do bloco, cujo nome é **rosacea_vars**. Mas para ter um bloco de nome mais sugestivo, opta-se pela opção **save script as**, criando-se um novo bloco local de nome **rotacao_ponteiro**. Para se obter o efeito de um ponteiro dos segundos é necessário definir valores correctos para a amplitude do ângulo de rotação e para o número de repetições. Definidos estes valores, o último passo consiste na utilização do bloco **import** para executar o novo SDU (Figura 6.25).

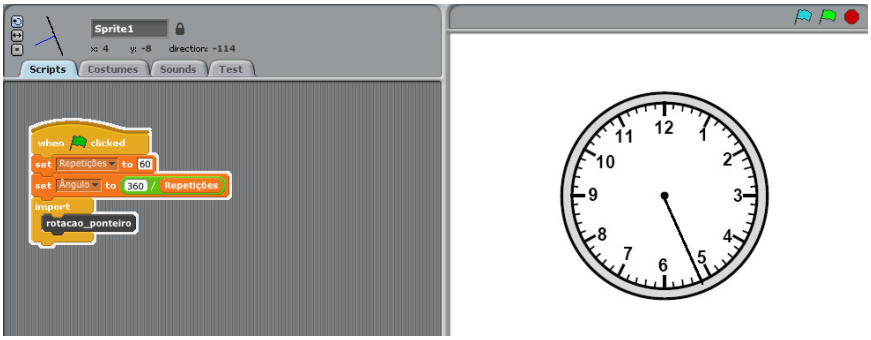
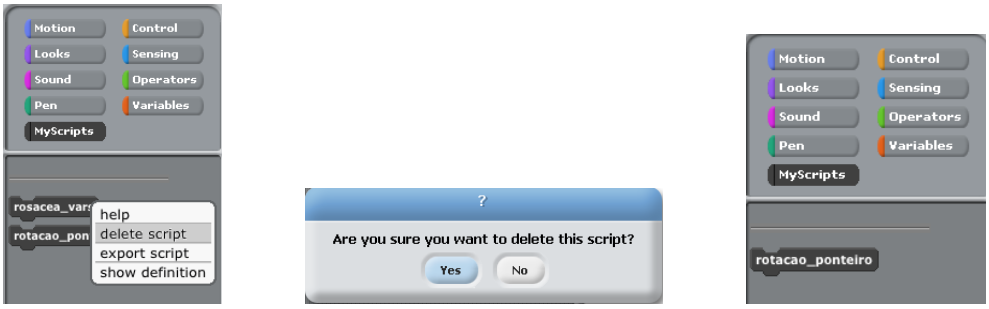


Figura 6.25: Simulação do ponteiro dos segundos num relógio, usando o SDU **rotacao_ponteiro**.

Uma vez que possui um novo bloco adaptado ao problema do relógio, o utilizador pode eliminar o bloco **rosacea_vars** inicialmente importado, usando a opção **delete script** do seu menu contextual (Figura 6.26).



(a) Opção **delete script** do menu contextual do SDU.

(b) Janela de diálogo de eliminação do SDU.

(c) Categoria **MyScripts** actualizada.

Figura 6.26: Processo de eliminação de um SDU.

6.2 Conclusão

Esta demonstração passo a passo permite que o utilizador perceba de que forma pode tirar partido das novas funcionalidades adicionadas ao ambiente Scratch. Foi utilizado como base um projecto partilhado no portal Sapo Kids relacionado com o desenho de rosáceas, o qual foi sendo modificado tirando partido das novas capacidades do sistema (e.g., a nova aba e a capacidade de criar novos blocos). Como forma de salientar a importância da componente de exportação e importação de SDUs, enquanto mecanismo de partilha de comportamentos entre projectos, foi criado um novo projecto (relógio), no qual se utilizou, com alterações mínimas, um *script* desenvolvido no projecto das rosáceas. Assim, um mesmo *script* com uma dada funcionalidade foi utilizado em dois contextos distintos.

Capítulo 7

Conclusões e Trabalho futuro

Nesta dissertação estudou-se em profundidade o ambiente e a linguagem de programação visual Scratch. No Capítulo 3, o estudo incidiu sobre a camada gráfica interactiva do Scratch, isto é, aquela com que o utilizador interage para construir os seus programas, tendo sido feito um levantamento exaustivo das funcionalidades do ambiente bem como dos princípios de programação suportados. Este estudo foi acompanhado pela aprendizagem da linguagem Squeak Smalltalk, na qual está implementada a camada inferior do Scratch, ou seja, a camada que lhe dá vida. Daqui, partiu-se para a análise do modelo de interactividade do Scratch, implementado pelo *framework* Morphic (Capítulo 4). Aqui se conheceu a entidade base de definição de interfaces neste *framework*: o *morph*. Analisou-se o modo de construção de interfaces complexas e as ideias chave por detrás deste *framework*. Uma vez adquirido todo este conhecimento, desenrolou-se todo o trabalho de carácter mais técnico. Foram documentados os aspectos estruturais mais importantes do Scratch, através de diagramas e esquemas, e desenvolvidas novas funcionalidades interactivas com vista a enriquecer as potencialidades do ambiente e da linguagem de programação (Capítulo 5).

Ao Scratch original foi adicionada a capacidade do utilizador definir os seus próprios blocos, os SDUs, que encapsulam comportamentos definidos por empilhamentos de blocos. À volta desta funcionalidade central foram desenvolvidas outras que a suportam: uma nova categoria de blocos, uma nova aba, um novo bloco que permite executar estes SDUs, consulta, modificação e eliminação dos SDUs, ecrãs de ajuda e sinalização de erros, escrita e leitura de projectos com SDUs e exportação e importação de SDUs.

Tal como exemplificado no Capítulo 6, estes novos blocos que o utilizador pode criar afiguram-se como o mecanismo que permite a criação dos padrões de comportamento que podem ser facilmente reutilizados em novos contextos, meta que havia sido definida nos objectivos a atingir.

Esta é uma das principais contribuições deste trabalho: a capacidade de reutilizar facilmente este tipo de *scripts*, quer dentro do mesmo projecto quer entre projectos diferentes, e de os partilhar entre utilizadores desta modificação do Scratch, promovendo a vertente colaborativa do mesmo.

Uma outra contribuição a destacar é o facto do resultado final deste trabalho, na forma de dissertação, constituir uma forma de documentação extensiva da componente técnica e de baixo nível do Scratch. Isto vem colmatar um dos principais problemas associados ao código fonte do Scratch, que é a sua falta de documentação, que até é assinalada na página *Web* onde o MIT disponibiliza o código fonte do Scratch: "Uma vez que o código fonte do Scratch não é destinado ao consumo público, não se deve esperar o nível de comentários ou de documentação da API que possuem sistemas como o Java, onde o ponto principal é ajudar os programadores

a desenvolver sobre a biblioteca de classes. Francamente, se esperássemos para libertar o código fonte do Scratch até que estivesse extensivamente e lindamente documentado, ele nunca seria publicado" [Scr12b].

Esta documentação ganha importância na medida em que existe interesse neste conhecimento técnico por parte do ecossistema existente na secção dos tópicos avançados do fórum do Scratch. A colaboração que existe entre entusiastas do Scratch que desenvolvem colectivamente as suas modificações deste, ajudando-se mutuamente e partilhando conhecimento adquirido, prova que há interesse, não só na vertente de alto nível do Scratch, mas também neste conhecimento mais técnico.

Existe alguma relutância por parte da equipa de desenvolvimento do Scratch em colocar este no mesmo nível de complexidade do BYOB. A implementação desenvolvida nesta dissertação sobe o tecto do Scratch, mas não o suficiente para o tornar demasiado complexo, situando-se entre o Scratch e o BYOB, ficando mais próximo do primeiro devido ao grau de complexidade do segundo. Ao mesmo tempo, a modificação do Scratch desenvolvida apresenta-se como uma implementação adiantada das ideias que a equipa de desenvolvimento do Scratch pretende integrar na sua próxima versão, no que concerne à possibilidade de definição de blocos [Scr10a].

A implementação da extensão do Scratch desenvolvida deixou certos pormenores por resolver. Como tal, um trabalho futuro incidiria sobre a adição de suporte total da nova entidade (SDU) às funcionalidades que ficaram parcialmente abrangidas: exportação de *Sprites*, importação de projectos, duplicação e cópia de SDUs entre *Sprites*, exportação e importação de SDUs com variáveis e listas.

A disponibilização na *Web* de projectos desenvolvidos nesta modificação do Scratch vê-se restringida pelas questões de licenciamento do código fonte do Scratch, que proíbem explicitamente a implementação da capacidade de enviar projectos para qualquer página do Scratch do MIT [Scr12a]. No entanto, não existe qualquer restrição relativamente a qualquer outra página *Web*. Sabendo isto, foi desenvolvida uma solução pela comunidade do Scratch para ultrapassar esta problemática, o Mod Share [Mod11], que é um portal onde podem ser depositados projectos desenvolvidos com modificações do Scratch. As modificações podem então implementar a funcionalidade de enviar os seus projectos para este portal. É permitido o envio de qualquer projecto que use a extensão de ficheiros original do Scratch, no entanto também suporta projectos desenvolvidos com modificações como o BYOB, Panther, etc, que geram ficheiros com extensão diferente. Desta forma, a investigação neste sentido consistiria em: 1) saber quais as condições necessárias para que os projectos desenvolvidos neste trabalho, de extensão `.extsb`, pudessem lá ser depositados e partilhados e 2) dotar a modificação do Scratch desenvolvida com a dita funcionalidade de envio de projectos. Este aspecto fomentaria a partilha de projectos desenvolvidos com a modificação do Scratch criada neste trabalho e ajudaria à sua divulgação. No entanto, e dado que um dos pontos diferenciadores deste trabalho é a capacidade de exportar SDUs, seria também interessante ter a possibilidade de ter um repositório *online* de SDUs desenvolvidos, que poderiam ser descarregados para serem utilizados nos mais diversos projectos dos utilizadores. A integração de uma funcionalidade de transferência de um SDU de e para este repositório seria algo a explorar.

Uma outra ideia que pode ser explorada consiste em dotar os *Sprites* de conhecimento inteligente, e.g., usando uma base de conhecimento em Prolog. Neste âmbito podem-se imaginar jogos de perguntas e respostas, onde o conhecimento para validação das respostas e/ou construção das perguntas estaria definido, não através de blocos do Scratch, como o **if-else** ou outros, mas numa base de conhecimento constituída por factos e predicados.

Para além disto, pode-se imaginar a aplicação destes SDUs no trabalho desenvolvido por Rui da Silva [dS12], que acrescenta novas estruturas de dados ao Scratch (e.g., grafos). Neste contexto, poderiam surgir projectos como a simulação de um jogo, onde o jogador avançaria

entre os diversos nós do grafo resolvendo desafios propostos em cada nó. Esses desafios estariam encapsulados sob a forma dos SDUs aqui desenvolvidos, o que daria azo à criação de diversos tipos de jogos com múltiplos desafios, que poderiam ser escolhidos a partir de uma biblioteca de SDUs.

Numa nota final, apenas há que referenciar qual o destino que o Scratch vai seguir. Neste momento, encontra-se em desenvolvimento a versão 2.0 do Scratch, a qual o transformará num ferramenta exclusivamente *Web*, uma vez que funcionará via *Web browser*, sem necessidade de descarregar ou enviar ficheiros. Quer o desenvolvimento quer a execução de projectos serão feitos num ambiente suportado pela tecnologia Flash. Será também criada uma versão *offline* para usar em situações em que a ligação à Internet é fraca ou inexistente. Esta nova versão permitirá executar projectos desenvolvidos na versão actual (1.4). De entre as novidades contam-se a possibilidade do utilizador definir os seus próprios blocos assim como uma funcionalidade "MyScripts", que terá o mesmo efeito daquela que foi desenvolvida neste trabalho: funcionará como uma biblioteca de *scripts* (blocos) que o utilizador poderá usar [Scr10b]. Também a criação de um repositório *online* de blocos definidos pelos utilizadores está a ser avaliada pela equipa de desenvolvimento do Scratch, com vista a ser introduzida na sua nova versão [Scr11]. A migração para a *Web* trará consigo a capacidade de partilha de projectos em dispositivos móveis e *tablets* (excepção feita para os da Apple devido aos conflitos com a tecnologia Flash) e a integração com ferramentas de *media* social como o Facebook, Twitter, Flickr, entre outros.

Bibliografia

[BD04] Marat Boshernitsan and Michael Downes. Visual Programming Languages: A Survey. Technical Report UCB/CSD-04-1368, Computer Science Division (EECS), University of California, Berkeley, Berkeley, CA 94720-1776 USA, December 2004.

[BDNP07] A.P. Black, S. Ducasse, O. Nierstrasz, and D. Pollet. *Squeak by Example*. Square Bracket Associates, 2007.

[Beg96] Andrew Begel. LogoBlocks: A Graphical Programming Language for Interacting with the World. Unpublished Advanced Undergraduate Project, MIT Media Lab, May 1996.

[BS94] Margaret M. Burnett and Benjamin Summers. Some Real-World Uses of Visual Programming Systems. Technical report, Oregon State University, Corvallis, OR, USA, 1994.

[Bur99] M. M. Burnett. Visual Programming. In *Wiley Encyclopedia of Electrical and Electronics Engineering*. John Wiley & Sons Inc., 1999.

[CDP00] Stephen Cooper, Wanda Dann, and Randy Pausch. Alice: a 3-D tool for introductory programming concepts. *J. Comput. Small Coll.*, 15:107–116, April 2000.

[CHK+93] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch what I do: programming by demonstration*. Number 0-262-03213-9. The MIT Press, Cambridge, MA, USA, 1993.

[dS12] Rui da Silva. Extensões estruturais e visuais para o ambiente de programação visual Scratch (tese em desenvolvimento). Master's thesis, Universidade do Minho, 2012.

[HM10] Brian Harvey and Jens Mönig. Bringing "No Ceiling" to Scratch: Can One Language Serve Kids and Computer Scientists? In *Proceedings of Constructionism 2010*, 2010.

[Kay05] Alan Kay. Squeak Etoys Authoring & Media. *Viewpoints*, (818), February 2005.

[Koh07] Lutz Kohl. Puck - a visual programming system for schools. In Raymond Lister and Simon, editors, *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)*, volume 88 of *CRPIT*, pages 191–194, Koli National Park, Finland, 2007. ACS.

[Köl10] Michael Kölling. Comparing Scratch, Alice and Greenfoot, December 2010.

[Mal01] John Maloney. *An Introduction to Morphic: The Squeak User Interface Framework*. Prentice Hall, 2001.

[MBK+04] John Maloney, Leo Burd, Yasmin Kafai, Natalie Rusk, Brian Silverman, and Mitchel Resnick. Scratch: A Sneak Preview. In *C5 '04: Proceedings of the Second International Conference on Creating, Connecting and Collaborating through Computing*, pages 104–109, Washington, DC, USA, 2004. IEEE Computer Society.

[MH07] Andrés Monroy-Hernández. ScratchR: sharing user-generated programmable media. In *Proceedings of the 6th international conference on Interaction design and children*, IDC '07, pages 167–168, New York, NY, USA, 2007. ACM.

[MH11] Jens Mönig and Brian Harvey. *BYOB Reference Manual, Version 3.1*, 2011.

[MHR08] Andrés Monroy-Hernández and Mitchel Resnick. FEATURE: Empowering kids to create and share programmable media. *interactions*, 15:50–53, March 2008.

[ML07] David J. Malan and Henry H. Leitner. Scratch for budding computer scientists. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, SIGCSE '07, pages 223–227, New York, NY, USA, 2007. ACM.

[Mod11] Mod Share Team. Mod Share. http://modshare.tk/, 2011.

[Mön08] Jens Mönig. Build Your Own Blocks in Scratch: a Prototype. October 2008.

[Mön09a] Jens Mönig. BYOB 2.0. 2009.

[Mön09b] Jens Mönig. Syntax-Elements for Smalltalk: A Scratch-like GUI for Smalltalk-80. February 2009.

[MPK+08] John H. Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. Programming by choice: urban youth learning programming with scratch. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, SIGCSE '08, pages 367–371, New York, NY, USA, 2008. ACM.

[MRR+08] John Maloney, Mitchel Resnick, Natalie Rusk, Kylie A. Peppler, and Yasmin B. Kafai. Media designs with scratch: what urban youth can learn about programming in a computer clubhouse. In *Proceedings of the 8th International Conference of the Learning Sciences*, volume 3 of *ICLS'08*, pages 81–82. International Society of the Learning Sciences, 2008.

