UNIVERSIDADE DE ÉVORA

ESCOLA CIÊNCIAS E TECNOLOGIA

DEPARTAMENTO DE INFORMÁTICA

Jogo para aprender a programar

Pedro Farias Mateus

Orientação: Lígia Maria Rodrigues da Silva Ferreira

Mestrado em Engenharia Informática

Dissertação

Évora, 2016

Esta dissertação inclui as críticas e as sugestões feitas pelo júri

UNIVERSIDADE DE ÉVORA

Escola de Ciências e Tecnologia

Departamento de Informática

Jogo para aprender a programar

Pedro Farias Mateus

Orientação *Lígia Maria Rodrigues da Silva Ferreira*

Engenharia Informática

Dissertação

16 de Dezembro de 2016

Esta dissertação inclui as críticas e sugestões feitas pelo Júri

Dedico este trabalho à minha avó e aos meus pais

Prefácio

Programar continua a ser uma das capacidades mais difíceis de aprender na área das Ciências de Computação. Neste trabalho foi feito um estudo acerca das dificuldades em aprender/ensinar programação, investigou-se o uso de linguagens de programação visual e de jogos como ferramentas auxiliares na educação deste tema. Para terminar, aplicaram-se todos os conteúdos encontrados nesta investigação para criar um jogo para aprender a programar, tendo este a sua própria linguagem de programação visual.

Agradecimentos

Gostaria de agradecer aos meus pais pelo apoio moral e financeiro dado, não só na elaboração desta tese como também do meu percurso académico. Da minha irmã da ajuda na preparação da defesa da dissertação.
Também gostaria de agradecer à minha orientadora Lígia Ferreira pelo apoio, dedicação e contribuição nesta dissertação mesmo de esta se tratar de um tópico obscuro e pouco investigado nos dias de hoje.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

URL *Uniform Resource Locator*

ECT Escola de Ciências e Tecnologia

UE Universidade de Évora

LVP Linguagem Visual de Programação

LPV Linguagem de Programação Visual

Sumário

Neste documento foi feita a apresentação do conceito de Programação Visual, estudados alguns exemplos de Linguagens de Programação Visual e investigado vantagens e desvantagens sobre o seu uso comparativamente à programação tradicional (por texto) e, adicionalmente, foram estudados três jogos para aprender a programar com fortes características visuais. A partir desta investigação, foi criada uma nova Linguagem de Programação Visual(LPV) sendo a base de um prototipo de um jogo para aprender a programar. Este jogo é representado pela deslocação e execução de tarefas feitas por um robô dentro de uma casa, sendo o objetivo do programador colocar objetos nesta casa que manipulam a deslocação do robô dentro desta de forma a resolver um problema.

Palavras chave: Programação Visual, Ensino, Jogo

Abstract

A game to learn programming

Aplication of concepts of visual programming to create a game to learn how to program

In this document we will explain the concept of Visual Programming, study some examples of Visual Programming Languages, analyze some arguments in favor and against its use in comparison to traditional programming (by text) and, additionally, study three games to learn programming with strong visual features. Using this research, a new Visual Programming Language(VPL) was created, being the basis of a prototype of a game to learn programming. This game is represented by the movement and execution of tasks done by a robot inside a house, being the goal of the programmer to place objects in this house that manipulate the movement of the robot inside of it in order to solve a problem.

Keywords: Visual Programming, Education, Game

1

Introdução

As linguagens de programação visual (LPV) são linguagens que utilizam símbolos ou ícones na construção de programas ao invés de utilizarem exclusivamente texto [3][4]. A ideia da utilização de linguagens visuais para programar surgiu nos anos 60, graças à existência de gráficos computorizados de baixo custo monetário. A investigação em LPVs evidenciou-se nas décadas seguintes com o intuito de explorar as vantagens no uso desta metodologia para a criação de software como alternativa às linguagens de programação por texto já existentes[3][5]. Com este trabalho pretende-se: utilizar os conceitos de programação visual na criação de um jogo para aprender a programar, explorar a possibilidade do uso de jogos como ferramenta de auxilio a programadores principiantes e contribuir mais para o tópico de LPVs.

1.1 Motivação

O ensino de programação sempre foi problemático e considera-se atualmente um dos grandes desafios de ensino na área das ciências de computação [6]. Para compreender melhor este problema, existe a analogia feita por Gries em 1974 [6]:
"Permite-me fazer uma analogia para tornar o meu ponto claro. Suponha que você assiste a um curso

de construção de armários. O instrutor mostra rapidamente uma serra, um plano, um martelo e outras ferramentas, permitindo que você use cada ferramenta por alguns minutos. De seguida, mostra-lhe um armário belíssimo e acabado. Para terminar, ele diz a si para desenhar e construir seu próprio armário e trazer o produto acabado em poucas semanas. Você pensaria que ele era louco!"
O que demonstra a importância de aprender utilizar as ferramentas com exemplos e com prática, não sendo o suficiente saber o que é possível criar com estas. Ao que parece, este problema persiste ao analisar-se livros educativos de programação mais recentes[6], não existindo ainda uma solução definitiva para o problema. Com este trabalho pretende-se criar um jogo que ajude novos programadores a aprender a programar, explorando a possibilidade do uso jogos como solução para este problema.

1.1.1 Oportunidades

Atualmente existem jogos que desenvolvem o raciocínio e alguns dos conceitos de programação (ver capitulo 4), como também ferramentas de auxilio criadas especificamente para o ensino da programação como o Flowgorithm[7]. No entanto, nenhuma destas aplicações teve sucesso na adoção e integração em cursos de ensino em programação.
Um dos estudos[8] apresentado no capitulo 3 mostrou resultados favoráveis na aprendizagem de uma LPV (Forms/3), indicando que talvez exista potencial no uso de uma linguagem programação visual para aprender a programar. Ainda neste estudo, existe o argumento de que as investigações atuais são insuficientes para a justificar o uso de LPVs no ensino de programação[8].
Pretende-se assim com esta dissertação, contribuir mais para o tema de LPVs no ensino de programação, assim como criar uma ferramenta auxiliar para programadores principiantes.

2

Linguagens visuais de programação e motivações do seu uso no ensino de programação

Entende-se por Programação Visual o conjunto de ferramentas utilizadas para criar um programa onde são utilizados ícones, diagramas ou qualquer outro tipo de objeto gráfico [5], com o intuito de tornar o processo de construção de programas mais facilitado [9]. Ao invés de escrever um programa de forma linear (numa só dimensão) só com texto, numa linguagem de programação visual, o programador manipula objetos gráficos, também chamados ícones, sendo estes representações das componentes do programa, num espaço de duas ou mais dimensões [9]. Este processo faz com que seja necessário construir um programa consoante uma gramática espacial de forma a que as relações/ligações, a ordem entre estes objetos e a posição dos mesmos constituem um programa compilável, sendo portanto necessário obedecer às regras de uma linguagem de programação visual.
Na Linguagem de Programação Visual ou LPV utilizam-se ícones gráficos e regras especiais de leitura e escrita na construção de um programa. Tal como qualquer outra linguagem de programação, de forma a criar um programa executável, é necessário seguir uma gramática com as suas regras.