[MRR+10] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The Scratch Programming Language and Environment. *Trans. Comput. Educ.*, 10(16):16:1–16:15, 2010.

[MS95] John H. Maloney and Randall B. Smith. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th annual ACM symposium on User interface and software technology*, UIST '95, pages 21–28, New York, NY, USA, 1995. ACM.

[Mye90] Brad A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97 – 123, 1990.

[Res02] Mitchel Resnick. Rethinking learning in the digital age. *Computer*, 46(1):32–37, 2002.

[Res03] Mitchel Resnick. Playful Learning and Creative Societies. *Education Update*, VIII(6), February 2003.

[Res07] Mitchel Resnick. Sowing the seeds for a more creative society. *Learning and Leading with Technology*, 35(4):18–22, 2007.

[RMMH+09] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: programming for all. *Commun. ACM*, 52:60–67, November 2009.

[Roq07] Ricarose Vallarta Roque. OpenBlocks : an extendable framework for graphical block programming systems. Master's thesis, MIT. Dept. of Electrical Engineering and Computer Science, May 2007.

[RSB05] Barbara G. Ryder, Mary Lou Soffa, and Margaret Burnett. The impact of software engineering research on modern progamming languages. *ACM Trans. Softw. Eng. Methodol.*, 14(1049-331X):431–477, October 2005.

[Rus09] Natalie Rusk. Programming concepts and skills supported in Scratch. http://scratched.media.mit.edu/resources/scratch-programming-concepts, August 2009.

[Sap12] Sapo Kids. Rosáceas... http://kids.sapo.pt/scratch/projects/EduScratch/2927, Janeiro 2012.

[Scr10a] Scratch Suggestions. Merge BYOB3 and Scratch! http://suggest.scratch.mit.edu/forums/60449-suggestions/suggestions/941139-merge-byob3-and-scratch-, July 2010.

[Scr10b] Scratch Suggestions. MyScripts Feature. http://suggest.scratch.mit.edu/forums/60449-suggestions/suggestions/971633-myscripts-feature, August 2010.

[Scr11] Scratch Suggestions. Block Store. http://suggest.scratch.mit.edu/forums/60449-suggestions/suggestions/1431893-block-store, January 2011.

[Scr12a] Scratch Team - MIT. Scratch - Home - imagine, program, share. http://scratch.mit.edu/, April 2012.

[Scr12b] Scratch Team - MIT. Scratch Source Code. http://info.scratch.mit.edu/Source_Code, March 2012.

[SL08] Paolo A. G. Sivilotti and Stacey A. Laugel. Scratching the surface of advanced topics in software engineering: a workshop module for middle school students. In *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, number 978-1-59593-799-5 in SIGCSE '08, pages 291–295, New York, NY, USA, 2008. ACM.

[Smi75] David Canfield Smith. *PYGMALION: A Creative Programming Environment*. PhD thesis, Stanford University, May 1975.

[Squ08] Squeak Swiki. How Morphic works. http://wiki.squeak.org/squeak/30, September 2008.

[Sut63] Ivan Edward Sutherland. *Sketchpad: A man-machine graphical communication system*. PhD thesis, Massachusetts Institute of Technology, January 1963.

[Sut66] William R. Sutherland. *The On-Line Graphical Specification of Computer Procedures*. PhD thesis, Massachusetts Institute of Technology, January 1966.

[UCK+10] Ian Utting, Stephen Cooper, Michael Kölling, John Maloney, and Mitchel Resnick. Alice, Greenfoot, and Scratch – A Discussion. *Trans. Comput. Educ.*, 10(4):17:1–17:11, November 2010.

[Zha07] K. Zhang. *Visual languages and applications*. Springer, 2007.

Anexos

Anexo A

Diagramas de classes

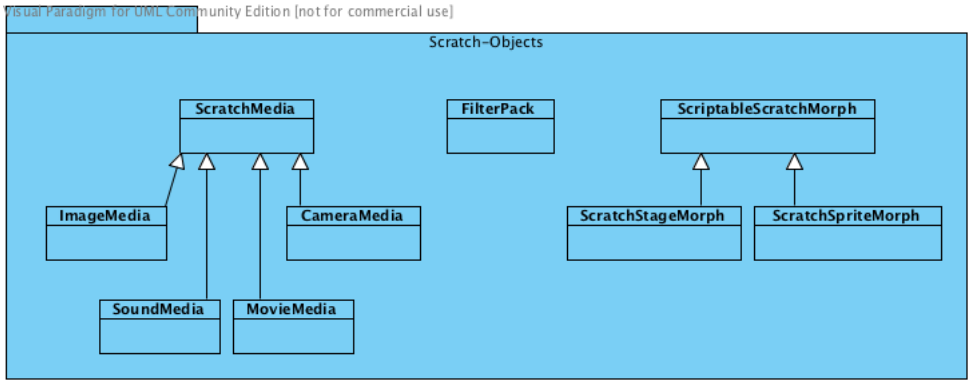


Figura A.1: Categoria Scratch-Objects.

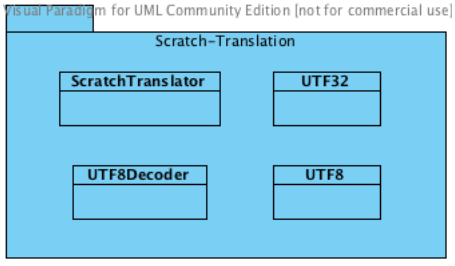


Figura A.2: Categoria Scratch-Translation.

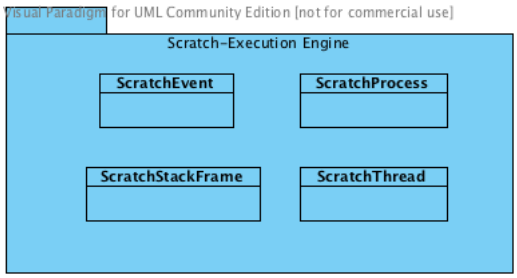


Figura A.3: Categoria Scratch-Execution Engine.

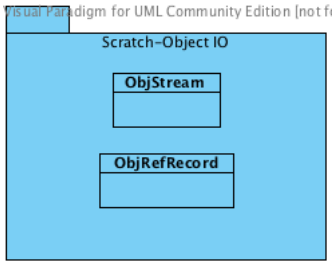


Figura A.4: Categoria Scratch-Object IO.

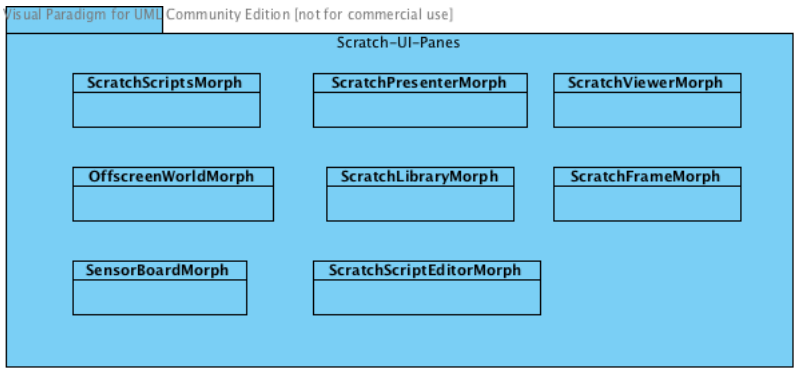


Figura A.5: Categoria Scratch-UI-Panes.

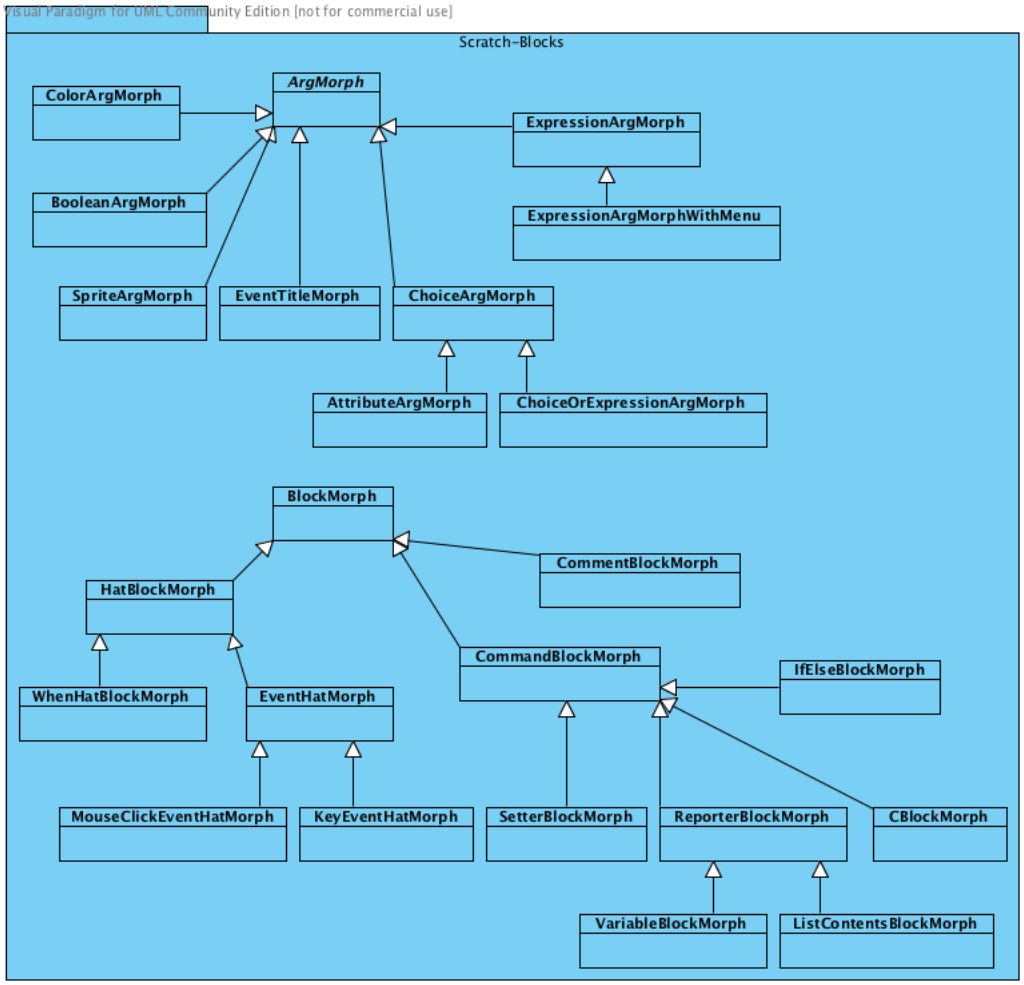


Figura A.6: Categoria Scratch-Blocks.

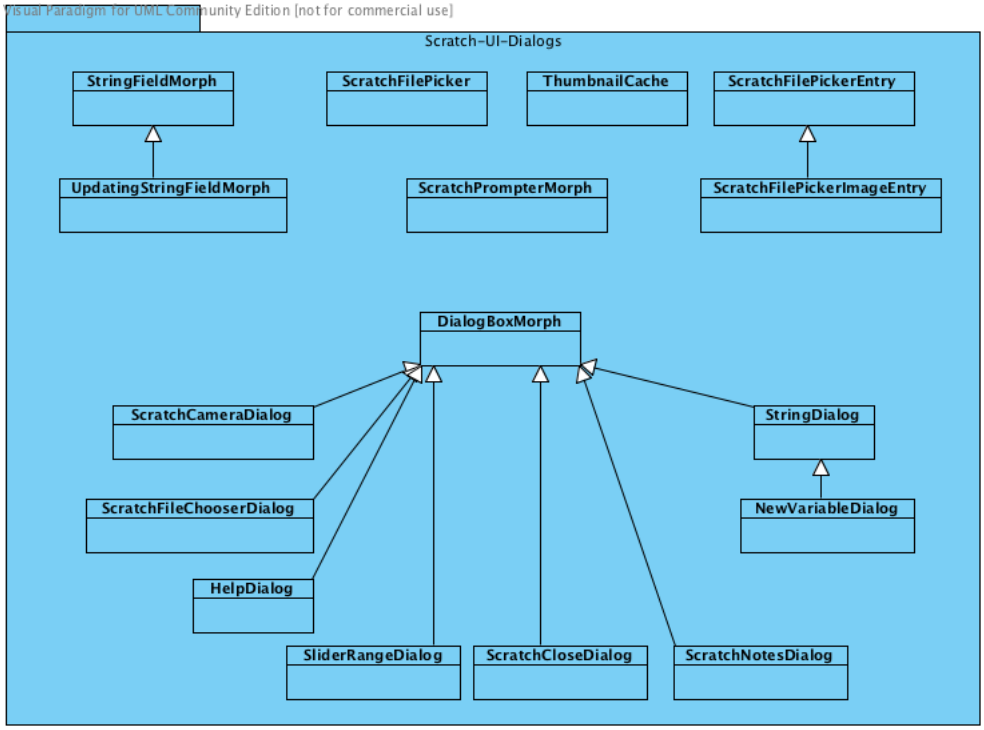


Figura A.7: Categoria Scratch-UI-Dialogs.

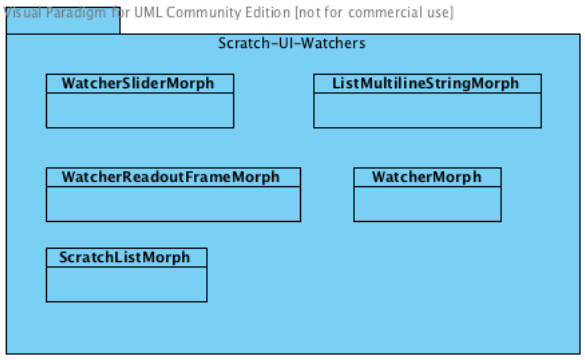


Figura A.8: Categoria Scratch-UI-Watchers.

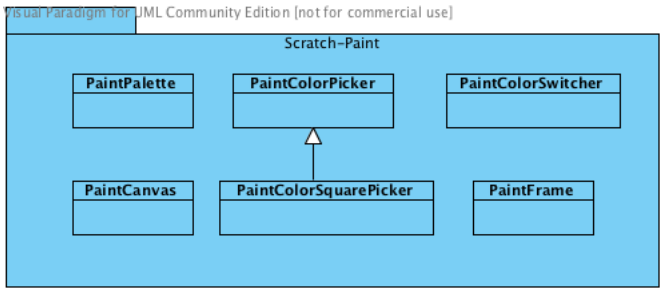


Figura A.9: Categoria Scratch-Paint.

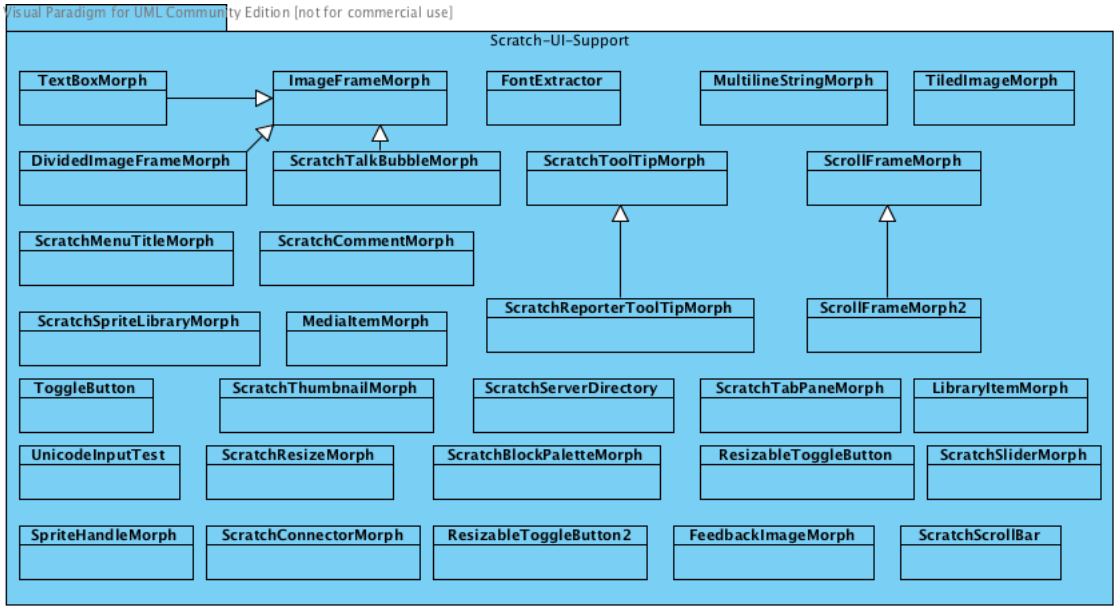


Figura A.10: Categoria Scratch-UI-Support.

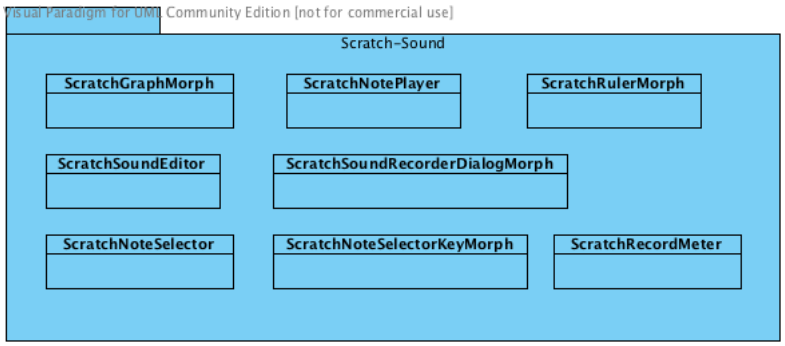


Figura A.11: Categoria Scratch-Sound.

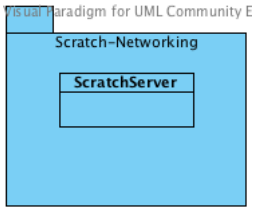


Figura A.12: Categoria Scratch-Networking.

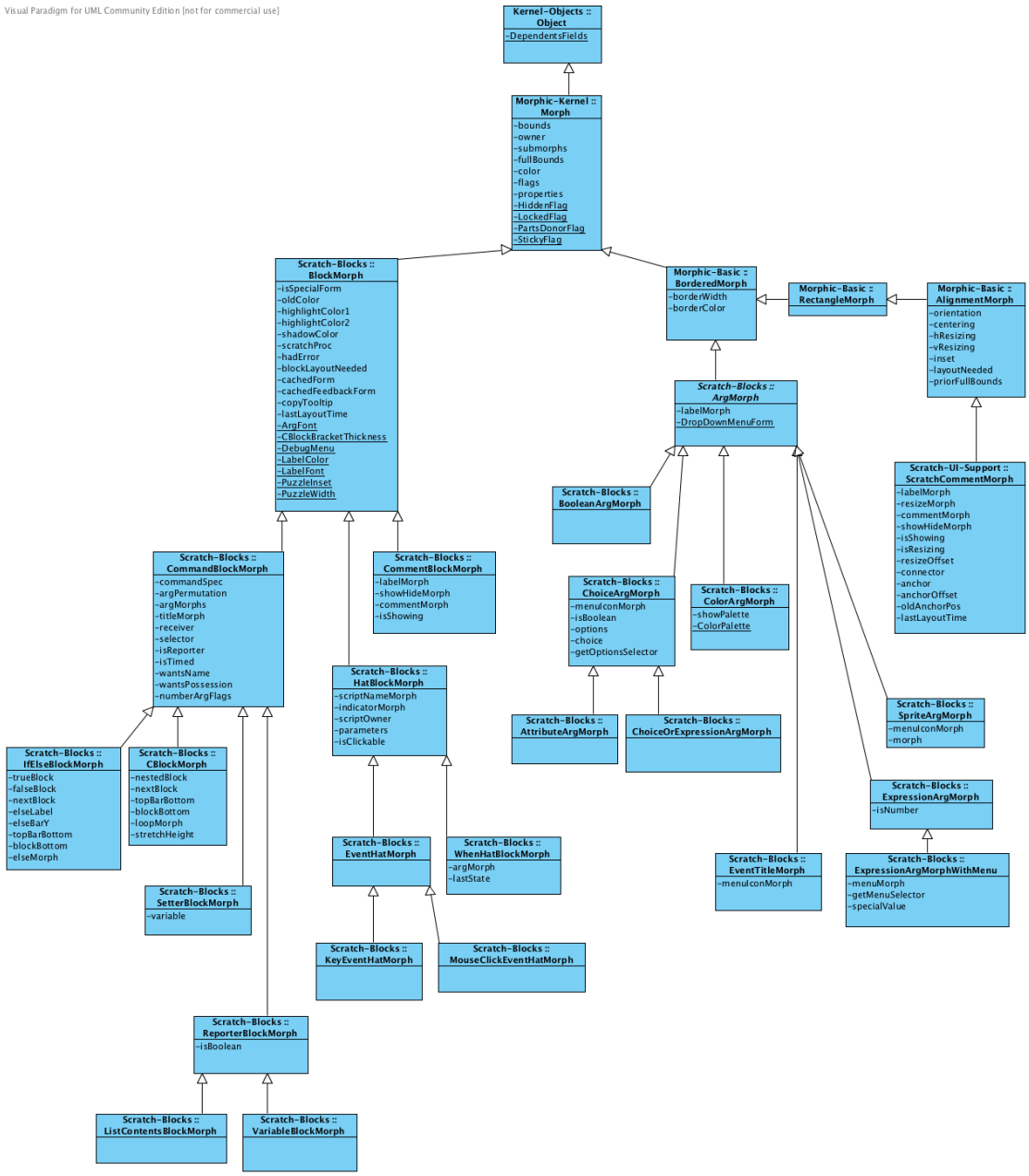
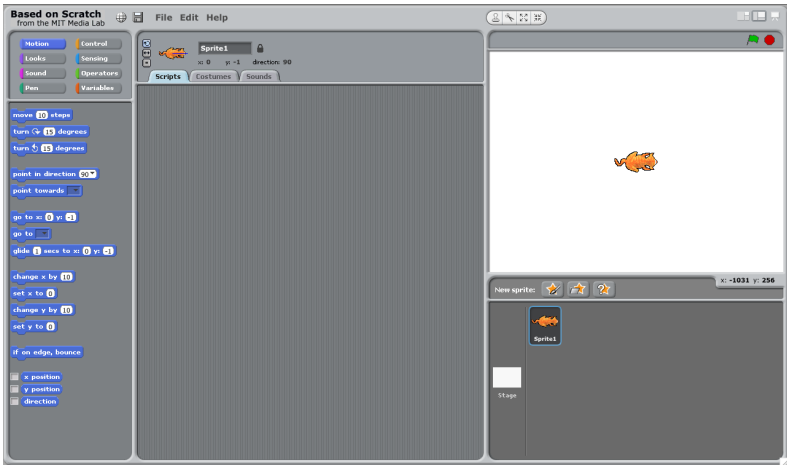


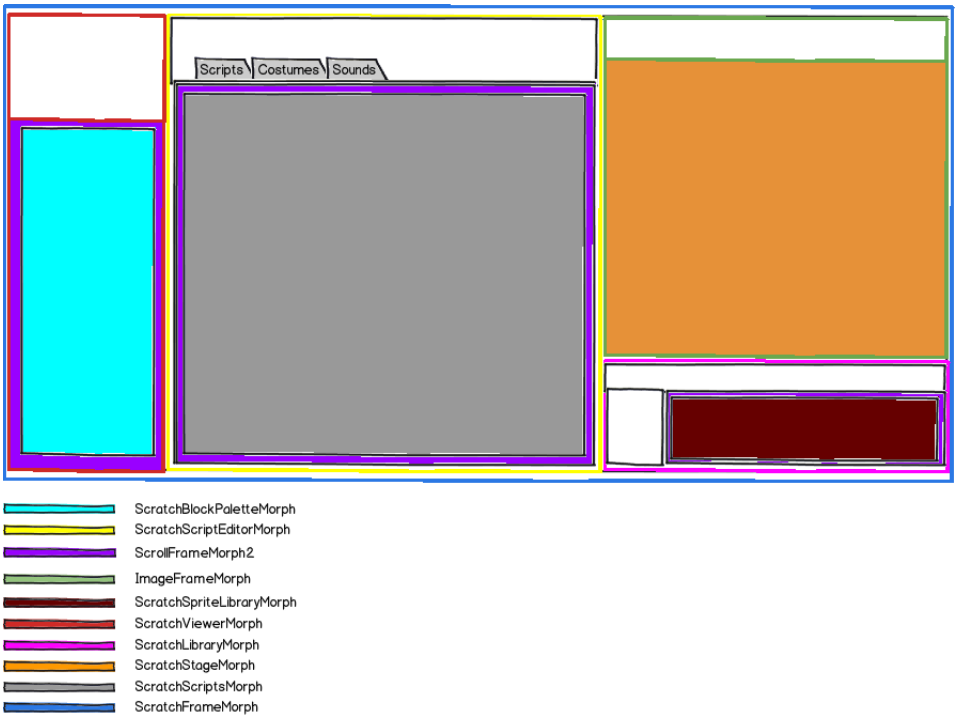
Figura A.13: Diagrama de classes dos blocos do Scratch.

Anexo B

Estrutura da interface do Scratch



(a) Interface do Scratch.



(b) Componentes da interface do Scratch.

Figura B.1: Mapeamento dos elementos da interface do Scratch para as classes que os representam.

Anexo C

Código

O código que aqui se apresenta serve de suporte ao documento.

```
acceptDroppingMorph: aMorph event: evt
        "Copy the dropped scripts of another object into its target."
        ...
        (aMorph isKindOf: BlockMorph) ifTrue: [
                evt hand rejectDropMorph: aMorph event: evt.
                ((self ownerThatIsA: ScratchFrameMorph) scriptsPane
                    currentCategory = 'Test') ifFalse: [
                        target addStack: aMorph fullCopy]].

        (aMorph isKindOf: ScratchCommentMorph) ifTrue: [
                evt hand rejectDropMorph: aMorph event: evt.
                ((self ownerThatIsA: ScratchFrameMorph) scriptsPane
                    currentCategory = 'Test') ifFalse: [
                        target addComment: aMorph fullCopy]].
```

Código C.1: LibraryItemMorph»acceptDroppingMorph:event:

```
duplicateNoAttach
    "Duplicate this sprite, but do not attach to the hand."

    | newSprite frame |
    newSprite _ self fullCopy.
    newSprite position: (newSprite position + 20).
    frame _ self ownerThatIsA: ScratchFrameMorph.
    frame ifNotNil: [
        frame workPane addMorphFront: newSprite.
        frame workPane sprites addLast: newSprite].
    ^ newSprite
```

Código C.2: ScriptableScratchMorph»duplicateNoAttach

```
fullCopy
    "Produce a copy of me with my entire tree of submorphs. Morphs mentioned more
    than once are all directed to a single new copy. Simple inst vars are not
    copied, so you must override to copy Arrays, Forms, editable text, etc."

    | dict new |
    dict _ IdentityDictionary new: 1000.
    new _ self copyRecordingIn: dict.
    new allMorphsDo: [:m | m updateReferencesUsing: dict].
    ^ new
```

Código C.3: Morph≫fullCopy

```
copyRecordingIn: dict
        "Copy my fields and scripts."

        | newCopy newBlocksBin |
        (self respondsTo: #sayNothing) ifTrue: [self sayNothing]. "remove talk
            bubble before copying"
        newCopy _ super copyRecordingIn: dict.
        newCopy renewFilterPack.
        newBlocksBin _ blocksBin fullCopy.
        newBlocksBin allMorphsDo: [:m |
                (m isKindOf: HatBlockMorph) ifTrue: [m scriptOwner: newCopy].
                (m isKindOf: CommandBlockMorph) ifTrue: [m mapReceiver: self to:
                    newCopy]].
        newCopy vars: vars copy lists: (self copyListsFor: newCopy) blocksBin:
            newBlocksBin.
        newCopy objName: self nextInstanceName.
        newCopy setMedia: (media collect: [:el | el copy]).
        newCopy lookLike: costume mediaName.
        ^ newCopy
```

Código C.4: Versão inicial do método ScriptableScratchMorph≫copyRecordingIn:

```
vars: varsDict lists: listsDict blocksBin: aBlocksBin
    "Private! Set my variables and blocks bin. Used by copyRecordingIn:."

    vars _ varsDict.
    lists _ listsDict.
    blocksBin _ aBlocksBin.
```

Código C.5: Versão inicial do método ScriptableScratchMorph≫vars:lists:blocksBin:

```
vars: varsDict lists: listsDict blocksBin: aBlocksBin testBlocksBin:
    aTestBlocksBin
        ...
        testBlocksBin _ aTestBlocksBin.
```

Código C.6: Versão final do método ScriptableScratchMorph≫vars:lists:blocksBin:

```
copyRecordingIn: dict
        | newCopy newBlocksBin newTestBlocksBin |
        ...
        newTestBlocksBin _ testBlocksBin fullCopy.
        newTestBlocksBin submorphsDo: [:m | (m isKindOf: EventHatMorph)
                ifTrue: [
                        m scriptOwner: newCopy.
                        m addBlockName: nil.
                        m nextBlock ifNotNil: [m nextBlock delete]]
                ifFalse: [m delete]].

        newCopy vars: vars copy lists: (self copyListsFor: newCopy) blocksBin:
            newBlocksBin testBlocksBin: newTestBlocksBin.
        ...
```

Código C.7: Versão final do método ScriptableScratchMorph»copyRecordingIn:

```
rightButtonMenu
...
    (owner isKindOf: ScratchBlockPaletteMorph) ifFalse: [
        ...
        (currentCategory = 'Scripts') ifTrue: [
            menu addLine.
            menu add: 'save script' action: #saveScript].
        ...
    ].
...
```

Código C.8: HatBlockMorph»rightButtonMenu

```
rightButtonMenu
...
    (owner isKindOf: ScratchBlockPaletteMorph) ifFalse: [
        ...
        ((currentCategory = 'Scripts') or: [currentCategory = 'Test' and:
            [(self topBlock isKindOf: EventHatMorph) not]]) ifTrue: [
                menu addLine.
                menu add: 'save script' action: #saveScript
        ].
        ...
    ].
...
    (#(presentHelpScreen duplicate delete saveScript) includes: choice)
        ifTrue: [^ self perform: choice].
...
```

Código C.9: CommandBlockMorph»rightButtonMenu

```
saveScript

    ^ self subclassResponsibility.
```

Código C.10: BlockMorph»saveScript

```
saveScript
    "This class is deprecated. As such, no implementation of this method is
        required."
```

Código C.11: CommentBlockMorph»saveScript

```
saveScript

    | firstBlock |
    (firstBlock _ self nextBlock) ifNil: [
        self beep.
        DialogBoxMorph warn: 'No script found.'.
        ^ nil]
    ifNotNil: [self receiver addBlockWithBody: firstBlock].
```

Código C.12: HatBlockMorph»saveScript

```
saveScript

    | firstBlock body tb |
    firstBlock _ self topBlock.
    (firstBlock isKindOf: HatBlockMorph)
        ifTrue: [body _ firstBlock nextBlock]
        ifFalse: [body _ firstBlock].
    tb _ ((self ownerThatIsA: ScratchFrameMorph) libraryPane spriteThumbnails
        detect: [:sp | sp isSelected] ifNone: [nil]).
    tb ifNotNil: [self receiver: tb target].
    self receiver addBlockWithBody: body.
```

Código C.13: CommandBlockMorph»saveScript

```
saveScript

    (owner isKindOf: BlockMorph) ifTrue: [
        ^ owner saveScript].
    (owner isKindOf: ScratchScriptsMorph) ifTrue: [
        self beep.
        DialogBoxMorph warn: 'Cannot create a script with only a reporter block.'.
        ^ nil.
    ]
```

Código C.14: ReporterBlockMorph»saveScript

```
addBlockWithBody: aBlock

    | sFrame result blockName isGlobal def |
    (sFrame _ self ownerThatIsA: ScratchFrameMorph) ifNil: [self beep. ^ nil].
    (self isKindOf: ScratchStageMorph) ifTrue: [
        result _ StringDialog askWithCancel: 'Script name?'.
        result = '' ifTrue: [^ nil].
        blockName _ result asUTF8.
        isGlobal _ false.
    ] ifFalse: [
        result _ NewVariableDialog askWithCancel: 'Script name?'.
        (result = #cancelled) ifTrue: [^ nil].
        isGlobal _ result second not.
        blockName _ result first asUTF8].
...
```

Código C.15: ScriptableScratchMorph»addBlockWithBody:

```
initialize

    super initialize.
    spec _ nil.
    body _ nil.
    isGlobal _ true.
```

Código C.16: UserBlockDefinition»initialize

```
initialize
...
    userBlocks _ Dictionary new.
...
```

Código C.17: ScriptableScratchMorph»initialize

```
addBlockWithBody: aBlock
...
    def _ UserBlockDefinition new spec: blockName; body: aBlock; isGlobal:
        isGlobal.
...
```