2.1 Tipos de LPVs e exemplos

As linguagens de programação visual podem ser categorizadas da seguinte forma:

- Linguagens visuais por diagramas - são linguagens que utilizam a notação de diagramas de fluxo para criar um programa (exemplo: Flowgorithm [7]);
- Linguagens visuais puras ou icónicas - são linguagens que apenas utilizam objetos gráficos para criar e executar um programa. O uso de texto nestas linguagens é inexistente ou muito escasso (exemplo: ToonTalk [10]);
- Linguagens híbridas - são linguagens que combinam algumas das características de linguagens visuais com as características de linguagens textuais (exemplo: Scratch [1]);
- Linguagens baseadas em folhas de cálculo (spreadsheets) - são linguagens que utilizam o paradigma das folhas de cálculo identificando objetos/valores em caixas de forma distribuir, identificar e programar de uma forma mais organizada. Assim sendo, a referenciação destes objetos torna-se mais facilitadora, ao utilizar-se o identificador atribuído a cada objeto do programa (exemplo: Forms/3 [11]);

2.1.1 Flowgorithm

Flowgorithm [7] é uma ferramenta de programação para auxiliar programadores principiantes na criação de programas utilizando diagramas de fluxo. O autor desta aplicação argumenta que, dependendo da linguagem de programação, a criação de programas simples requer em alguns casos demasiadas linhas de código confuso e que o Flowgorithm permite o programador focar-se apenas na construção do algoritmo do programa não sendo necessário preocupar-se com os detalhes e nuances da linguagem de programação. Os diagramas criados podem ser executados diretamente nesta aplicação ou convertidos para outras linguagens de alto nível (C#, C++, Java, Pascal, Python, etc.) podendo-se observar a respetiva correspondência do código convertido (ver figura 2.1).
Os programas são construídos a partir de um diagrama inicial muito simples: 2 elementos com o nome de "*Main*"e "*End*", sendo estas representações do inicio e do fim do programa. O paradigma da linguagem pode ser descrita como uma linguagem estruturada[12] (devido à disposição do código) e imperativa[12] (cada caixa representa uma instrução).

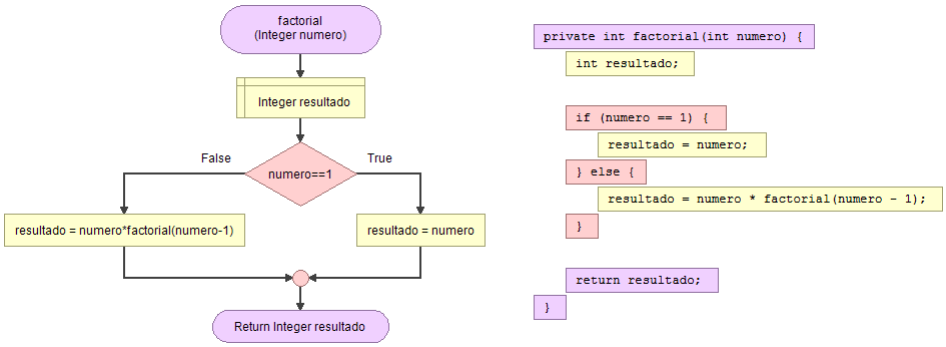


Figura 2.1: Exemplo de um programa do Flowgorithm e o código correspondente ao diagrama

A partir desse diagrama é possível adicionar mais elementos (de diversos tipos) entre esses dois objetos, de forma a construir um programa. A ferramenta tem os seguintes grupos de objetos disponíveis para construir o programa:

- Input/Output - contem duas figuras: uma para o "*input*", utilizado para receber uma variável do teclado e a de "*output*", utilizado para apresentar uma mensagem no ecrã;
- Variáveis - contem duas figuras uma para declarar variáveis e outra para executar "*assign*"de uma expressão (não gráfica) a uma variável já declarada;
- Control - contem duas figuras responsáveis pelo controlo do programa: uma para criar uma condição "*if*"e outra para chamar uma função criada pelo utilizador;
- Loop - contem as figuras representativas das instruções utilizadas em ciclos: uma para criar um ciclo "*for*", outra para criar um ciclo "*while*"e ainda outra para criar um ciclo "*do*".

No diagrama é também possível adicionar comentários em cada ligação entre dois objetos.
Tal como foi mencionado anteriormente, existe a possibilidade de criar múltiplos diagramas num só programa, os quais podem ser utilizados noutros diagramas, da mesma forma como se utilizam funções num programa textual, assim como a possibilidade de chamar funções recursivamente. Flowgorithm tem também acesso a um conjunto de funções matemáticas e funcionalidades disponíveis noutras linguagens de programação como por exemplo: cos, log, sqrt, len, ToChar, random, size, entre outros [13].

2.1.2 Toontalk

Toontalk [10], lançado oficialmente em 1999, é um jogo de programação interativo, criado com o objetivo de ensinar crianças a aprender a programar de uma forma intuitiva e divertida. O programador toma controlo de uma personagem, um "avatar", que interage com uma cidade de Lego, a representação de uma computação. O jogador desloca-se pela cidade construindo casas de Lego, cada casa é a representação de um processo, trazendo também consigo uma caixa de ferramentas contendo um conjunto de objetos para "programar cada casa". Dentro de cada casa são programados robôs para efetuarem diversas tarefas sendo cada robô configurado dando exemplos das ações a efetuar por este. O paradigma da linguagem é descrita como programação lógica de restrição concorrente[14]: restrição, visto que as ações do robô são restringidas pelos exemplos dados pelo programador; e concorrente, visto que cada casa representa um processo em execução podendo estas executar instruções em simultâneo. Utilizando o "avatar", ao entrar e sentar-se no chão de uma casa (para começar a programar), o jogador terá de utilizar os seguintes objetos:

- Caixa - representação de um objeto, uma mensagem ou um "*array*". As caixas são utilizadas para guardar palavras, números, outras caixas ou outros objetos;
- Robô - representação de um conjunto de instruções, um método que será executado dentro da casa. É necessário ensinar primeiro ao robô as ações que deve fazer de forma manual (mexendo letras para uma caixa por exemplo), as quais serão repetidas quando este for ativado. O robô consegue utilizar e manipular as mesmas ferramentas que o próprio programador manipula;
- Balança - representação das operações de comparação. A balança é utilizada para comparar dois objetos;
- Letra ou número - representação de constantes, podendo ser guardados dentro de caixas, comparados com a balança ou movidos pelo robô;

- Camião - representação da criação de novos processos. O camião é utilizado para construir novas casas na cidade de lego (fora da casa), transportando com ele objetos já existentes (por exemplo robôs) para que possam ser utilizados na nova casa;
- Bomba - representação da terminação de um processo. A bomba destrói a casa onde é utilizada;
- Pássaro - representação da transmissão dos dados entre dois processos. O pássaro pode enviar caixas, com ou sem objetos, para outra casa;
- Ninho - representação do local onde o pássaro irá entregar a mensagem, no caso de serem enviadas múltiplas mensagens a mais recente é posta no fim da pilha;
- Caderno - representação dos ficheiros e/ou objetos que estão guardados no disco rígido como imagens ou sons, podendo os mesmos serem abertos e utilizados por um robô.

Para além destes objetos, o programador tem também acesso a ferramentas utilizadas para copiar, modificar e apagar objetos:

- Varinha mágica - utilizada para copiar objetos;
- Aspirador - utilizado para aspirar e expirar objetos.