Código C.18: ScriptableScratchMorph»addBlockWithBody:

```
existsBlock: aSpec global: aBoolean

    | isGlobal stage |
    isGlobal _ aBoolean.
    isGlobal ifTrue: [
        stage _ self ownerThatIsA: ScratchStageMorph.
        stage sprites do: [:sp | (sp existsBlock: aSpec) ifTrue: [^ true]].
        ^ false.
    ]
    ifFalse: [^ self existsBlock: aSpec].
```

Código C.19: ScratchSpriteMorph»existsBlock:global:

```
existsBlock: aSpec global: aBoolean

    ^ self existsBlock: aSpec
```

Código C.20: ScratchStageMorph»existsBlock:global:

```
existsBlock: aString

    ^ userBlocks includesKey: aString
```

Código C.21: ScriptableScratchMorph»existsBlock:

```
addBlockWithBody: aBlock
...
    (self existsBlock: blockName global: isGlobal) ifTrue: [
        self beep.
        DialogBoxMorph warn: 'That block name is already in use!'.
        ^ nil].
    self createUserBlockWithDefinition: def global: isGlobal.
...
```

Código C.22: ScriptableScratchMorph»addBlockWithBody:

```
addVariable: varName
    "Add a new user variable with the given name to this object. Do nothing if
    the variable already exists or is built in."

    (vars includesKey: varName asString) ifFalse: [
        vars at: varName asString put: 0].
    self isClone: false.
```

Código C.23: ScriptableScratchMorph»addVariable:

```
createUserBlockWithDefinition: aDefinition global: aBoolean

    | isGlobal stage |
    isGlobal _ aBoolean.
    isGlobal ifTrue: [
        stage _ self ownerThatIsA: ScratchStageMorph.
        stage sprites do: [:sp | sp createUserBlockWithDefinition: aDefinition].

        "global scripts with local vars/lists must ensure those exist in all
            sprites"
        aDefinition body submorphsDo: [:m |
            ((m isKindOf: SetterBlockMorph) or: [m isKindOf: VariableBlockMorph])
                ifTrue: [
                (vars includes: m variable) ifTrue: [
                    stage sprites do: [:sp | sp addVariable: m variable].
                ]
            ].
            (m isKindOf: ListContentsBlockMorph) ifTrue: [
                (lists includes: m commandSpec) ifTrue: [
                    stage sprites do: [:sp | sp lists at: m commandSpec put:
                        (ScratchListMorph new listName: m commandSpec target: sp)
                            ].
                ]
            ].
        ]
    ]
    ifFalse: [^ self createUserBlockWithDefinition: aDefinition]
```

Código C.24: ScratchSpriteMorph»createUserBlockWithDefinition:global:

```
createUserBlockWithDefinition: aDefinition global: aBoolean

    ^ self createUserBlockWithDefinition: aDefinition
```

Código C.25: ScratchStageMorph»createUserBlockWithDefinition:global:

```
createUserBlockWithDefinition: aDefinition

    | definition blockName |
    definition _ aDefinition deepCopy.
    blockName _ definition spec.
    definition body newScriptOwner: self.
    userBlocks at: blockName put: definition
```

Código C.26: ScriptableScratchMorph»createUserBlockWithDefinition:

```
addBlockWithBody: aBlock
...
    sFrame viewerPane currentCategory: 'MyScripts'.
```

Código C.27: ScriptableScratchMorph»addBlockWithBody:

```
viewerPageForCategory: aCategoryName
    "Answer a morph containing blocks for the given category for use in the
    given ScratchViewer."
...
    aCategoryName = 'MyScripts' ifTrue: [^ self viewerPageForMyScripts].
...
```

Código C.28: ScriptableScratchMorph»viewerPageForCategory:

```
viewerPageForMyScripts

    | x y stage b bin hasBlocks localDefs globalDefs |
    bin _ ScratchBlockPaletteMorph new.
    x _ 12.
    y _ 10.
    hasBlocks _ false.
    userBlocks ifNotNil: [
        hasBlocks _ (userBlocks size > 0)].
    hasBlocks ifTrue: [
        localDefs _ (userBlocks values select: [:def | def isGlobal not]) sort:
            [:fst :snd | fst spec <= snd spec].
        globalDefs _ (userBlocks values select: [:def | def isGlobal]) sort:
            [:fst :snd | fst spec <= snd spec].
        globalDefs size > 0 ifTrue: [
            globalDefs do: [:def |
                b _ self newUserBlockFor: def.
                bin addMorph: (b position: (x@y)).
                y _ y + b height + 3]].
        y _ y + 7.
        bin addMorph: ((ImageMorph new form: (ScratchFrameMorph skinAt: #connector
            ))
            position: x@y).
        y _ y + 20.
        localDefs size > 0 ifTrue: [
            localDefs do: [:def |
                b _ self newUserBlockFor: def.
                bin addMorph: (b position: (x@y)).
                y _ y + b height + 3]]].
    ^ bin.
```

Código C.29: ScriptableScratchMorph»viewerPageForMyScripts

```
newUserBlockFor: aDefinition

    | block |
    block _ UserCommandBlockMorph new.
    block receiver: self.
    block color: (self class blockColorFor: 'MyScripts').
    block commandSpec: aDefinition spec.
    block selector: #userBlockWarn. "import block is responsible for running this
        block"
    ^ block.
```

Código C.30: ScriptableScratchMorph»newUserBlockFor:

```
userBlockWarn

    self beep.
    DialogBoxMorph warn: 'A user-defined block can only be executed inside
        an import block.'.
    ^ nil.
```

Código C.31: UserCommandBlockMorph»userBlockWarn

```
CBlockMorph subclass: #ImportCBlockMorph
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Scratch-Blocks'
```

Código C.32: Criação da classe ImportCBlockMorph

```
blockSpecs
...
        ('import'                       i    importScript)
...
```

Código C.33: ScriptableScratchMorph»blockSpecs

```
blockFromSpec: spec color: blockColor
    "Create a block from the given block specification. Answer nil if I don't
    implement the block spec selector."
...
    "basic block type: normal or C-shaped"
    (blockType includes: $c)
        ifTrue: [
            selector = #doIfElse
                ifTrue: [block _ IfElseBlockMorph new isSpecialForm: true]
                ifFalse: [block _ CBlockMorph new isSpecialForm: true]]
...
    (blockType includes: $i) ifTrue: [block _ ImportCBlockMorph new isSpecialForm:
         true].
...
```

Código C.34: ScriptableScratchMorph»blockFromSpec:color:

```
evaluateSpecialForm
    "Evaluates the current special form expression.  Requires that no arguments
    have been evaluated, and that the current expression be a special form."

    self perform: stackFrame expression selector.
```

Código C.35: ScratchProcess»evaluateSpecialForm

```
nestedBlock

    ^ nestedBlock
```

Código C.36: ImportCBlockMorph»nestedBlock

```
showError
    "Make this block to show an error."

    super color: (Color r: 0.831 g: 0.156 b: 0.156).
```

Código C.37: BlockMorph»showError

```
stop

    scratchProc ifNotNil: [
        self changed.
        scratchProc stop.
        scratchProc _ nil].
```

Código C.38: BlockMorph»stop

```
stop
    "Permanently terminates this process."

    stackFrame ifNotNil: [stackFrame stopMIDI; stopMotors; stopTalkThinkAsk].
    readyToYield _ true.
    readyToTerminate _ true.
    topBlock ifNotNil: [topBlock scratchProc: nil].
```

Código C.39: ScratchProcess»stop

```
importBlockWarn

    self beep.
    DialogBoxMorph warn: 'An import block can only have 1 block as argument
        and it must be a user-defined one.'.
    ^ nil.
```

Código C.40: ImportCBlockMorph»importBlockWarn

```
errorFlag: aBoolean
    "Set the error flag for this process."

    errorFlag _ aBoolean.
    stackFrame expression showError.
    stackFrame expression topBlock showErrorFeedback.
```

Código C.41: ScratchProcess»errorFlag:

```
updateCachedFeedbackForm
    "Create a cached feedback Form to show this stack's running (or error) status
        ."

    | outlineColor |
    cachedForm ifNil: [^ self].
    outlineColor _ (scratchProc notNil and: [scratchProc errorFlag])
        ifTrue: [Color r: 0.831 g: 0.156 b: 0.156]
        ifFalse: [Color gray: 0.953].

    cachedFeedbackForm _ cachedForm
        outlineWidth: 3
        color: outlineColor
        depth: 8.
```

Código C.42: BlockMorph»updateCachedFeedbackForm

```
importScript
    "Runs a user-defined block."

    | importBlock blockOwner blockName exp |
    importBlock _ stackFrame expression.
    self popStackFrame.

    "when importBlock is moved and catches a regular stack inside itself"
    ((importBlock nestedBlock isKindOf: UserCommandBlockMorph) not
        or: [importBlock firstBlockList size > 1])
        ifTrue: [
            importBlock showError.
            importBlock stop.
            importBlock importBlockWarn.
            self errorFlag: true.
            ^ self doReturn.
        ].
...
```

Código C.43: ScratchProcess»importScript

```
importScript
    "Runs a user-defined block."

    | importBlock blockOwner blockName exp |
    importBlock _ stackFrame expression.
    self popStackFrame.

    importBlock nestedBlock ifNil: [^ nil]. "when import block has no argument"

    "when importBlock is moved and catches a regular stack inside itself"
    ((importBlock nestedBlock isKindOf: UserCommandBlockMorph) not
        or: [importBlock firstBlockList size > 1])
        ifTrue: [
            importBlock showError.
            importBlock stop.
            importBlock importBlockWarn.
            self errorFlag: true.
            ^ self doReturn.
        ].
    blockOwner _ importBlock nestedBlock receiver.
    blockName _ importBlock nestedBlock commandSpec.
    exp _ (blockOwner definitionOfUserBlock: blockName) body blockSequence.
    self pushStackFrame: (ScratchStackFrame new expression: exp)
```

Código C.44: ScratchProcess»importScript

```
definitionOfUserBlock: spec
    "Returns the UserBlockDefinition for the user-defined block with spec <spec>"

    ^ self userBlocks at: spec ifAbsent: [nil]
```

Código C.45: ScriptableScratchMorph»definitionOfUserBlock:

```
click: evt
    "If a tool is selected, handle a click with that tool.
    Otherwise, toggle my process."

    evt hand toolType ifNotNil: [
        "handle mouse clicks when the mouse is in different modes"
        ^ self handleTool: evt hand toolType hand: evt hand].

    self topBlock toggleProcess.
```

Código C.46: BlockMorph»click:

```
doubleClick: evt
    "If I'm a block with a receiver that's in a Scratch window, execute me."

    self topBlock toggleProcess.
```

Código C.47: BlockMorph»doubleClick:

```
click: evt

    (self owner isKindOf: ImportCBlockMorph)
        ifTrue: [super click: evt]
        ifFalse: [self userBlockWarn]
```

Código C.48: UserCommandBlockMorph»click:

```
doubleClick: evt

    (self owner isKindOf: ImportCBlockMorph)
        ifTrue: [super doubleClick: evt]
        ifFalse: [self userBlockWarn]
```

Código C.49: UserCommandBlockMorph»doubleClick:

```
applyPrimitive
    "Apply the current expression (which must be a CommandBlock) to the current
    arguments (which must all have been evaluated)."

    | value |
    value _ stackFrame expression evaluateWithArgs: stackFrame arguments.

    "save the return value in the parent frame before popStackFrame because
    popFrame adds a frame while single-stepping"
    self returnValueToParentFrame: value.
    self popStackFrame.
```

Código C.50: ScratchProcess»applyPrimitive

```
evaluateWithArgs: rawArgs
    "Evalue this block with the given argument list."

    | args |
    "special case for math and boolean infix operators"
    selector isInfix ifTrue: [^ self evaluateInfixWithArgs: rawArgs].

    args _ self coerceArgs: rawArgs..

    "special case for unary operators"
    (#(abs not rounded sqrt truncated) includes: selector)
        ifTrue: [^ args first perform: selector].

    ^ receiver perform: selector withArguments: args
```

Código C.51: CommandBlockMorph≫evaluateWithArgs:

```
evaluateWithArgs: rawArgs
...
    #userBlockWarn = selector ifTrue: [
        self showError.
        self stop.
        self userBlockWarn.
        ^ #stop].
...
```

Código C.52: CommandBlockMorph≫evaluateWithArgs:

```
applyPrimitive

    | value |
    value _ stackFrame expression evaluateWithArgs: stackFrame arguments.

    value = #stop ifTrue: [self errorFlag: true. ^ self doReturn].
...
```

Código C.53: ScratchProcess≫applyPrimitive

```
makeNewSpriteButtons: aScratchFrameMorph
   "Return a morph containing a set of new sprite buttons."
...
    buttonSpecs _ #(
        "    icon name              selector                    tooltip"
        (newSpritePaint          paintSpriteMorph         'Paint new sprite')
        (newSpriteLibrary        addSpriteMorph           'Choose new sprite from
            file')
        (newSpriteSurprise       surpriseSpriteMorph      'Get surprise sprite')
    ).
...
```

Código C.54: ScratchLibraryMorph≫makeNewSpriteButtons:

```
importSpriteOrProject: fileNameOrData
    "Read the sprite or project file and merge into the current project."
...
    importedStage submorphs do: [:m |
        (m isKindOf: ScratchSpriteMorph) ifTrue: [
                        ...
            self addAndView: m.  "assigns a new name"
...
```

Código C.55: ScratchFrameMorph»importSpriteOrProject:

```
addAndView: aSpriteMorph
    "Add given morph to the work pane and view it."

    | pos i p |
    aSpriteMorph center: workPane center.
    pos _ self scratchObjects collect: [:o | o referencePosition].
    i _ 0.
    [pos includes: (p _ (10 * i) asPoint)] whileTrue: [i _ i + 1].
    workPane addMorphFront: aSpriteMorph.
    aSpriteMorph objName: aSpriteMorph nextInstanceName.
    aSpriteMorph referencePosition: p.
    aSpriteMorph startStepping.
    workPane sprites addLast: aSpriteMorph.
    self view: aSpriteMorph tab: 'Scripts' category: 'motion'.
```

Código C.56: Versão inicial do método ScratchFrameMorph»addAndView:

```
addAndView: aSpriteMorph
    "Add given morph to the work pane and view it."

    | pos i p oldSprite globalDefs |
...
    aSpriteMorph startStepping.

    aSpriteMorph blocksBin allMorphsDo: [:m |
        (m isKindOf: SetterBlockMorph) ifTrue: [m receiver: aSpriteMorph]].
    aSpriteMorph testBlocksBin allMorphsDo: [:m |
        (m isKindOf: SetterBlockMorph) ifTrue: [m receiver: aSpriteMorph]].

    "add existing global scripts to the new sprite"
    (oldSprite _ workPane sprites first) ifNotNil: [
        globalDefs _ oldSprite userBlocks values select: [:def | def isGlobal].
        globalDefs do: [:def |
            (aSpriteMorph existsBlock: def spec) ifFalse: [
                aSpriteMorph createUserBlockWithDefinition: def.
            ]
        ]
    ].
    "add new sprite's global scripts to existing sprites"
    globalDefs _ aSpriteMorph userBlocks values select: [:def | def isGlobal].
    workPane sprites do: [:sp |
        globalDefs do: [:def | sp createUserBlockWithDefinition: def]
    ].

    workPane sprites addLast: aSpriteMorph.
    self view: aSpriteMorph tab: 'Scripts' category: 'motion'.
```

Código C.57: Versão final do método ScratchFrameMorph»addAndView:

```
rightButtonMenu

    | menu choice spec |
    menu _ CustomMenu new.
...
    (owner isKindOf: ScratchBlockPaletteMorph) ifFalse: [
        ...
    ].
    menu addLine.
    menu add: 'show definition' action: #showDefinition.
...
    (choice _ menu localize; startUp) ifNil: [^ self].
    (#(presentHelpScreen duplicate delete saveScript showDefinition) includes:
        choice)
        ifTrue: [^ self perform: choice]
...
```

Código C.58: UserCommandBlockMorph»rightButtonMenu

```
showDefinition
    "Shows the body (definition) of a user-defined block"

    | rcvr body sBlock editor |
    rcvr _ self receiver.
    body _ (rcvr definitionOfUserBlock: (self commandSpec)) body deepCopy.
    editor _ (self ownerThatIsA: ScratchFrameMorph) findA:
        ScratchScriptEditorMorph.
    editor tabPane currentTab: 'Test'.
    sBlock _ rcvr scriptBlock.
    (sBlock ownerThatIsA: ScratchScriptsMorph) submorphsDo: [:subm |
        (subm isKindOf: HatBlockMorph) ifFalse: [subm delete]]. "delete free
            stacks"
    sBlock nextBlock ifNotNil: [sBlock nextBlock delete]. "delete any attached
        stack"
    sBlock nextBlock: body.
    rcvr inspectedBlock: self commandSpec.
    sBlock addBlockName: self commandSpec.
```

Código C.59: UserCommandBlockMorph»showDefinition

```
addBlockName: blockName
    "Adds string 'run <blockName>' to blue flag hat. In case blockName is nil,
    it puts the default block label 'when [blue flag] clicked'."

    self submorphsDo: [:subm |
        ((subm isKindOf: StringMorph) and: [subm contents ~= 'when' and:
            [subm contents ~= 'clicked']])
                ifTrue: [subm delete]].
    (blockName ~= nil) ifTrue: [
        self addMorphBack: (StringMorph new contents: 'run ' , blockName
            asMacRoman;
            font: (ScratchFrameMorph getFont: #Label); color: Color white)].
```

Código C.60: EventHatMorph»addBlockName:

```
rightButtonMenu
...
    (owner isKindOf: ScratchBlockPaletteMorph) ifFalse: [
        ...
        currentCategory _ (self ownerThatIsA: ScratchScriptEditorMorph)
            currentCategory.
        (currentCategory = 'Scripts') ifTrue: [
            menu addLine.
            menu add: 'save script' action: #saveScript].
        (currentCategory = 'Test') ifTrue: [
            menu addLine.
            self receiver inspectedBlock ifNotNil: [
                menu add: 'save script' action: #updateBlockDefinition].
            menu add: 'save script as...' action: #saveScriptAs]].
...
```

Código C.61: HatBlockMorph»rightButtonMenu

```
rightButtonMenu

    (owner isKindOf: ScratchBlockPaletteMorph) ifFalse: [
        ...
        currentCategory _ (self ownerThatIsA: ScratchScriptEditorMorph)
            currentCategory.
        ((currentCategory = 'Scripts') or: [currentCategory = 'Test' and:
            [(self topBlock isKindOf: EventHatMorph) not]]) ifTrue: [
                menu addLine.
                menu add: 'save script' action: #saveScript
            ].
        ((currentCategory = 'Test') and: [self topBlock isKindOf: EventHatMorph])
            ifTrue: [
                menu addLine.
                tb _ ((self ownerThatIsA: ScratchFrameMorph) libraryPane
                    spriteThumbnails detect: [:sp | sp isSelected] ifNone: [nil]).
                tb
                    ifNotNil: [rcvr _ tb target]
                    ifNil: [rcvr _ self receiver].
                rcvr inspectedBlock ifNotNil: [
                    menu add: 'save script' action: #updateBlockDefinition].
                menu add: 'save script as...' action: #saveScriptAs]
    ].
...
    (choice _ menu localize; startUp) ifNil: [^ self].
    (#(presentHelpScreen duplicate delete saveScript updateBlockDefinition
        saveScriptAs) includes: choice) ifTrue: [^ self perform: choice].
...
```

Código C.62: CommandBlockMorph»rightButtonMenu

```
updateBlockDefinition

    ^ self subclassResponsibility.
```

Código C.63: BlockMorph»updateBlockDefinition

```
updateBlockDefinition

    | body def isGlobal |
    self nextBlock ifNil: [
        self beep.
        DialogBoxMorph warn: 'No script found.'.
        ^ nil]
    ifNotNil: [body _ self nextBlock].
    def _ self receiver definitionOfUserBlock: self receiver inspectedBlock.
    isGlobal _ def isGlobal.
    def body: body.
    self receiver createUserBlockWithDefinition: def global: isGlobal.
    isGlobal ifTrue: [self receiver updateTestTabScript: body].
```

Código C.64: HatBlockMorph»updateBlockDefinition

```
updateBlockDefinition

    | body def isGlobal tb |
    body _ self topBlock nextBlock.
    tb _ ((self ownerThatIsA: ScratchFrameMorph) libraryPane spriteThumbnails
        detect: [:sp | sp isSelected] ifNone: [nil]).
    tb ifNotNil: [self receiver: tb target].
    def _ self receiver definitionOfUserBlock: self receiver inspectedBlock.
    isGlobal _ def isGlobal.
    def body: body.
    self receiver createUserBlockWithDefinition: def global: isGlobal.
    isGlobal ifTrue: [self receiver updateTestTabScript: body].
```

Código C.65: CommandBlockMorph»updateBlockDefinition

```
updateTestTabScript: aBlock

    | stage |
    stage _ self ownerThatIsA: ScratchStageMorph.
    stage sprites do: [:sprite |
        (sprite ~~ self and: [sprite inspectedBlock = self inspectedBlock])
            ifTrue: [
                sprite scriptBlock nextBlock ifNotNil:
                    [sprite scriptBlock nextBlock delete].
                sprite scriptBlock nextBlock: aBlock deepCopy.
            ]
        ]
```

Código C.66: ScriptableScratchMorph»updateTestTabScript:

```
updateBlockDefinition

    ^ self owner updateBlockDefinition
```

Código C.67: ReporterBlockMorph»updateBlockDefinition

```
saveScriptAs

    | firstBlock |
    (firstBlock _ self nextBlock) ifNil: [
        self beep.
        DialogBoxMorph warn: 'No script found.'.
        ^ nil]
    ifNotNil: [self receiver saveBlockWithBody: firstBlock].
```

Código C.68: HatBlockMorph»saveScriptAs

```
saveScriptAs

    | body tb |
    body _ self topBlock nextBlock.
    tb _ ((self ownerThatIsA: ScratchFrameMorph) libraryPane spriteThumbnails
        detect: [:sp | sp isSelected] ifNone: [nil]).
    tb ifNotNil: [self receiver: tb target].
    self receiver saveBlockWithBody: body.
```

Código C.69: CommandBlockMorph»saveScriptAs

```
saveBlockWithBody: body
...
    self createUserBlockWithDefinition: def global: isGlobal.
    self inspectedBlock: blockName.
    self scriptBlock addBlockName: blockName.
...
```

Código C.70: ScriptableScratchMorph»saveBlockWithBody:

```
saveScriptAs

    ^ self owner saveScriptAs
```

Código C.71: ReporterBlockMorph»saveScriptAs

```
scriptsMenu: aPosition
    "Present a menu of Scratch script operations."

    | menu choice |
    ...
    menu _ CustomMenu new.
    ...
    (self tabPane currentTab = 'Test') ifTrue: [
        menu add: 'clear' action: #clear].
...
```

Código C.72: ScratchScriptEditorMorph»scriptsMenu:

```
clear
    "Deletes all blocks from Test tab, except the hat, and returns the label of
     its hat block to its default value."

    | hat |
    hat _ self target scriptBlock.
    (hat ownerThatIsA: ScratchScriptsMorph) submorphsDo: [:m |
        (m isKindOf: HatBlockMorph) ifFalse: [m delete]].
    hat nextBlock ifNotNil: [hat nextBlock delete].
    hat receiver inspectedBlock: nil.
    hat receiver scriptBlock addBlockName: nil.
```

Código C.73: ScratchScriptEditorMorph»clear

```
rightButtonMenu

    | menu choice spec |
    menu _ CustomMenu new.
...
    (owner isKindOf: ScratchBlockPaletteMorph) ifFalse: [
        ...
    ]
    ifTrue: [
        menu addLine.
        menu add: 'delete script' action: #deleteScript.
    ].
...
    (choice _ menu localize; startUp) ifNil: [^ self].
    (#(presentHelpScreen duplicate delete saveScript showDefinition deleteScript
        updateBlockDefinition saveScriptAs) includes: choice)
            ifTrue: [^ self perform: choice].
...
```

Código C.74: UserCommandBlockMorph»rightButtonMenu

```
deleteScript
    "Permanently deletes this script from its scriptable object.
    If the script is global to all sprites then it is deleted from all of them."

    | scriptable isGlobal frame spec response |
    response _ DialogBoxMorph ask: 'Are you sure you want to delete this script?'.
    response = #cancelled ifTrue: [^ nil].
    response ifTrue: [
        frame _ self ownerThatIsA: ScratchFrameMorph.
        scriptable _ self receiver.
        spec _ self commandSpec.
        isGlobal _ (scriptable definitionOfUserBlock: spec) isGlobal.
        scriptable removeUserBlock: spec.
        scriptable deleteUserBlock: spec.
        isGlobal ifTrue: [
            scriptable owner sprites do: [:sprite |
                (sprite ~~ scriptable) ifTrue: [
                    sprite removeUserBlock: spec.
                    sprite deleteUserBlock: spec.
                ].
                (sprite inspectedBlock = spec) ifTrue: [
                    sprite inspectedBlock: nil.
                    sprite scriptBlock addBlockName: nil.
                ].
            ]
        ] ifFalse: [
            scriptable inspectedBlock: nil.
            scriptable scriptBlock addBlockName: nil.
        ].
        frame viewerPane currentCategory: 'MyScripts'].
```

Código C.75: UserCommandBlockMorph»deleteScript

```
removeUserBlock: aSpec

    self userBlocks removeKey: aSpec ifAbsent: [^ nil].
```

Código C.76: ScriptableScratchMorph»removeUserBlock:

```
deleteUserBlock: spec
    "Deletes all instances of the user-defined block with spec <spec>
    from the Scripts and Test tabs"

    self blocksBin deleteUserBlock: spec.
    self testBlocksBin deleteUserBlock: spec.
```

Código C.77: ScriptableScratchMorph»deleteUserBlock:

```
fileMenu: aMenuTitleMorph

    | menu |
    menu _ CustomMenu new.
    menu add: 'New' action: #newScratchProject.
    menu add: 'Open' action: #openScratchProject.
    menu add: 'Save' action: #saveScratchProjectNoDialog.
    menu add: 'Save As' action: #saveScratchProject.
    menu addLine.
    menu add: 'Import Project' action: #importScratchProject.
    menu add: 'Export Sprite' action: #exportSprite.
    menu addLine.
    menu add: 'Project Notes' action: #editNotes.
...
```

Código C.78: ScratchFrameMorph»fileMenu:

```
saveScratchProjectNoDialog

        | fName dir |
        self closeMediaEditorsAndDialogs ifFalse: [^ self].

        projectName ifNil: [projectName _ ''].
        fName _ self nameFromFileName: projectName.

        dir _ ScratchFileChooserDialog getLastFolderForType: #project.
        (fName size = 0 | (dir fileExists: fName , '.sb') not) ifTrue: [^ self
            saveScratchProject].
        ScratchFileChooserDialog lastFolderIsSampleProjectsFolder ifTrue: [^ self
             saveScratchProject].

        self updateLastHistoryEntryIfNeeded.

        projectName _ FileDirectory localNameFor: (fName, '.sb'). "ignore path,
            if any; save in the original project directory"
        projectDirectory _ dir.

        self updateHistoryProjectName: projectName op: 'save'.
        self writeScratchProject.
```

Código C.79: ScratchFrameMorph»saveScratchProjectNoDialog

```
saveScratchProject

        | fName result |
        self closeMediaEditorsAndDialogs ifFalse: [^ self].
        self stopAll.

        fName _ ScratchFileChooserDialog saveScratchFileFor: self.
        (fName size = 0 or: [fName = #cancelled]) ifTrue: [^ self].

        [(result _ ScratchFileChooserDialog confirmFileOverwriteIfExisting: fName)
             = false] whileTrue: [
                fName _ ScratchFileChooserDialog saveScratchFileFor: self.
                (fName size = 0 or: [fName = #cancelled]) ifTrue: [^ self]].
        (result = #cancelled) ifTrue: [^ self].

        self updateLastHistoryEntryIfNeeded.

        fName _ (self nameFromFileName: fName), '.sb'.
        projectDirectory _ FileDirectory on: (FileDirectory dirPathFor: fName).
        projectName _ FileDirectory localNameFor: fName.

        projectInfo at: 'author' put: author.
        self updateHistoryProjectName: projectName op: 'save'.
        self writeScratchProject.
```

Código C.80: ScratchFrameMorph≫saveScratchProject

```
writeScratchProject
...
        workPane allMorphsDo: [:m |
                (m isKindOf: ScriptableScratchMorph) ifTrue: [
                    m blocksBin allMorphsDo: [:b |
                        (b isKindOf: BlockMorph) ifTrue: [b stop]].
                    m convertStacksToTuples]].
...
```

Código C.81: ScratchFrameMorph≫writeScratchProject

```
convertStacksToTuples
        "Convert my blocks bin from a morph containing block stack into a
            collection of (<point>, <tuple>) pairs the represent the same stacks
            in compact, language-independent form."

        | stacks blocks comments |

        (blocksBin isKindOf: Array) ifTrue: [^ self]. "already converted"

        stacks _ (blocksBin submorphs select: [:m | m respondsTo: #tupleSequence])
            .
        blocks _ stacks select: [:m | m isKindOf: BlockMorph].
        comments _ stacks select: [:m | m isKindOf: ScratchCommentMorph].

        blocks _ blocks collect: [:blockM |
                Array
                        with: blockM position - blocksBin position
                        with: blockM tupleSequence].

        comments _ comments collect: [:blockM |
                Array
                        with: blockM position - blocksBin position
                        with: blockM tupleSequence].

        blocksBin _ blocks, comments.
```

Código C.82: ScriptableScratchMorph»convertStacksToTuples

```
convertTestStacksToTuples
        "Convert my test blocks bin from a morph containing block stack into a
            collection of (<point>, <tuple>) pairs the represent the same stacks
            in compact, language-independent form."

        | stacks blocks comments |

        (testBlocksBin isKindOf: Array) ifTrue: [^ self]. "already converted"

        stacks _ (testBlocksBin submorphs select: [:m | m respondsTo: #
            tupleSequence]).
        blocks _ stacks select: [:m | m isKindOf: BlockMorph].
        comments _ stacks select: [:m | m isKindOf: ScratchCommentMorph].

        blocks _ blocks collect: [:blockM |
                Array
                        with: blockM position - testBlocksBin position
                        with: blockM tupleSequence].

        comments _ comments collect: [:blockM |
                Array
                        with: blockM position - testBlocksBin position
                        with: blockM tupleSequence].

        testBlocksBin _ blocks, comments.
```

Código C.83: ScriptableScratchMorph»convertTestStacksToTuples

```
writeScratchProject
...
        workPane allMorphsDo: [:m |
                (m isKindOf: ScriptableScratchMorph) ifTrue: [
                    m blocksBin allMorphsDo: [:b |
                        (b isKindOf: BlockMorph) ifTrue: [b stop]].
                    m testBlocksBin allMorphsDo: [:b |
                        (b isKindOf: BlockMorph) ifTrue: [b stop]].
                    m convertStacksToTuples.
                    m convertTestStacksToTuples]].
...
```

Código C.84: ScratchFrameMorph»writeScratchProject

```
writeScratchProject
...
        workPane allMorphsDo: [:m |
                (m isKindOf: ScriptableScratchMorph) ifTrue: [
                    m convertTuplesToStacks]].
...
```

Código C.85: ScratchFrameMorph»writeScratchProject

```
convertTuplesToStacks
        "Convert my blocks bin from a collection of (<point>, <tuple>) pairs into
            a morph containing a number of block stacks."

        | tuplesList stack |
        (blocksBin isKindOf: Array) ifFalse: [^ self].  "already converted"
        tuplesList _ blocksBin.
        blocksBin _ ScratchScriptsMorph new.
        tuplesList do: [:pair |
                stack _ self stackFromTupleList: pair second receiver: self.
                stack position: pair first.
                blocksBin addMorph: stack].
```

Código C.86: ScriptableScratchMorph»convertTuplesToStacks

```
stackFromTupleList: tupleList receiver: scriptOwner
    "Answer a new block stack from the given sequence of tuples."
    "self stackFromTupleList: #() receiver: nil"
...
    tupleList do: [:tuple |
        block _ self blockFromTuple: tuple receiver: scriptOwner.
...
```

Código C.87: ScriptableScratchMorph»stackFromTupleList:receiver:

```
blockFromTuple: tuple receiver: scriptOwner
    "Answer a new block for the given tuple."
...
    (#(EventHatMorph KeyEventHatMorph MouseClickEventHatMorph WhenHatBlockMorph)
       includes: k) ifTrue: [
      block _ self hatBlockFromTuple: tuple receiver: scriptOwner.
      (block isKindOf: WhenHatBlockMorph) ifTrue: [block color: Color red].
      ^ block].
...
```