Para programar cada casa, não basta colocar só objetos no chão da mesma, é necessário utilizá-los e interagir com os mesmos de forma a efetuar as tarefas desejadas. Por exemplo, queremos criar uma casa que recebe uma caixa com dois números e que devolve a soma destes. Existem duas maneiras de resolver este problema, nomeadamente o programador efetua a soma manualmente ou treinando robôs. Para tal, é necessário ter-se um ninho (para receber as caixas), um pássaro (para devolver o resultado da soma) e dois robôs - um robô que apanhe as caixas no ninho e que faça a soma e outro que apanhe a caixa com o resultado e que dê ao pássaro para retornar a mesma. Para treinar o robô basta dar uma caixa como exemplo (que contenha 2 números) e efetuar as ações manualmente: retirar os dois números da caixa, colocar um número sobre o outro, apresentando uma animação da soma (um rato a martelar os números) e colocar o número somado dentro da caixa. Depois é necessário treinar o segundo robô para apanhar qualquer caixa no chão contendo só um número e dar a mesma ao pássaro, mais uma vez manualmente. Quando os robôs estiverem programados, a casa estará preparada para fazer a soma de todas as caixas com dois números e devolver simultaneamente as caixas com as respetivas soma.

Para ensinar de que forma estes objetos funcionam e de como se programa nesta linguagem existe um tutorial, um jogo, baseado num conjunto de puzzles para resolver. Esta componente será investigada na secção de jogos no estado de arte (secção 5).

NOTA: Devido à natureza na forma como os programas são criados e executados no Toontalk, não é possível colocar uma imagem exemplo de um programa, visto que tanto a configuração como a execução dos robôs é interativa, só podendo-se observar um programa Toontalk num vídeo e não numa imagem.

2.1.3 Scratch

Scratch [1] é uma ferramenta educativa de programação que permite criar jogos, animações, histórias interativas, entre outros. Ao contrário do que muitos pensam, Scratch não é um jogo para aprender a programar, visto não conter qualquer tipo de elementos que compõem um jogo, tais como: um objetivo a completar, um conjunto de puzzles a resolver, pontuação, etc. A ferramenta é descrita como um palco (a

zona onde se visualiza a execução do programa) com "atores"(imagens), podendo-se programar o cenário do palco e a forma como estes "atores"se deslocam, os sons que fazem e a forma como interagem com o utilizador. Para tal o programador tem ao seu dispor um conjunto diverso de blocos com formas diferentes, agrupados consoante o tipo de funcionalidade (identificados pela sua cor). O programa constrói-se ligando blocos em pilha como peças puzzle, restringindo a ligação de certos blocos com outros para evitar erros de compilação. Cada bloco contem uma frase que descreve o funcionamento do bloco e os valores editáveis pelo programador. Por exemplo, "anda [x] passos"em que o valor x é editável pelo programador. Um exemplo de um programa Scratch pode ser observado na figura 2.2.

O paradigma da linguagem pode ser descrita como programação imperativa[15] (cada bloco representa uma instrução) e acionada por evento[15] (existem blocos especiais que apenas se ativam após ocorrido um evento como clicar no botão do rato, carregar numa tecla, etc.)
Os blocos estão categorizados da seguinte forma:

- Movimento (azul escuro) - são responsáveis pela deslocação dos "atores"no programa, como por exemplo, deslocar o ator para a esquerda ou direita (descrito como passos), girar, deslocar para um conjunto de coordenadas específico, entre outros;

- Aparência (roxo) - afetam a aparência dos "atores"e do cenário. Exemplo: o "ator"dizer uma frase, ocultar/mostrar o "ator", mudar o cenário para outra imagem, aumentar o tamanho do "ator", entre outros;

- Som (rosa) - são blocos que controlam o som do programa, nomeadamente, tocar um som, aumentar e diminuir o volume, etc.;

- Caneta (verde escuro) - são blocos que permitem que os "atores"escrevam ou desenhem no cenário, como: fazer carimbos da imagem do "ator", desenhar linhas conforme o deslocamento do "ator", entre outros;

- Dados (laranja) - permitem o uso de variáveis e listas podendo estas serem utilizadas para guardar números ou "*strings*"e serem referenciados dentro de outros blocos;

- Eventos (castanho) - são blocos que controlam diversos eventos do programa, como por exemplo, o input do utilizador (carregar numa tecla, clicar num "ator") ou quando ocorre algo (uma mensagem é mostrada, o volume do som diminui para um determinado nível, etc.);

- Controlo (amarelo) - contem as funções de controlo do programa, alguns exemplos são: ciclos, condições (se x então y), esperar x segundos, entre outros;

- Sensores (azul claro) - são blocos que retiram informação do programa, tais como, devolver as coordenadas de um "ator", devolver as coordenadas da seta do rato, entre outros;

- Operadores (verde claro) - são todos os blocos que lidam com operações matemáticas (subtração, divisão, etc.), operações de comparação e operadores booleanos (AND, OR, NOT, etc.);

Para além dos 145 blocos disponíveis na ferramenta também é possível criar blocos personalizados.

Figura 2.2: Exemplo de um programa Scratch "*Animate the Crab*", disponível na página oficial da LPV[1]

2.1.4 Forms/3

Forms/3 [11] é uma ferramenta de programação visual, que permite criar programas utilizando o paradigma utilizado em folhas de cálculo. Isto é, os objetos, valores, funções (ou neste caso chamadas de fórmulas) e variáveis do programa estão contidos dentro de células, podendo cada uma ser referenciada noutras porções do programa. Um programa em Forms/3 (ver figura 2.3) é constituído por vários formulários ou (dentro do paradigma) folhas, em que cada folha é constituída por um conjunto de células desenhadas e configuradas pelo programador. Todas as células em cada folha têm de ser identificadas, podendo isto ser feito pelo programador, dando um nome à célula criada, ou a ferramenta nomeia automaticamente, ficando com um identificador único. Cada célula pode guardar uma expressão/fórmula, uma variável (uma String, um número ou um boolean), uma imagem ou uma ou mais células. Para além disso, existem também células especiais para a interface do programa: um botão e um menu seleção de opções (*radio buttons*). As fórmulas de uma célula podem conter expressões aritméticas, booleanas ou expressões com a notação "*if-then-else*", normalmente observadas em linguagens de programação. Dentro destas fórmulas, as células existentes podem ser referenciadas evocando o identificador, tal como é feito em folhas de cálculo, com a exceção de que quando se referencia uma célula dentro de outra, é necessário evocar o nome da célula que contem o que se deseja aceder, usando a mesma notação utilizada em "*arrays*"nas linguagens de programação (por exemplo: nomedoarray[coordenada]). O paradigma da linguagem pode ser descrita como programação declarativa[2] (visto que a estrutura da linguagem faz com que não seja necessário definir o fluxo da execução do programa).

Esta linguagem não suporta o uso de ciclos ("*for*"ou "*while*"), normalmente presentes noutras linguagens de programação, como tal utiliza um modelo temporal, onde é possível controlar os cálculos ou as atualizações feitas nas células conforme uma escala temporal, definida pela ferramenta de programação. Para se fazer uso desta escala, o programador necessita de utilizar operadores especiais dentro de expressões com intuito de referenciar valores anteriormente calculados (*earlier*) ou o valor inicial da célula, no inicio da escala temporal (*initially*) ou fixar o valor da célula ao cumprir um certo requisito (*until*). Ao utilizar estes operadores é possível criar-se ciclos dentro do programa e até mesmo criar-se expressões recursivas.

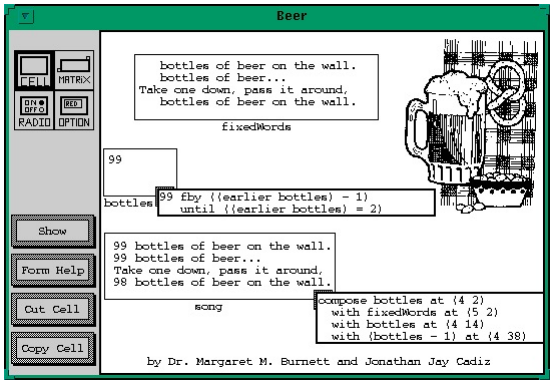


Figura 2.3: Programa Forms/3 "99 Beers on the wall", disponível na página oficial da LPV[2]

2.2 Estrutura e funcionamento de LPVs

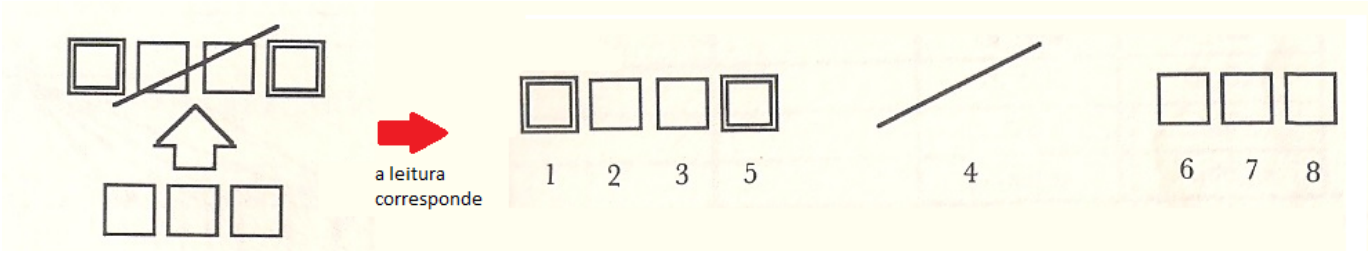
Tal como nas linguagens de programação por texto, as visuais têm passos de compilação muito semelhantes, destacando-se os processos de leitura e de *parsing*. À componente responsável por isto dá-se o nome de interpretador de ícones [4], que tem a capacidade de identificar os ícones gráficos, assim como de os ler na ordem correta. Para tal, faz-se uso da posição dos mesmos no programa ou utilizam-se as relações/ligações existentes entre os símbolos. Para que isto seja possível, o interpretador necessita de identificar cada símbolo conforme uma estrutura de dados, onde fica registada a informação do símbolo, permitindo assim a leitura e interpretação de cada símbolo no programa. Para tal, é necessária a seguinte informação:

- Representação lógica - de que forma o símbolo é identificado em texto, o seu tipo e o nome do identificador;
- Representação gráfica - a imagem que representa o símbolo da linguagem;
- Posição no programa - onde está posicionado este símbolo, a posição pode ser definida utilizando eixos cartesianos ou identificando a posição de cada um em relação aos outros símbolos do programa (por exemplo o ícone X está à esquerda do Y) ou em certas linguagens utilizando as conexões existentes entre estes;

Dentro das LPVs existe uma distinção entre as linguagens visuais icónicas e as linguagens visuais de diagramas pelas metodologias e estruturas utilizadas na leitura e *parsing* de programas construídos em cada linguagem.

2.2.1 Linguagens visuais icónicas

Seguindo os exemplos de linguagens visuais icónicas presentes no livro "Visual Languages and Visual Programming", é feita a descrição de uma linguagem visual, inspirada nos ícones Heidelberg[4], sendo utilizada para editar texto. Podemos constatar nesta linguagem que cada frase visual é composta por um conjunto de ícones elementares e por um ou mais ícones de processo que representam uma operação (concatenação, eliminação, etc.), cuja ordem de leitura depende destes operadores. Um exemplo na forma como os ícones elementares são lidos e identificados pode ser visto na figura 2.4.



Fonte: Livro "Visual Languages and Visual Programming"

Figura 2.4: Exemplo de leitura e identificação dos ícones elementares. A seta representa um ícone processo, não sendo necessário numerar a leitura deste ícone

A partir da leitura dos ícones é construída uma "*string*"de padrão, em que descreve a frase (em texto) correspondente à visual (semelhante ao processo "*tokenization*"utilizado nos compiladores de texto). Esta tradução é baseada num dicionário de ícones (ver figura 2.7) que identifica a parte lógica e a parte gráfica de todos os símbolos da linguagem visual. Para determinar o posicionamento espacial dos ícones numa "*string*"de padrão são utilizados três operadores espaciais: "" representa a concatenação horizontal; "&"representa concatenação vertical; "+"representa sobreposição espacial (ícone sobreposto a outro). Um exemplo da "*string*"de padrão correspondente à leitura dos ícones da figura 2.4 pode ser vista na figura 2.5.

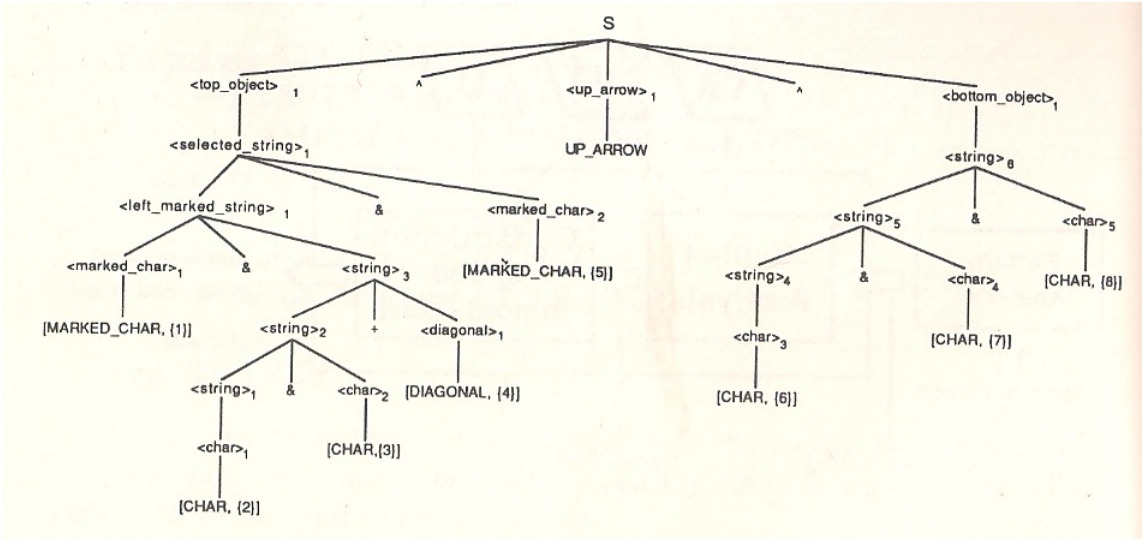
$$[\text{MARKED\_CHAR}, 1] \,\&\, [\text{CHAR}, 2] \,\&\, [\text{CHAR}, 3]$$
$$+ [\text{DIAGONAL}, 4] \,\&\, [\text{MARKED\_CHAR}, 5]$$
$$\wedge\ \text{UP\_ARROW} \wedge [\text{CHAR}, 6] \,\&\, [\text{CHAR}, 7] \,\&\, [\text{CHAR}, 8]$$