Código C.88: ScriptableScratchMorph»blockFromTuple:receiver:

```
hatBlockFromTuple: tuple receiver: scriptOwner
    "Answer a new block for the given variable reference tuple."

    | blockClass block eventName arg |
    blockClass _ Smalltalk at: tuple first.
    block _ blockClass new scriptOwner: scriptOwner.

    blockClass = EventHatMorph ifTrue: [
        eventName _ tuple at: 2.
        eventName = 'Scratch-StartClicked'
            ifTrue: [block forStartEvent; scriptOwner: scriptOwner]
            ifFalse: [eventName = 'Scratch-StartScriptClicked'
                ifTrue: [block forStartScriptEvent; scriptOwner: scriptOwner]
                ifFalse: [block eventName: eventName]]].
...
```

Código C.89: ScriptableScratchMorph»hatBlockFromTuple:receiver:

```
convertTestTuplesToStacks
        "Convert my test blocks bin from a collection of (<point>, <tuple>) pairs
            into a morph containing a number of block stacks."

        | tuplesList stack |
        (testBlocksBin isKindOf: Array) ifFalse: [^ self]. "already converted"
        tuplesList _ testBlocksBin.
        testBlocksBin _ ScratchScriptsMorph new.
        tuplesList do: [:pair |
                stack _ self testStackFromTupleList: pair second receiver: self.
                stack position: pair first.
                testBlocksBin addMorph: stack].
```

Código C.90: ScriptableScratchMorph»convertTestTuplesToStacks

```
testStackFromTupleList: tupleList receiver: scriptOwner
...
    tupleList do: [:tuple |
        block _ self testBlockFromTuple: tuple receiver: scriptOwner.
...
```

Código C.91: ScriptableScratchMorph»testStackFromTupleList:receiver:

```
testBlockFromTuple: tuple receiver: scriptOwner
...
   #scratchComment = k ifTrue: [
        block _ ScratchCommentMorph new.
        tuple size > 1 ifTrue: [block commentMorph contents: (tuple at: 2)].
        tuple size > 2 ifTrue: [(tuple at: 3) ifFalse: [block toggleShowing]].
        tuple size > 3 ifTrue: [block width: (tuple at: 4)].
        tuple size > 4 ifTrue: [block anchor: (self testBlockWithID: (tuple at: 5)
            )].
        ^ block].
...
```

Código C.92: ScriptableScratchMorph»testBlockFromTuple:receiver:

```
blockFromTuple: tuple receiver: scriptOwner
...
        #scratchComment = k ifTrue: [
                block _ ScratchCommentMorph new.
                tuple size > 1 ifTrue: [block commentMorph contents: (tuple at: 2)
                    ].
                tuple size > 2 ifTrue: [(tuple at: 3) ifFalse: [block
                    toggleShowing]].
                tuple size > 3 ifTrue: [block width: (tuple at: 4)].
                tuple size > 4 ifTrue: [block anchor: (self blockWithID: (tuple at
                    : 5))].
                ^ block].
...
```

Código C.93: ScriptableScratchMorph»blockFromTuple:receiver:

```
blockWithID: id

    | topBlockList blockList|

    topBlockList _ (blocksBin submorphs select: [:m |
        (m isKindOf: BlockMorph) ]) reversed.
    blockList _ OrderedCollection new.
    topBlockList do: [:top | (top allMorphs select: [:b| b isKindOf: BlockMorph ])
        do: [:m | blockList add: m ]].

    ^ blockList at: id
```

Código C.94: ScriptableScratchMorph»blockWithID:

```
testBlockWithID: id

    | topBlockList blockList|

    topBlockList _ (testBlocksBin submorphs select: [:m |
        (m isKindOf: BlockMorph) ]) reversed.
    blockList _ OrderedCollection new.
    topBlockList do: [:top | (top allMorphs select: [:b| b isKindOf: BlockMorph ])
        do: [:m | blockList add: m ]].

    ^ blockList at: id
```

Código C.95: ScriptableScratchMorph»testBlockWithID:

```
writeScratchProject
...
        workPane allMorphsDo: [:m |
                (m isKindOf: ScriptableScratchMorph) ifTrue: [
                    m convertTuplesToStacks.
                    m convertTestTuplesToStacks]].
...
```

Código C.96: ScratchFrameMorph»writeScratchProject

```
projectIsEmpty
...
        allScriptables do: [:m |
                m blocksBin submorphs size > 0 ifTrue: [^ false]. "any stacks?"
                m testBlocksBin submorphs size > 0 ifTrue: [^ false].
...
```

Código C.97: ScratchFrameMorph»projectIsEmpty

```
extractProjectFrom: aByteArray
        "Answer a Scratch project (i.e. a ScratchStageMorph possibly containing
            sprites) from the given ByteArray. Answer nil if the project cannot be
             unpacked."

        | s version proj |
        s _ ReadStream on: aByteArray.
        version _ ObjStream scratchFileVersionFrom: (s next: 10) asString.
        version = 0 ifTrue: [
                s position: 0.
                proj _ ObjStream new readObjFrom: s showProgress: true].
        (version = 1) | (version = 2) ifTrue: [
                s skip: s uint32. "skip header"
                proj _ ObjStream new readObjFrom: s showProgress: true].

        proj class = ScratchStageMorph ifFalse: [
                version > 2
                        ifTrue: [self error: 'Project created by a later version
                            of Scratch']
                        ifFalse: [self error: 'Problem reading project.'].
                ^ nil].

        ScriptableScratchMorph buildBlockSpecDictionary.
        proj allMorphsDo: [:m |
                (m isKindOf: ScriptableScratchMorph) ifTrue: [ "covert to new
                    blocks"
                        m convertStacksToTuples.
                        m convertTuplesToStacks]].

        ^ proj
```

Código C.98: ScratchFrameMorph»extractProjectFrom:

```
extractProjectFrom: aByteArray
...
        proj allMorphsDo: [:m |
                (m isKindOf: ScriptableScratchMorph) ifTrue: [ "covert to new
                    blocks"
                    m convertStacksToTuples.
                    m convertTestStacksToTuples.
                    m convertTuplesToStacks.
                    m convertTestTuplesToStacks]].
...
```

Código C.99: ScratchFrameMorph»extractProjectFrom:

```
writeObject: anObject objEntry: objEntry
    "Write the object described by the given entry."

    | entry classID putSelector |
    entry _ self classEntryFor: anObject.
    classID _ entry at: 1.
    putSelector _ entry at: 4.

    fields _ objEntry at: 4.
    fieldIndex _ 0.

    putSelector = #putUserObj:id: ifTrue: [
        stream nextPut: classID.
        stream nextPut: anObject fieldsVersion.
        stream nextPut: fields size].

    self perform: putSelector with: anObject with: classID.
```

Código C.100: ObjStream»writeObject:objEntry:

```
initialize
...
    self userClasses do: [:pair |
        entry _ pair, #(unused putUserObj:id:).
...
```

Código C.101: ObjStream»initialize

```
userClasses
    "Answer an array of (<class id>, <class name>) records for all version
        numbered
    user classes."
...
        (124            ScratchSpriteMorph)
        (125            ScratchStageMorph)
...
```

Código C.102: ObjStream»userClasses

```
putUserObj: anObject id: ignored
    "Ask the given user-defined object to write its fields."

    anObject storeFieldsOn: self.
```

Código C.103: ObjStream»putUserObj:id:

```
readObjectRecord
    "Read the next object record. Answer an array of the form
    (<obj> <class ID> [<version> <fieldsArray>]). The version and fields
    array are supplied only for user-defined objects."

    | classID obj classVersion fieldCount fieldList readSelector |
    classID _ stream next.
    classID > ObjectReferenceID
        ifTrue: [ "user defined object"
            obj _ (self classForID: classID) new.
            classVersion _ stream next.
            fieldCount _ stream next.
            fieldList _ (1 to: fieldCount) collect: [:i | self readField].
            ^ Array with: obj with: classID with: classVersion with: fieldList]
        ifFalse: [ "fixed format object"
            readSelector _ (IDToClassEntry at: classID) at: 3.
            obj _ self perform: readSelector with: nil with: classID.
            ^ Array with: obj with: classID].
```

Código C.104: ObjStream≫readObjectRecord

```
initializeUserDefinedFields: objectRecord
    "If the given object record designates a user-defined object, ask that object
    to initialiize itself from its fields list. Otherwise, do nothing."

    | obj classID classVersion |
    obj _ objectRecord at: 1.
    classID _ objectRecord at: 2.
    classID > ObjectReferenceID ifTrue: [ "user defined class"
        classVersion _ objectRecord at: 3.
        fields _ objectRecord at: 4.
        fieldIndex _ 0.
        obj initFieldsFrom: self version: classVersion].
```

Código C.105: ObjStream≫initializeUserDefinedFields:

```
initFieldsFrom: anObjStream version: classVersion

    super initFieldsFrom: anObjStream version: classVersion.
    self initFieldsNamed: #(
        zoom
        hPan
        vPan
    ) from: anObjStream.
    classVersion = 1 ifTrue: [^ self].

    "fields added in version 2"
    self initFieldsNamed: #(
        obsoleteSavedState
    ) from: anObjStream.
    classVersion = 2 ifTrue: [^ self].

    "fields added in version 3"
    self initFieldsNamed: #(
        sprites
    ) from: anObjStream.
    classVersion = 3 ifTrue: [^ self].

    "fields added in version 4"
    self initFieldsNamed: #(
        volume
        tempoBPM
    ) from: anObjStream.
    classVersion = 4 ifTrue: [^ self].

    "fields added in version 5"
    self initFieldsNamed: #(
        sceneStates
        lists
    ) from: anObjStream.
    lists ifNil: [lists _ Dictionary new].  "work around"
```

Código C.106: ScratchStageMorph»initFieldsFrom:version:

```
initFieldsFrom: anObjStream version: classVersion

    super initFieldsFrom: anObjStream version: classVersion.
...
    "fields added in version 3"
    self initFieldsNamed: #(
        sceneStates
        lists
    ) from: anObjStream.
    lists ifNil: [lists _ Dictionary new].  "work around"
```

Código C.107: ScratchSpriteMorph»initFieldsFrom:version:

```
fieldsVersion

    ^ 4
```

Código C.108: ScratchSpriteMorph»fieldsVersion

```
initFieldsFrom: anObjStream version: classVersion

    super initFieldsFrom: anObjStream version: classVersion.
...
    "fields added in version 3"
    self initFieldsNamed: #(
        sceneStates
        lists
    ) from: anObjStream.
    lists ifNil: [lists _ Dictionary new].  "work around"
    classVersion = 3 ifTrue: [^ self].

    "fields added in version 4"
    self initFieldsNamed: #(
        testBlocksBin
    ) from: anObjStream.
```

Código C.109: ScratchSpriteMorph»initFieldsFrom:version:

```
fieldsVersion

    ^ 6
```

Código C.110: ScratchStageMorph»fieldsVersion

```
initFieldsFrom: anObjStream version: classVersion

    super initFieldsFrom: anObjStream version: classVersion.
...
    "fields added in version 5"
    self initFieldsNamed: #(
        sceneStates
        lists
    ) from: anObjStream.
    lists ifNil: [lists _ Dictionary new].  "work around"
    classVersion = 5 ifTrue: [^ self].

    "fields added in version 6"
    self initFieldsNamed: #(
        testBlocksBin
    ) from: anObjStream.
```

Código C.111: ScratchStageMorph»initFieldsFrom:version:

```
storeFieldsOn: anObjStream

    super storeFieldsOn: anObjStream.
    self storeFieldsNamed: #(
        zoom
        hPan
        vPan
        obsoleteSavedState
        sprites
        volume
        tempoBPM
        sceneStates
        lists
        testBlocksBin
    ) on: anObjStream.
```

Código C.112: ScratchStageMorph»storeFieldsOn:

```
storeFieldsOn: anObjStream
...
    super storeFieldsOn: anObjStream.
    self storeFieldsNamed: #(
        visibility
        scalePoint
        rotationDegrees
        rotationStyle
        volume
        tempoBPM
        draggable
        sceneStates
        lists
        testBlocksBin
    ) on: anObjStream.
...
```

Código C.113: ScratchSpriteMorph»storeFieldsOn:

```
storeFieldsOn: anObjStream

        | oldBlockBinOwner oldTestBlocksBinOwner |
        super storeFieldsOn: anObjStream.
        (blocksBin isKindOf: Morph) ifTrue: [
                oldBlockBinOwner _ blocksBin owner.
                blocksBin delete].
        (testBlocksBin isKindOf: Morph) ifTrue: [
                oldTestBlocksBinOwner _ testBlocksBin owner.
                testBlocksBin delete].
...
        oldBlockBinOwner ifNotNil: [oldBlockBinOwner addMorph: blocksBin].
        oldTestBlocksBinOwner ifNotNil: [oldTestBlocksBinOwner addMorph:
            testBlocksBin].
```

Código C.114: ScriptableScratchMorph»storeFieldsOn:

```
storeFieldsOn: anObjStream

    super storeFieldsOn: anObjStream.
    self storeFieldsNamed: #(
        zoom
        hPan
        vPan
        obsoleteSavedState
        sprites
        volume
        tempoBPM
        sceneStates
        lists
        testBlocksBin
        userBlocks
        inspectedBlock
    ) on: anObjStream.
```

Código C.115: ScratchStageMorph»storeFieldsOn:

```
initFieldsFrom: anObjStream version: classVersion
...
    "fields added in version 6"
    self initFieldsNamed: #(
        testBlocksBin
        userBlocks
        inspectedBlock
    ) from: anObjStream.
```

Código C.116: ScratchStageMorph»initFieldsFrom:version:

```
storeFieldsOn: anObjStream
...
    super storeFieldsOn: anObjStream.
    self storeFieldsNamed: #(
        visibility
        scalePoint
        rotationDegrees
        rotationStyle
        volume
        tempoBPM
        draggable
        sceneStates
        lists
        testBlocksBin
        userBlocks
        inspectedBlock
    ) on: anObjStream.
...
```

Código C.117: ScratchSpriteMorph»storeFieldsOn:

```
initFieldsFrom: anObjStream version: classVersion
...
    "fields added in version 4"
    self initFieldsNamed: #(
        testBlocksBin
        userBlocks
        inspectedBlock
    ) from: anObjStream.
```

Código C.118: ScratchSpriteMorph»initFieldsFrom:version:

```
fieldsVersion

    ^ 1
```

Código C.119: UserBlockDefinition»fieldsVersion

```
initFieldsFrom: anObjStream version: classVersion

    self initFieldsNamed: #(
        spec
        body
        isGlobal
    ) from: anObjStream.
```

Código C.120: UserBlockDefinition»initFieldsFrom:version:

```
storeFieldsOn: anObjStream

    self storeFieldsNamed: #(
        spec
        body
        isGlobal
    ) on: anObjStream.
```

Código C.121: UserBlockDefinition»storeFieldsOn:

```
initialize
    "self initialize"

    ObjectReferenceID _ 99.
    IDToClassEntry _ Dictionary new.
    NameToClassEntry _ Dictionary new.

    self fixedFormatClasses do: [:entry |
        (IDToClassEntry includesKey: entry first) ifTrue:
            [self error: 'duplicate fixed class ID'].
        IDToClassEntry at: entry first put: entry.
        NameToClassEntry at: entry second put: entry].

    self userClasses do: [:pair |
        entry _ pair, #(unused putUserObj:id:).
        (IDToClassEntry includesKey: entry first) ifTrue:
            [self error: 'duplicate user class ID'].
        IDToClassEntry at: entry first put: entry.
        NameToClassEntry at: entry second put: entry].

    FloatClassID _ (NameToClassEntry at: #Float) first.
    FirstPointerClassID _ (NameToClassEntry at: #Array) first.
```

Código C.122: ObjStream»initialize

```
userClasses
    "Answer an array of (<class id>, <class name>) records for all version
    numbered user classes."
    "The following finds obsolete user classes:"
    "self initialize.
    self userClasses reject: [:rec | Smalltalk includesKey: rec second]"

    ^ #(
        "id             class"
        (100        Morph)
        (101        BorderedMorph)
        (102        RectangleMorph)
        (103        EllipseMorph)
        (104        AlignmentMorph)
        (105        StringMorph)
...
        (174        WatcherSliderMorph)
        (175        ScratchListMorph)
        (176        ScrollingStringMorph)
        )
```