Fonte: Livro "Visual Languages and Visual Programming"

Figura 2.5: "*String*"de padrão dos ícones lidos na figura 2.4

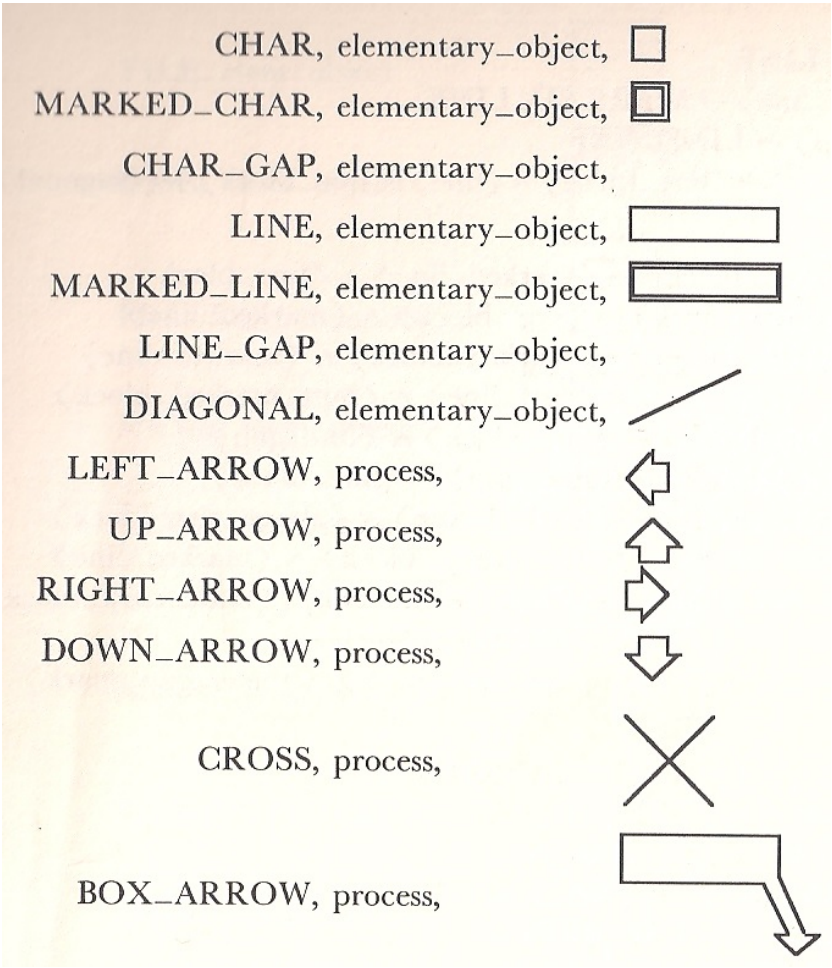
Com a "*string*"de padrão construída, e para certificar que foi lida uma frase válida dentro da linguagem visual, é utilizada uma gramática de ícones, que descreve as combinações de ícones permitidas numa frase desta linguagem(ver figura 2.8). A gramática utiliza, mais uma vez, os identificadores dos símbolos e os operadores espaciais ("", "&"e "+") para determinar de que forma estes podem-se colocar em relativamente a outros símbolos numa frase. Estando verificada a "*string*"de padrão, é então construído a árvore de *parsing*, conforme a posição relativa entre estes ícones (ícones adjacentes ficam em posições próximas na árvore e vice-versa). A árvore de *parsing* correspondente à "*string*"de padrão da figura 2.5 está apresentado na figura 2.6.

Isto faz com que a posição exata dos ícones não seja muito importante nesta linguagem visual, mas sim, a ordem que os ícones têm entre si. Lidos os ícones do programa, o resto do processo de compilação é semelhante ao utilizado nas linguagens textuais, utilizando estas árvores de *parsing*.



Fonte: Livro "Visual Languages and Visual Programming"

Figura 2.6: A árvore de *parsing* construída a partir da "*string*"de padrão da figura 2.6



Fonte: Livro "Visual Languages and Visual Programming"

Figura 2.7: Dicionário de ícones da linguagem visual do editor de texto

⟨char⟩ → CHAR
⟨marked_char⟩ → MARKED_CHAR
⟨char_gap⟩ → CHAR_GAP
⟨diagonal⟩ → DIAGONAL
⟨string⟩ → ⟨string⟩ & ⟨char⟩ | ⟨string⟩ + ⟨diagonal⟩ | ⟨char⟩
⟨left_mark_string⟩ → ⟨marked_char⟩ & ⟨string⟩
⟨right_mark_string⟩ → ⟨string⟩ & ⟨marked_char⟩
⟨selected_string⟩ → ⟨marked_char⟩ & ⟨right_mark_string⟩
| ⟨left_mark_string⟩ & ⟨marked_char⟩
⟨left_gap_string⟩ → ⟨char_gap⟩ & ⟨string⟩
⟨right_gap_string⟩ → ⟨string⟩ & ⟨char_gap⟩
⟨left_gap_mark⟩ → ⟨right_gap_string⟩ & ⟨marked_char⟩
⟨right_gap_mark⟩ → ⟨marked_char⟩ & ⟨left_gap_string⟩

⟨marked_string_family⟩ → ⟨left_mark_string⟩ | ⟨right_mark_string⟩
| ⟨selected_string⟩
⟨gap_string_family⟩ → ⟨left_gap_mark⟩ | ⟨right_gap_mark⟩

⟨line⟩ → LINE
⟨marked_line⟩ → MARKED_LINE
⟨line_gap⟩ → LINE_GAP
⟨line_block⟩ → ⟨line_block⟩ ∧ ⟨line⟩ | ⟨line_block⟩ + ⟨diagonal⟩ | ⟨line⟩

⟨top_marked_block⟩ → ⟨marked_line⟩ ∧ ⟨line_block⟩
⟨bott_marked_block⟩ → ⟨line_block⟩ ∧ ⟨marked_line⟩
⟨selected_block⟩ → ⟨top_marked_block⟩ ∧ ⟨marked_line⟩
| ⟨marked_line⟩ ∧ ⟨bott_marked_block⟩
⟨under_gap_block⟩ → ⟨line_block⟩ ∧ ⟨line_gap⟩
⟨above_gap_block⟩ → ⟨line_gap⟩ ∧ ⟨line_block⟩
⟨under_gap_mark⟩ → ⟨marked_line⟩ ∧ ⟨above_gap_block⟩
⟨above_gap_mark⟩ → ⟨under_gap_block⟩ ∧ ⟨marked_line⟩
⟨marked_line_family⟩ → ⟨top_mark_block⟩ | ⟨bott_mark_block⟩
| ⟨selected_block⟩
⟨gap_line_family⟩ → ⟨under_gap_mark⟩ | ⟨above_gap_mark⟩.

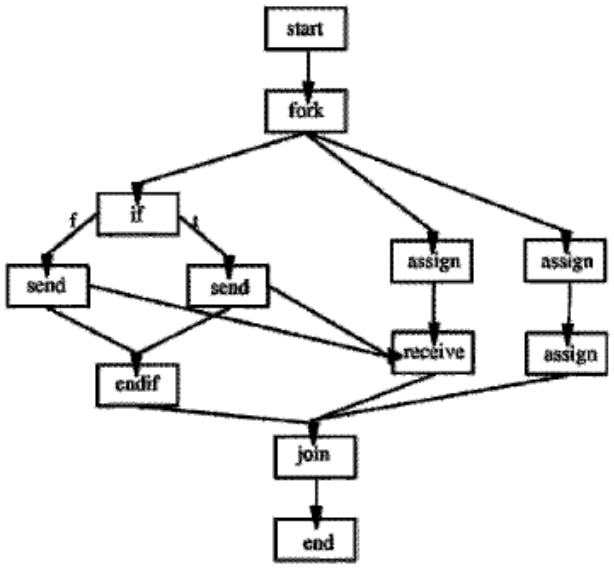
⟨right_arrow⟩ → RIGHT_ARROW
⟨left_arrow⟩ → LEFT_ARROW
⟨up_arrow⟩ → UP_ARROW
⟨down_arrow⟩ → DOWN_ARROW
⟨box_arrow⟩ → BOX_ARROW
⟨cross⟩ → CROSS
S → ⟨object_into_box⟩ + ⟨box_arrow⟩ ∧ ⟨box_arrow_argument⟩
⟨object_into_box⟩ → ⟨line⟩ | ⟨marked_line_family⟩
⟨box_arrow_argument⟩ → ⟨gap_string_family⟩
S → ⟨left_object⟩ & ⟨right_arrow⟩ & ⟨right_object⟩
S → ⟨right_object⟩ & ⟨left_arrow⟩ & ⟨left_object⟩
⟨left_object⟩ → ⟨line_block⟩ | ⟨marked_line_family⟩ | ⟨empty_object⟩
⟨right_object⟩ → ⟨marked_line_family⟩ | ⟨gap_line_family⟩
⟨empty_object⟩ → ⟨line_gap⟩ | ⟨char_gap⟩
S → ⟨top_object⟩ ∧ ⟨up_arrow⟩ ∧ ⟨bottom_object⟩
S → ⟨bottom_object⟩ ∧ ⟨down_arrow⟩ ∧ ⟨up_object⟩
⟨top_object⟩ → ⟨gap_string_family⟩ | ⟨marked_string_family⟩
⟨bottom_object⟩ → ⟨string⟩ | ⟨marked_string_family⟩ | ⟨empty_object⟩
S → ⟨cross⟩ + ⟨under_cross_object⟩
⟨under_cross_object⟩ → ⟨marked_string_family⟩ | ⟨marked_line_family⟩

Fonte: Livro "Visual Languages and Visual Programming"

Figura 2.8: Gramática de ícones da linguagem visual do editor de texto

2.2.2 Linguagens visuais de diagramas

Já numa linguagem visual de diagramas, a leitura não depende da ordem dos ícones, mas sim das ligações ou relações que estes têm entre si. No livro "Visual Languages and Applications"[9] são descritos dois tipos de gramáticas baseadas em grafos e utilizadas em linguagens visuais de diagramas. Na primeira, chamada gramática de grafos reservados, são utilizadas apenas as ligações existentes entre cada nó do grafo e na segunda, chamada gramática espacial de grafos, utilizam-se as ligações existentes, assim como a posição e a distância entre cada nó do grafo de um programa. Devido às semelhanças destas duas gramáticas, apenas se irá abordar a gramática de grafos reservados.

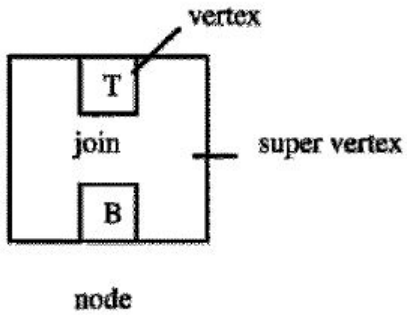


Fonte: Livro "Visual Languages and Applications"

Figura 2.9: Exemplo de um diagrama de fluxo de processo

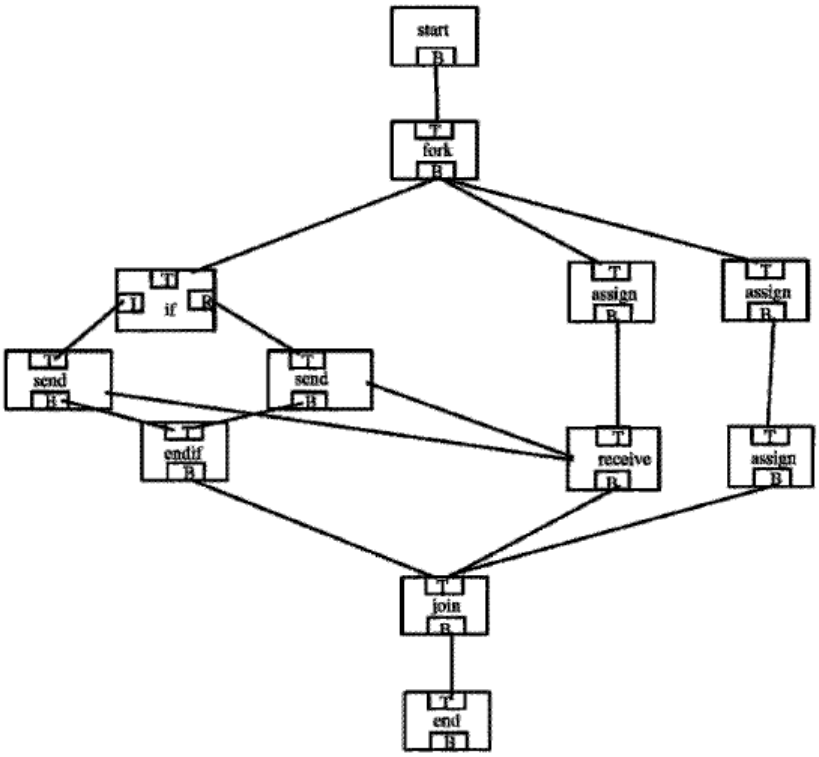
Utilizando como exemplo a linguagem visual apresentada em diagramas de fluxo de processo(ver figura 2.9), podemos observar na figura 2.10 que cada caixa do diagrama é representado por uma estrutura chamada de nó-margem (*node-edge*), a qual é composta por um retângulo chamado de super vértice (*super vertex*) e por retângulos menores, chamados de vértices (*vertex*).

O super vértice armazena informação da caixa do diagrama (geralmente o seu tipo) enquanto que os vértices guardam a informação das ligações feitas a esta caixa. Dependendo do tipo de caixa, é possível ter dois ou mais vértices de ligação, tal como se pode observar na figura 2.11. Para identificar corretamente as ligações feitas em cada caixa, cada vértice tem de ser devidamente identificado. Neste caso, cada vértice é identificado conforme a sua posição no super vértice (cima, baixo, esquerda e direita). A partir desta estrutura, ao ler-se um diagrama nós-margem é aplicando um conjunto de regras gramaticais da linguagem utilizada no programa, na qual são realizadas um conjunto de transformações (reduções ou *redex*) ao grafo deste até se obter um grafo inicial, podendo-se considerar este como um grafo elementar (o mais simples possível). Se o algoritmo de *parsing* não encontrar um único caminho ou uma sequencia de reduções em que se obtenha um grafo inicial, então o grafo não é válido, segundo a gramática utilizada.