Código C.123: ObjStream»userClasses

```
fixedFormatClasses
    "Answer an array of records for fixed-format classes."

    ^ #(
        "id     class                   read selector          write selector"
        (1   UndefinedObject         getConst:id:           putConst:id:)
        (2   True                    getConst:id:           putConst:id:)
        (3   False                   getConst:id:           putConst:id:)
        (4   SmallInteger            getSmallInt:id:        putSmallInt:id:)
        (5   SmallInteger16          getSmallInt:id:        putSmallInt:id:)
            "optimization for ints that fit into 16 bits"
        (6   LargePositiveInteger    getBigInt:id:          putBigInt:id:)
        (7   LargeNegativeInteger    getBigInt:id:          putBigInt:id:)
        (8   Float                   getFloat:id:           putFloat:id:)
        (9   String                  getBytes:id:           putBytes:id:)
        (10 Symbol                   getBytes:id:           putBytes:id:)
        (11 ByteArray                getBytes:id:           putBytes:id:)
        (12     SoundBuffer              getSoundBuf:id:        putSoundBuf:id:)
        (13     Bitmap                   getBitmap:id:          putBitmap:id:)
        (14     UTF8                     getBytes:id:           putBytes:id:)
    "12-19 reserved for additional non-pointer objects"
        (20     Array                    getArray:id:           putArray:id:)
        (21     OrderedCollection        getCollection:id:      putCollection:id:)
        (22     Set                      getCollection:id:      putCollection:id:)
        (23     IdentitySet              getCollection:id:      putCollection:id:)
        (24     Dictionary               getDict:id:            putDict:id:)
        (25     IdentityDictionary       getDict:id:            putDict:id:)
    "26-29 reserved for additional collections"
        (30 Color                    getColor:id:           putColor:id:)
        (31     TranslucentColor         getColor:id:           putColor:id:)
...
    "99 reserved for object references"
    "100-255 reserved for user-defined classes"
    )
```

Código C.124: ObjStream≫fixedFormatClasses

```
userClasses
...
        (176        ScrollingStringMorph)

        (177        UserBlockDefinition)
    )
```

Código C.125: ObjStream≫userClasses

```
userClasses
...
        (177        UserBlockDefinition)
        (178        ChoiceOrExpressionArgMorph)
        (179        StringFieldMorph)
        (180        IfElseBlockMorph)
        (181        UserCommandBlockMorph)
        (182        AttributeArgMorph)
        (183        ImportCBlockMorph)
    )
```

Código C.126: ObjStream≫userClasses

```
fixedFormatClasses
...
        (14 UTF8                    getBytes:id:            putBytes:id:)
        (15 UTF32                   getBytes:id:            putBytes:id:)
...
```

Código C.127: ObjStream»fixedFormatClasses

```
getBytes: anObject id: classID

    | byteCount result |
    byteCount _ stream uint32.
    result _ stream next: byteCount.
    classID = 9 ifTrue: [^ result asString].
    classID = 10 ifTrue: [^ result asString asSymbol].
    classID = 14 ifTrue: [^ UTF8 withAll: result].
    classID = 15 ifTrue: [^ UTF32 withAll: result].
    ^ result
```

Código C.128: ObjStream»getBytes:id:

```
ObjStream initialize
```

Código C.129: Invocação do método de classe initialize da classe ObjStream.

```
asBlockTuple

    | blockTuple defTuple |
    defTuple _ (self receiver definitionOfUserBlock: (self commandSpec)) body
        tupleSequence.
    blockTuple _ Array with: #userBlockWarn with: (self commandSpec) with:
        defTuple.
    ^ blockTuple
```

Código C.130: UserCommandBlockMorph»asBlockTuple

```
blockFromTuple: tuple receiver: scriptOwner
    "Answer a new block for the given tuple."

    | k spec blockColor block argCount arg argBlock |
    k _ tuple first.
...
    #userBlockWarn = k ifTrue: [
        block _ UserCommandBlockMorph new
            receiver: scriptOwner;
            selector: k;
            commandSpec: tuple second;
            color: (self class blockColorFor: 'MyScripts').
            (scriptOwner definitionOfUserBlock: (tuple second)) body:
                (self stackFromTupleList: tuple last receiver: scriptOwner).
            ^ block].
...
```

Código C.131: ScriptableScratchMorph»blockFromTuple:receiver:

```
testBlockFromTuple: tuple receiver: scriptOwner
...
    #userBlockWarn = k ifTrue: [
        block _ UserCommandBlockMorph new
            receiver: scriptOwner;
            selector: k;
            commandSpec: tuple second;
            color: (self class blockColorFor: 'MyScripts').
            (scriptOwner definitionOfUserBlock: (tuple second)) body:
                (self testStackFromTupleList: tuple last receiver: scriptOwner).
            ^ block].
...
```

Código C.132: ScriptableScratchMorph»testBlockFromTuple:receiver:

```
writeScratchProject
...
    workPane allMorphsDo: [:m |
        (m isKindOf: ScriptableScratchMorph) ifTrue: [
            m convertTuplesToStacks.
            m convertTestTuplesToStacks.
            m scriptBlock: (m testBlocksBin submorphs detect: [:subm |
                subm isKindOf: EventHatMorph]).
            m scriptBlock addBlockName: m inspectedBlock]].
...
```

Código C.133: ScratchFrameMorph»writeScratchProject

```
extractProjectFrom: aByteArray
...
    proj allMorphsDo: [:m |
        (m isKindOf: ScriptableScratchMorph) ifTrue: [  "covert to new blocks"
            m convertStacksToTuples.
            m convertTestStacksToTuples.
            m convertTuplesToStacks.
            m convertTestTuplesToStacks.
            m scriptBlock: (m testBlocksBin submorphs detect: [:subm |
                subm isKindOf: EventHatMorph]).
            m scriptBlock addBlockName: m inspectedBlock.]].
...
```

Código C.134: ScratchFrameMorph»extractProjectFrom:

```
projectIsEmpty
    "Answer true if the current project has no scripts, no variables, no special
    costumes or sounds, and at most a single sprite."
...
    allScriptables do: [:m |
        m blocksBin submorphs size > 0 ifTrue: [^ false].  "any stacks?"
        m testBlocksBin submorphs size > 0 ifTrue: [^ false].
        m varNames size > 1 ifTrue: [^ false].  "any variables?"
        m userBlocks size > 0 ifTrue: [^ false].
...
```

Código C.135: ScratchFrameMorph»projectIsEmpty

```
createScratchFileChooserFor: aScratchFrameMorph saving: savingFlag
    "Create a Scratch file chooser dialog box with a project thumbnail and
    info box."
...
    list _ ScratchFilePicker new extensions: #(scratch sb extsb).
...
```

Código C.136: ScratchFileChooserDialog»createScratchFileChooserFor:saving:

```
importScratchProject
    "Allow the user to select a project to open, then merge that project's
    sprites with the current project."
...
    response _ ScratchFileChooserDialog
        chooseExistingFileType: #project
        extensions: #(scratch sb extsb)
        title: 'Import Project'.
...
```

Código C.137: ScratchFrameMorph»importScratchProject

```
confirmFileOverwriteIfExisting: aFilename
        "If the given file exists, ask the user if they want to overwrite it or
            pick
        a different file name."

    | response fName |
    fName _ aFilename.
    (fName endsWith: '.extsb') ifFalse: [fName _ fName, '.extsb'].
...
```

Código C.138: ScratchFileChooserDialog»confirmFileOverwriteIfExisting:

```
saveScratchFileFor: aScratchFrameMorph
    "Choose a file for saving the current Scratch project file.
    Display the thumbnail and info string for the current project and allow the
        info
    string to be edited. Answer the full name of the file in which to save the
        project
    or #cancelled if the operation is cancelled."
...
    (result asLowercase endsWith: '.extsb') ifFalse: [result _ result, '.extsb'].
...
```

Código C.139: ScratchFileChooserDialog»saveScratchFileFor:

```
nameFromFileName: fileName
    "Return the given Scratch file name without the trailing .sb or .scratch
    extension, if it has one. Ensure the the result is UTF8."

    | s |
    s _ fileName.
    (s asLowercase endsWith: '.scratch') ifTrue: [s _ s copyFrom: 1 to: s size -
        8].
    (s asLowercase endsWith: '.sb') ifTrue: [s _ s copyFrom: 1 to: s size - 3].
    (s asLowercase endsWith: '.extsb') ifTrue: [s _ s copyFrom: 1 to: s size - 6].
    s isUnicode ifFalse: [s _ UTF8 withAll: s].

    ^ s
```

Código C.140: ScratchFrameMorph»nameFromFileName:

```
processDroppedFiles
    "Process any files that have been dropped onto me."
...
    fName _ file fullName.
        ((fName asLowercase endsWith: '.scratch') |
        (fName asLowercase endsWith: '.sb') |
        (fName asLowercase endsWith: '.extsb'))
            ifTrue: [self openScratchDroppedProjectNamed: fName]
...
```

Código C.141: ScratchFrameMorph»processDroppedFiles

```
saveScratchProject
...
    fName _ (self nameFromFileName: fName), '.extsb'.
...
```

Código C.142: ScratchFrameMorph»saveScratchProject

```
saveScratchProjectNoDialog
...
    (fName size = 0 | (dir fileExists: fName, '.extsb') not) ifTrue: [^ self
        saveScratchProject].
...
    projectName _ FileDirectory localNameFor: (fName, '.extsb').
...
```

Código C.143: ScratchFrameMorph»saveScratchProjectNoDialog

```
startup
...
    fileName _ startupFileNames
        detect: [:fn |
            ((fn asLowercase endsWith: '.sb') or:
            [fn asLowercase endsWith: '.scratch']) or:
            [fn asLowercase endsWith: '.extsb']]
...
```

Código C.144: ScratchFrameMorph»startup

```
writeMultipleSummaries
    "Write the summary for all Scratch projects in a given folder."
...
    dir allFileNamesDo: [:fn |
        (fn asLowercase endsWith: '.scratch') |
        (fn asLowercase endsWith: '.sb') |
        (fn asLowercase endsWith: '.extsb') ifTrue: [
            self openScratchProjectNamed: fn.
            World doOneCycleNoInput.
            self writeSummaryFile: fn]].
...
```

Código C.145: ScratchFrameMorph»writeMultipleSummaries

```
copyForExport
    "Answer a copy of me for exporting."

    | objToExport |
    objToExport _ self fullCopy.
    objToExport objName: objName.
    objToExport blocksBin allMorphsDo: [:m |
        (m isKindOf: BlockMorph) ifTrue: [m stop].
        (m isKindOf: SpriteArgMorph) ifTrue: [m clearMorphReference]].
    objToExport convertStacksToTuples.
    ^ objToExport
```

Código C.146: ScriptableScratchMorph»copyForExport

```
copyForExport
    "Answer a copy of me for exporting."

    | objToExport |
    objToExport _ self fullCopy.
    objToExport objName: objName.
    objToExport blocksBin allMorphsDo: [:m |
        (m isKindOf: BlockMorph) ifTrue: [m stop].
        (m isKindOf: SpriteArgMorph) ifTrue: [m clearMorphReference]].
    objToExport testBlocksBin allMorphsDo: [:m |
        (m isKindOf: BlockMorph) ifTrue: [m stop].
        (m isKindOf: SpriteArgMorph) ifTrue: [m clearMorphReference]].
    objToExport convertStacksToTuples.
    objToExport convertTestStacksToTuples.
    ^ objToExport
```

Código C.147: ScriptableScratchMorph»copyForExport

```
importSpriteOrProject: fileNameOrData
        "Read the sprite or project file and merge into the current project."
...
        "fix references to old stage"
        importedStage allMorphsDo: [:m |
            (m isKindOf: WatcherMorph) ifTrue: [m mapReceiver: importedStage to:
                workPane].
            (m isKindOf: ScriptableScratchMorph) ifTrue: [
                m blocksBin submorphs do: [:stack |
                    (stack isKindOf: BlockMorph) ifTrue: [
                        stack blockSequence do:
                            [:b | b mapReceiver: importedStage to: workPane]]].
                m testBlocksBin submorphs do: [:stack |
                    (stack isKindOf: BlockMorph) ifTrue: [
                        stack blockSequence do:
                            [:b | b mapReceiver: importedStage to: workPane]]]]].
...
```

Código C.148: ScratchFrameMorph≫importSpriteOrProject:

```
importSpriteOrProject: fileNameOrData
...
        "add imported stage scripts"
            importedStage blocksBin submorphs do: [:stack |
                (stack isKindOf: BlockMorph) ifTrue: [workPane addStack: stack
                    fullCopy]].
...
```

Código C.149: ScratchFrameMorph≫importSpriteOrProject:

```
addStack: aBlockStack
    "Aligns the newly added script below the lowest script in the pane."

    | y bottom |
    y _ 10.
    blocksBin submorphsDo: [:m |
        bottom _ (m fullBounds bottom) - (blocksBin position y).
        (bottom > y) ifTrue:
            [y _ bottom]].
    aBlockStack position: blocksBin position + (20@(y+10)).

    aBlockStack newScriptOwner: self.
    blocksBin addMorph: aBlockStack.
```

Código C.150: ScriptableScratchMorph≫addStack:

```
importSpriteOrProject: fileNameOrData
...
        "add imported stage scripts"
            importedStage blocksBin submorphs do: [:stack |
                (stack isKindOf: BlockMorph) ifTrue: [workPane addStack: stack
                    fullCopy]].
            importedStage testBlocksBin submorphs do: [:stack |
                (stack isMemberOf: EventHatMorph) ifTrue: [
                    stack nextBlock ifNotNil: [
                        stack _ stack nextBlock. workPane addTestStack: stack
                            fullCopy]]
                ifFalse: [(stack isKindOf: BlockMorph) ifTrue: [
                    workPane addTestStack: stack fullCopy]]].
...
```

Código C.151: ScratchFrameMorph»importSpriteOrProject:

```
addTestStack: aBlockStack
    "Aligns the newly added script below the lowest script in the pane of Test tab
        ."

    | y bottom |
    y _ 10.
    testBlocksBin submorphsDo: [:m |
        bottom _ (m fullBounds bottom) - (testBlocksBin position y).
        (bottom > y) ifTrue:
            [y _ bottom]].
    aBlockStack position: testBlocksBin position + (20@(y+10)).

    aBlockStack newScriptOwner: self.
    testBlocksBin addMorph: aBlockStack.
```

Código C.152: ScriptableScratchMorph»addTestStack:

```
extractProjectFrom: aByteArray
...
   proj allMorphsDo: [:m |
        (m isKindOf: ScriptableScratchMorph) ifTrue: [  "covert to new blocks"
            m convertStacksToTuples.
            m convertTestStacksToTuples.
            m convertTuplesToStacks.
            m convertTestTuplesToStacks]].
...
```

Código C.153: PaintCanvas»extractProjectFrom:

```
setLanguage: aString
    "Set my language and update my blocks."
...
    (workPane submorphs copyWith: workPane) do: [:m |
        (m isKindOf: ScriptableScratchMorph) ifTrue: [
            m convertStacksToTuples.
            m convertTestStacksToTuples]].
...
    (workPane submorphs copyWith: workPane) do: [:m |
        (m isKindOf: ScriptableScratchMorph) ifTrue: [
            m convertTuplesToStacks.
            m convertTestTuplesToStacks]].
...
```

Código C.154: ScratchFrameMorph»setLanguage:

```
stopAllProcesses
...
    allObjs do: [:obj |
        (obj blocksBin isKindOf: Morph) ifTrue: [
            obj blocksBin submorphs do: [:b |
                (b isKindOf: BlockMorph) ifTrue: [b clearProcess]]].
        (obj testBlocksBin isKindOf: Morph) ifTrue: [
            obj testBlocksBin submorphs do: [:b |
                (b isKindOf: BlockMorph) ifTrue: [b clearProcess]]]].
...
```

Código C.155: ScratchStageMorph»stopAllProcesses

```
fileMenu: aMenuTitleMorph

    | menu |
    menu _ CustomMenu new.
...
    menu add: 'Write Project Summary' action: #writeSummaryFile.
    menu add: 'Write Multiple Project Summaries' action: #writeMultipleSummaries.
...
```

Código C.156: ScratchFrameMorph»fileMenu:

```
writeSummaryTotalsOn: aStream
    "Write the totals for this project on the given stream."

    | sprites uniqueCostumes uniqueSounds stackCount testStackCount |
    sprites _ workPane submorphs select: [:m | m isKindOf: ScriptableScratchMorph
        ].
    sprites _ sprites asArray copyWith: workPane.
    uniqueCostumes _ IdentitySet new: 100.
    uniqueSounds _ IdentitySet new: 100.
    stackCount _ 0.
    testStackCount _ 0.
    sprites do: [:m |
        m media do: [:item |
            item isImage ifTrue: [uniqueCostumes add: item form].
            item isSound ifTrue: [uniqueSounds add: item sound]].
        stackCount _ stackCount + m blocksBin submorphCount.
        testStackCount _ testStackCount + m testBlocksBin submorphCount.].

    aStream nextPutAll: 'Totals: '; crlf.
    aStream nextPutAll: '    Sprites: ', (sprites size - 1) printString; crlf.
    aStream nextPutAll: '    Scripts Stacks: ', stackCount printString; crlf.
    aStream nextPutAll: '    Test Stacks: ', testStackCount printString; crlf.
    aStream nextPutAll: '    Unique costumes: ', uniqueCostumes size printString;
        crlf.
    aStream nextPutAll: '    Unique sounds: ', uniqueSounds size printString; crlf
        .
```