Fonte: Livro "Visual Languages and Applications"

Figura 2.10: Imagem da estrutura nó-margem que representa a caixa de um diagrama



Fonte: Livro "Visual Languages and Applications"

Figura 2.11: Diagrama de nós-margem correspondente ao do diagrama da figura 2.9

2.3 Aprender programação

Programar é a ação de construir um programa capaz de realizar algum tipo de atividade ou operação num computador[16], como por exemplo enviar/receber e-mails, efetuar cálculos matemáticos, abrir/criar/editar

imagens ou comunicar com outra pessoa a partir da Internet. Um programa, dito de uma forma simplificada, é um conjunto de instruções que juntamente irão ser executadas, com o objetivo de efetuarem uma determinada tarefa no computador[17]. A criação de um programa é feita através da escrita destas instruções, numa linguagem de programação, com as regras gramaticais (a sintaxe) e da semântica próprias da linguagem, existindo centenas de linguagens de programação diferentes[18].
Isto significa que um programador principiante tem de aprender[6][19][20][21]:

- uma linguagem de programação, inclusive a sua sintaxe e semântica;
- os conceitos básicos de programação e a lógica computacional (ciclos, condições, funções, variáveis, declarações, etc.);
- a resolver problemas utilizando esta linguagem de programação, incluindo a identificação, estruturação e divisão do problema que se pretende resolver;
- a encontrar e reparar falhas no programa em construção, incluindo testar o programa e resolver os "*bugs*"presentes no mesmo;
- a lidar com erros ou valores inesperados no programa, ou seja, fazer tratamento de erros que possam surgir durante a execução do programa;
- a reutilizar código já existente ou construído pelo próprio programador, inclusive saber ler a documentação do código a ser reutilizado, assim como, saber escrever a sua própria documentação do código criado.

Para além disto (apesar de não ser obrigatório para programar), é cada vez mais popular e incentivado o uso de IDEs ("*Integrated Development Environments*"ou em português "Ambiente de Desenvolvimento Integrado") na construção de programas, visto facilitar a construção, compilação e correção de erros nestes. Sendo por vezes requerido aprender também a utilizar estas ferramentas para além das linguagens e dos conceitos de programação[6].

2.3.1 Problemas e dificuldades observadas em programadores principiantes

O número de problemas que os iniciantes enfrentam para aprender a programar são diversos, nomeadamente ao nível da aprendizagem de uma nova linguagem, assim como na aplicação desta para resolver novos problemas. Vários estudos [19][22][23][6][24] foram realizados com o intuito de determinar os conteúdos de programação de maior dificuldade para os alunos (a maioria alunos universitários) e identificar os problemas mais pertinentes durante a sua aprendizagem. A partir destes estudos, destacam-se os seguintes problemas e observações na aprendizagem dos programadores iniciantes:

- Problemas ao nível da sintaxe e semântica - tal como na aprendizagem de um nova linguagem falada, aprender uma nova sintaxe e semântica são tarefas difíceis. Por vezes, a sintaxe utilizada para descrever um determinado mecanismo funcional pode provocar confusão quanto à forma como o programa a irá executar (por exemplo, no ciclo "*while*", o seu funcionamento pode ser interpretado que a condição a testar é feita continuamente durante o bloco de instruções e não em cada iteração do ciclo);
- Não saber por onde começar - perante um problema não sabem quais os passos iniciais a tomar, mesmo tendo uma compreensão básica dos conceitos de programação e da sintaxe da linguagem a programar;

- Problemas em perceber e utilizar corretamente os conceitos funcionais de programação - ciclos, funções recursivas, apontadores, etc.;
- Programação feita linha por linha - em vez de pensarem dentro de um contexto global, na forma como resolver um determinado problema, fixam-se numa pequena parte do programa em construção;
- Falta de capacidades, conhecimentos ou experiência para resolver problemas, especificamente o uso de lógica;
- Problemas em projetar no futuro, de que forma o programa irá funcionar - os novos programadores, têm normalmente muita dificuldade na leitura e compreensão do funcionamento de um programa, o que por consequência, faz com que a previsão do comportamento do programa seja mais difícil;
- Pensar que programar é um processo direto, fixo e que existe uma solução única.

Porque razão existem estes problemas? As possíveis explicações dadas pelos investigadores são:

1. A própria natureza de programação faz com que seja difícil de aprender;
2. Os estudantes não estão devidamente preparados por falta de conhecimentos necessários para aprender e aplicar os conceitos de programação;
3. A forma como esta é ensinada, os materiais ou ferramentas educativas utilizados são insuficientes e inadequados para ensinar programação.

Das 3 descrições, a terceira será o tópico em foco neste trabalho, visto estar diretamente relacionado com o objetivo pretendido.

2.3.2 Métodos e ferramentas utilizados no ensino de programação

Existe um vasto conjunto de ferramentas e técnicas criadas especificamente para ensinar programação (ex: Scratch[1], Flowgorithm[7], Toontalk[10], entre outros), no entanto, a metodologia utilizada na maior parte das vezes é a que faz uso de materiais de ensino estáticos tradicionais, tais como, slides de apresentação, livros e apontamentos, ferramentas estas que não são ideais para descrever um processo dinâmico como o da programação[6].

Outro problema com esta metodologia, é o facto de transmitir a ideia de só existir um caminho e uma solução para criar um programa, não mostrando no entanto, que é possível criar programas com comportamentos iguais, mas com implementações diferentes, ou em que o código mostrado teve de sofrer várias alterações até chegar à versão apresentada[6].

Para resolver esta situação, uma das soluções é a observação de um programador experiente, o qual descreve o seu raciocínio durante a construção de um programa, demonstrando assim, que é comum alterar o seu código várias vezes, assim como corrigir eventuais erros e "*bugs*"encontrados, também fazem parte deste processo[6].

Ainda assim, este método só por si não é suficiente para a resolução do problema, devendo o mesmo ser considerado como uma ferramenta para ensinar o processo de programar e não um método de ensino completo.

Para além de ensinar corretamente o processo de programação, existe também a problemática da leitura e da interpretação do código de um programa, o qual dificulta a capacidade de prever de que forma um programa irá executar. Esta capacidade torna-se essencial para detetar e corrigir "*bugs*", como também estudar novos algoritmos ou programas já existentes[19][6]. O uso de técnicas ou ferramentas gráficas parecem mitigar este problema. Apresentam um código mais organizado e alguns deles conseguem também expor visualmente a execução de um programa, fazendo uso de animações das instruções executadas [19][21][6].

Existem também LVPs que permitem simultaneamente criar programas, assim como expor a estrutura do programa numa representação gráfica organizada.