Código C.157: ScratchFrameMorph»writeSummaryTotalsOn:

```
printSummaryOn: aStream
...
    stacks _ blocksBin submorphs select: [:m | m isKindOf: BlockMorph].
    stacks size = 0 ifTrue: [
        aStream nextPutAll: '  No stacks.'; crlf; crlf.
        ^ self].
    aStream nextPutAll: '  Scripts Stacks (', stacks size printString, '):'; crlf.

    hats _ stacks select: [:m | m isKindOf: HatBlockMorph].
    otherStacks _ stacks select: [:m | (m isKindOf: HatBlockMorph) not].

    hats, otherStacks do: [:item |
        item printCodeOn: aStream indent: 1.
        (item isKindOf: ReporterBlockMorph) ifTrue: [aStream crlf].
        aStream crlf].

    testStacks _ testBlocksBin submorphs select: [:m | m isKindOf: BlockMorph].
    testStacks size = 0 ifTrue: [
        aStream nextPutAll: '  No stacks.'; crlf; crlf.
        ^ self].
    aStream nextPutAll: '  Test Stacks (', testStacks size printString, '):'; crlf
        .

    testHats _ testStacks select: [:m | m isKindOf: HatBlockMorph].
    testOtherStacks _ testStacks select: [:m | (m isKindOf: HatBlockMorph) not].

    testHats, testOtherStacks do: [:item |
        item printCodeOn: aStream indent: 1.
        (item isKindOf: ReporterBlockMorph) ifTrue: [aStream crlf].
        aStream crlf].
```

Código C.158: ScriptableScratchMorph»printSummaryOn:

```
rightButtonMenu

    | menu |
    menu _ CustomMenu new.
    menu add: 'help' action: #presentHelpScreen.
...
```

Código C.159: BlockMorph»rightButtonMenu

```
presentHelpScreen
    "Answer the name of the help screen for this block,
    or nil if no help is available."

    | fr |
    fr _ self ownerThatIsA: ScratchFrameMorph.
    fr
        ifNil: [^ nil]
        ifNotNil: [fr presentHelpScreen: self helpScreenName]
```

Código C.160: BlockMorph»presentHelpScreen

```
helpScreenName
    "Answers the name of the help screen for this block."

    ^ 'importBlock'
```

Código C.161: ImportCBlockMorph»helpScreenName

```
helpScreenName
    "Answers the name of the help screen for this block."

    ^ 'scriptBlock'
```

Código C.162: UserCommandBlockMorph»helpScreenName

```
helpScreenName
    "Answer the name of the help screen for this hat block,
    or nil if no help is available."

    (self isKindOf: MouseClickEventHatMorph) ifTrue: [^ 'mouseclickHat'].
    (self isKindOf: KeyEventHatMorph) ifTrue: [^ 'keyHat'].
    self eventName = 'Scratch-StartClicked'
        ifTrue: [^ 'startHat']
        ifFalse: [(self eventName = 'Scratch-StartScriptClicked')
            ifTrue: [^ 'startScriptHat']
            ifFalse: [^ 'broadcastHat']].

    ^ nil
```

Código C.163: EventHatMorph»helpScreenName

```
rightButtonMenu
...
    (owner isKindOf: ScratchBlockPaletteMorph) ifFalse: [
        ...
    ]
    ifTrue: [
        menu addLine.
        menu add: 'delete script' action: #deleteScript.
        menu addLine.
        menu add: 'export script' action: #exportScript.
    ].
...
    (choice _ menu localize; startUp) ifNil: [^ self].
    (#(presentHelpScreen duplicate delete saveScript showDefinition deleteScript
        updateBlockDefinition saveScriptAs exportScript) includes: choice)
            ifTrue: [^ self perform: choice].
...
```

Código C.164: UserCommandBlockMorph»rightButtonMenu

```
saveScriptFile: spec
...
        result _ ScratchFileChooserDialog
            chooseNewFileDefault: spec
            title: 'Export script'
            type: #scripts.
...
```

Código C.165: ScratchFileChooserDialog»saveScriptFile:

```
type: t
    "Set the type of thing being opened/saved in the file dialog box, in order
    to include the appropriate shortcuts.  Then add relevant shortcut buttons
    and set the directory. Types include:
    #background
    #costume
    #list
    #project
    #projectSummary
    #scriptSnapshot
    #sound
    #sprite
    #stageShot
    #scripts"

    type _ t.
    self addShortcutButtons.
    self setDirectory: (ScratchFileChooserDialog getLastFolderForType: type).
```

Código C.166: ScratchFileChooserDialog»type:

```
getDefaultFolderForType: type

    | mediaDir |
    (type = #project) ifTrue: [^ self userScratchProjectsDir].
    (type = #scripts) ifTrue: [^ self userScriptsDir].
...
```

Código C.167: ScratchFileChooserDialog»getDefaultFolderForType:

```
userScriptsDir
    "Return the path to the user's 'My Scripts' project folder, usually located
    inside the user's 'Documents' folder. If the folder does not already exists,
    attempt to create it. If the .ini file specifies an alternate home directory,
    create the folder there. If the directory can't be created, return the user's
    home folder."

    | scriptsFolderName homeDir |
    scriptsFolderName _ 'My Scripts'.  "if this were localized a user could get
    multiple project folders for different languages..."

    homeDir _ self homeDir.

    "try to create My Scripts folder in the user's homeDir"
    (homeDir directoryExists: scriptsFolderName) ifFalse: [
        [homeDir createDirectory: scriptsFolderName] ifError: []].

    ^ (homeDir directoryExists: scriptsFolderName)
        ifTrue: [homeDir directoryNamed: scriptsFolderName]
        ifFalse: [homeDir]
```

Código C.168: ScratchFileChooserDialog»userScriptsDir

```
addShortcutButtons
    "Add shortcut buttons for my type to the shortcutColumn."
...
    #scripts = self type ifTrue: [
        shortcutColumn addMorphBack:
            (self shortcutButtonLabel:
                'Scripts' action: #sampleScripts icon: #folderIcon).
        shortcutColumn addMorphBack: spacer fullCopy.
        shortcutColumn addMorphBack:
            (self shortcutButtonLabel:
                'My Scripts' action: #userScripts icon: #folderIcon)].
```

Código C.169: ScratchFileChooserDialog»addShortcutButtons

```
sampleScripts
    "Sample Scripts button was pressed."

    (FileDirectory default directoryExists: 'Scripts') ifTrue: [
        list currentDirectory:
            (FileDirectory default directoryNamed: 'Scripts')].
```

Código C.170: ScratchFileChooserDialog»sampleScripts

```
userScripts
    "My Scripts button was pressed."

    list currentDirectory: self class userScriptsDir.
```

Código C.171: ScratchFileChooserDialog»userScripts

```
saveScriptFile: spec
    "Choose a file for saving the current script file. Answer the full name of
    the file in which to save the script or #cancelled if the operation is
        cancelled."
    "ScratchFileChooserDialog saveScriptFileFor: nil"

    | result |
    ScratchFileChooserDialog deleteDuplicates.
    result _ ScratchFileChooserDialog
        chooseNewFileDefault: spec
        title: 'Export script'
        type: #scripts.

    result = #cancelled ifTrue: [^ result].

    (result asLowercase endsWith: '.script') ifFalse: [result _ result, '.script
        '].
    ^ result
```

Código C.172: ScratchFileChooserDialog≫saveScriptFile:

```
confirmScriptFileOverwriteIfExisting: aFilename
    "If the given file exists, ask the user if they want to overwrite it or
    pick a different file name."

    | response fName |
    fName _ aFilename.
    (fName endsWith: '.script') ifFalse: [fName _ fName, '.script'].
    (FileDirectory default fileExists: fName) ifFalse: [^ aFilename].

    response _ DialogBoxMorph
        askWithCancel: 'The file name already exists. Overwrite existing file?'.
    response = #cancelled ifTrue: [^ #cancelled].
    response ifTrue: [^ fName] ifFalse: [^ false].
```

Código C.173: ScratchFileChooserDialog≫confirmScriptFileOverwriteIfExisting:

```
nameFromFileName: fileName
    "Return the given Scratch file name without the trailing .sb, .scratch, .extsb
    or .script extension, if it has one. Ensure the the result is UTF8."

    | s |
    s _ fileName.
    ...
    (s asLowercase endsWith: '.script') ifTrue: [s _ s copyFrom: 1 to: s size -
        7].
    s isUnicode ifFalse: [s _ UTF8 withAll: s].

    ^ s
```

Código C.174: ScratchFrameMorph≫nameFromFileName:

```
exportScript
    "Export this script to file."

    | fName result sFrame |
    sFrame _ self ownerThatIsA: ScratchFrameMorph.
    fName _ (ScratchFileChooserDialog saveScriptFile: self commandSpec).
    (fName size = 0 or: [fName = #cancelled]) ifTrue: [^ self].

    [(result _ ScratchFileChooserDialog confirmScriptFileOverwriteIfExisting:
        fName)
        = false] whileTrue: [
            fName _ ScratchFileChooserDialog saveScriptFile: 'script1'.
            (fName size = 0 or: [fName = #cancelled]) ifTrue: [^ self]].
    (result = #cancelled) ifTrue: [^ self].

    fName _ (sFrame nameFromFileName: fName), '.script'.
    self writeScriptFile: fName.
```

Código C.175: UserCommandBlockMorph≫exportScript

```
writeScriptFile: fileName
...
    blockDef _ (self receiver definitionOfUserBlock: self commandSpec) deepCopy.
    blockDef body: blockDef body tupleSequence.
...
```

Código C.176: UserCommandBlockMorph≫writeScriptFile:

```
initialize

    super initialize.
    spec _ nil.
    body _ nil.
    isGlobal _ true.
    receiver _ nil.
```

Código C.177: UserBlockDefinition≫initialize

```
storeFieldsOn: anObjStream

    self storeFieldsNamed: #(
        spec
        body
        isGlobal
        receiver
    ) on: anObjStream.
```

Código C.178: UserBlockDefinition≫storeFieldsOn:

```
initFieldsFrom: anObjStream version: classVersion

    self initFieldsNamed: #(
        spec
        body
        isGlobal
        receiver
    ) from: anObjStream.
```

Código C.179: UserBlockDefinition≫initFieldsFrom:version:

```
addBlockWithBody:
...
    def _ UserBlockDefinition new spec: blockName; body: body; isGlobal: isGlobal;
        receiver: self class asString.
...
```

Código C.180: ScriptableScratchMorph»addBlockWithBody:

```
writeScriptFile: fileName

    | scriptDirectory scriptName blockDef saveError out |
    scriptDirectory _ FileDirectory on: (FileDirectory dirPathFor: fileName).
    scriptName _ FileDirectory localNameFor: fileName.
    blockDef _ (self receiver definitionOfUserBlock: self commandSpec) deepCopy.
    blockDef body: blockDef body tupleSequence.

    saveError _ nil.
    [   out _ FileStream newFileNamed: (scriptDirectory unusedNameStartingWith: '
        tmp').
        out
            ifNil: [saveError _ 'Folder may be locked or read-only']
            ifNotNil: [
                out binary.
                ObjStream new storeObj: blockDef on: out.
                out close].
    ] ifError: [:err :rcvr |
        out ifNotNil: [
            [   out close.
                scriptDirectory deleteFileNamed: out localName.
            ] ifError: []].  "clean up, ignoring any errors"
        saveError _ err].

    saveError
        ifNil: [
            scriptDirectory deleteFileNamed: scriptName.
            [scriptDirectory rename: out localName toBe: scriptName]
                ifError: [^ self inform: 'Save failed' withDetails:
                    'Is the folder read-only?' localized].
            scriptDirectory setMacFileNamed: scriptName type: 'STsb' creator: '
                MITS']
        ifNotNil: [
            scriptName _ ''.
            self inform: 'Save failed' withDetails: saveError].
```

Código C.181: UserCommandBlockMorph»writeScriptFile:

```
userClasses
...
        (184        ScratchFrameMorph)
        (185        ScrollFrameMorph2)
        (186        PasteUpMorph)
        (187        ScratchScriptEditorMorph)
        (188        ImageFrameMorph)
        (189        ScratchLibraryMorph)
        (190        ScratchViewerMorph)
        (191        SystemWindow)
        (192        ScratchTabPaneMorph)
        (193        ScratchThumbnailMorph)
        (194        UpdatingStringFieldMorph)
        (195        ResizableToggleButton2)
        (196        PluggableTextMorph)
        (197        IconicButton)
        (198        PluggableListMorph)
        (199        PluggableMessageCategoryListMorph)
        (200        LibraryItemMorph)
        (201        ScratchSpriteLibraryMorph)
        (202        ScratchBlockPaletteMorph)
        (203        ScratchMenuTitleMorph)
        (204        TransformMorph)
        (205        PluggableButtonMorph)
        (206        TextMorphForEditView)
        (207        ScratchScrollBar)
        (208        ScratchCommentMorph)
        (209        ScratchConnectorMorph)
        (210        ScratchResizeMorph)
        (211        ListContentsBlockMorph)
    )
```

Código C.182: ObjStream≫userClasses

```
fileMenu: aMenuTitleMorph
...
    menu addLine.
    menu add: 'Import Project' action: #importScratchProject.
    menu add: 'Export Sprite' action: #exportSprite.
    menu add: 'Import Script' action: #importScript.
    menu addLine.
...
```

Código C.183: ScratchFrameMorph≫fileMenu:

```
importScript

    scriptsPane target importScript.
```

Código C.184: ScratchFrameMorph≫importScript

```
readScriptFile: fileName

    | f def data s |
    (FileDirectory default fileExists: fileName) ifFalse: [^ self].
    f _ (FileStream readOnlyFileNamed: fileName) binary.
    f ifNil: [self inform: 'Could not read' withDetails: fileName. ^ nil].
    [   data _ f contentsOfEntireFile.
        s _ ReadStream on: data.
        s position: 0.
        def _ ObjStream new readObjFrom: s.
    ] ifError: [:err :rcvr | self inform: 'Could not read project; file may be
        damaged' withDetails: '(', err, ')'. ^ nil].
    ^ def.
```

Código C.185: ScriptableScratchMorph»readScriptFile:

```
createScriptWithDefinition: definition

    definition body: (self stackFromTupleList: definition body receiver: self).
    self createUserBlockWithDefinition: definition global: definition isGlobal.
```

Código C.186: ScriptableScratchMorph»createScriptWithDefinition:

```
importScript

    | response def |
    response _ ScratchFileChooserDialog
        chooseExistingFileType: #scripts
        extensions: #(script)
        title: 'Import Script'.
    response = #cancelled ifTrue: [^ nil].
    def _ self readScriptFile: response.
    def ifNil: [^ nil].
    (def isGlobal and: [self isKindOf: ScratchStageMorph]) ifTrue: [
        self beep.
        DialogBoxMorph warn: 'Cannot import a global script to the Stage.'.
        ^ nil.
    ].
    (def isGlobal not and: [(def receiver = 'ScratchSpriteMorph') and:
        [self isKindOf: ScratchStageMorph]]) ifTrue: [
            self beep.
            DialogBoxMorph warn: 'Cannot import a sprite''s local script to the
                Stage.'.
            ^ nil.
    ].
    (def isGlobal not and: [(def receiver = 'ScratchStageMorph') and:
        [self isKindOf: ScratchSpriteMorph]]) ifTrue: [
            self beep.
            DialogBoxMorph warn: 'Cannot import a Stage''s script to a sprite.'.
            ^ nil.
    ].
    (self existsBlock: def spec global: def isGlobal) ifTrue: [
        self beep.
        DialogBoxMorph warn: 'A script with that spec already exists.'.
        ^ nil.
    ].
    self createScriptWithDefinition: def.
    (self ownerThatIsA: ScratchFrameMorph) viewerPane currentCategory: 'MyScripts
        '.
```

Código C.187: ScriptableScratchMorph»importScript

Anexo D

Ferramentas utilizadas

- Balsamiq Mockups 2.1.10;
- Pixlr;
- Visual Paradigm for UML 8.0 (Community Edition).