Uso de LVPs para ensinar programação

O interesse no uso de LVPs não se limita apenas a facilitar a criação de programas, possibilitando ainda aceder com mais facilidade à aprendizagem destas linguagens visuais. Existem estudos[22][8][25] que mostram que LVPs como Scratch, Forms/3 e Hank foram utilizadas com sucesso para ensinar aos novos programadores os conceitos base de programação. As consequências ou as vantagens observadas em cada uma destas linguagens foram as seguintes:

- Forms/3 - o estudo em [8] demonstrou que Forms/3 apresentou ser uma linguagem mais fácil de aprender a utilizar comparativamente com outras duas linguagens de texto, nomeadamente Pascal e APL[26]. Para além disto, a performance dos estudantes na resolução de problemas revelou maiores resultados quantitativos com resoluções corretas comparados com os das linguagens de texto.
- LabVIEW - uma LVP baseada em diagramas, teve uma performance bastante positiva versus a linguagem textual C [8]. O teste foi feito com duas equipas de programadores profissionais, onde foi utilizado uma das duas linguagens e dadas as mesmas condições às duas equipas para completar o projeto ou seja, o mesmo financiamento, os mesmos requisitos e o mesmo limite de tempo. Devido à natureza da linguagem LabVIEW, a equipa que utilizou a mesma, conseguiu implementar todas as funcionalidades do projeto juntamente com os seus clientes, os quais conseguiam ler com facilidade a linguagem do código. Como consequência, esta equipa cumpriu todos os requisitos pedidos, além de outros não planeados, dentro do prazo estipulado. Por conseguinte, a equipa da linguagem de C não utilizou uma metodologia de desenvolvimento de software tão pro-ativa como a que foi utilizada pela equipa anteriormente referenciada. O teste realizado não foi feito dentro dos mesmos parâmetros, no entanto, foi possível observar o potencial do uso de uma LVP para desenvolver software, podendo esta ser lida por clientes sem experiência em programação.
- Scratch - esta linguagem foi utilizada num estudo[22] durante um curso de introdução à Informática, com o objetivo de facilitar o ensino de programação a estudantes completamente inexperientes. A forma como os programas são criados nesta linguagem, arrastando as componentes do programa a partir de uma lista e encaixando como peças de puzzle, aparenta ser mais fácil para programar, visto que o editor Scratch permite ao programador visualizar as listas de todos os ícones disponíveis na linguagem e de arrastar cada ícone ao código do seu programa, não sendo necessário memorizar ou escrever totalmente a sintaxe da linguagem Scratch. Existe também uma LVP semelhante ao Scratch, chamada de Blockly[27], sendo atualmente utilizada no website educativo Code.org[28] para ensinar programação a milhões de estudantes de diversas línguas.
- Hank - uma LVP criada especificamente como alternativa às linguagens de texto para aprender programação[25]. Esta linguagem foi utilizada com sucesso para ensinar programação, mostrando-se bastante acessível e fácil de usar, mesmo por estudantes principiantes em programação. Os investigadores concluíram que o motivo deste sucesso deve-se à sintaxe gráfica simples de utilizar, no

entanto, argumentam que o uso de LVPs no ensino de programação devem ser utilizadas como uma ferramenta auxiliar, sendo necessário materiais de suporte e apoio dos professores para a certificação de que os conceitos são devidamente apreendidos.

- Toontalk - esta linguagem, no estudo referido em [29], teve também sucesso no ensino de conceitos de programação a crianças que frequentam jardins-de-infância. O ensino do uso e funcionamento da linguagem foi feita através de puzzles disponíveis nesta ferramenta e a maioria das crianças conseguiram resolver os primeiros 25 puzzles sozinhas, utilizando pela primeira vez o Toontalk, durante os testes. Alguns dos puzzles são considerados complexos para programadores principiantes, no entanto, as crianças conseguiram resolver alguns destes puzzles, concluindo-se que estavam efetivamente a aprender programação[29].

2.3.3 Jogos para aprender a programar

A maioria dos jogos de programação existentes têm como objetivo ensinar conceitos de programação às crianças. Nesta secção será feita a descrição e análise de três jogos, com a intenção de determinar quais os conceitos de programação aprendidos e aqueles que não foram assimilados, assim como, determinar de que forma estes jogos preparam o jogador no uso de conceitos base de programação.

NOTA: Nenhum dos 3 jogos descritos nesta secção foi completado, tendo ficado alguns puzzles por resolver devido a restrições de tempo ou por outros motivos(ver Toontalk), fazendo com que as observações feitas pelo autor possam induzir a erro.

ToonTalk

Partindo da descrição feita da linguagem ToonTalk na secção 2.1.2, a componente didática é composta por 74 puzzles [30] servindo como tutorial da ferramenta. Cada puzzle ensina algo de novo, seja a introdução de um novo objeto ou a fazer uma determinada atividade. Visto que esta aplicação foi feita a pensar nas crianças, o jogo conta uma história de um marciano que despistou a sua nave espacial e o jogador tem o objetivo de reparar a nave espacial, completando um conjunto de tarefas (os puzzles). O marciano serve de guia ao jogador, explicando quais as tarefas que o jogador terá de completar e de que forma funcionam os novos objetos disponíveis em cada tarefa.

Devido à idade da aplicação (lançado em 1999 e com a última atualização feita em 2007), alguns dos puzzles foram difíceis de resolver, devido a maus controlos de interação com os objetos. Um exemplo disto, verificou-se no puzzle #13 [30] - cujo objetivo é a criação de uma caixa com 10 zeros, dada uma caixa incluindo outra com 5 zeros e um buraco vazio. Para selecionar uma caixa com um número, neste caso o zero, requer-se muita precisão com o ponteiro do rato, o cursor tem de estar exatamente entre o fim da caixa e o número, o que faz com que por vezes se selecione o número dentro da caixa e não a caixa inteira. Outro problema encontrado ao resolver cada puzzle do jogo, teve a ver com o espaço de trabalho ser demasiado pequeno em relação aos objetos utilizados, fazendo com que as caixas com vários números sejam grandes demais para visualizar tudo, novamente outro problema do puzzle #13, sendo necessário "deslizar"a caixa ao longo do ecrã para encontrar o buraco ou o número desejado, dentro da mesma.

Relativamente à componente de aprendizagem, o Toontalk ensina:

- A programar funções/processos - ensina que dentro de cada casa há um conjunto de tarefas a serem

executadas e que cada função pode enviar e receber mensagens ao utilizar-se ninhos e pássaros(dados).

- A programar ciclos - nos puzzles com robôs programáveis, o número de ações feitas pelo robô ou o número de uso de um determinado objeto pelo robô, (por exemplo a varinha mágica) é limitado ensinando ao jogador o conceito de ciclos.
- A programar instruções condicionais - ao programar um robô, o objeto dado a este representa a condição para a sua ativação, ou seja, só se ativa se existir um objeto no chão da casa semelhante ao utilizado no seu treino, permitindo assim, programar uma casa utilizando vários robôs com um comportamento equivalente às declarações "*if-then-else*"ou "*switches*";
- Ensina um pouco o uso de *"arrays"* - ao utilizar-se caixas para guardar números, letras e outros objetos;

Resumindo, o jogo ensina alguns conceitos base de programação como ciclos, funções (de certa forma) e instruções condicionais. Apesar de não ser ensinado nos puzzles do jogo, é também possível utilizar programas recursivamente de duas formas: guardando o robô no caderno com as instruções a reutilizar e treinando outro para aceder e ativar este robô guardado ou usando a varinha para copiar o próprio robô a utilizar recursivamente.
No entanto, a natureza abstrata da linguagem faz com que alguns conceitos não estejam bem explicados (por exemplo: o uso de "*arrays*") e também não explora o uso de variáveis, à exceção de caixas ("*arrays*") que são utilizadas para armazenar e deslocar informação entre casas. Outro problema encontrado no jogo é o facto de não destacar bem a ideia de que um programa pode ser resolvido de diversas maneiras, visto que o conjunto de objetos e ferramentas disponíveis em cada puzzle serem limitados para resolver o problema.

Light-Bot

Light-Bot [31] é um jogo que envolve o controlo de um robô, numa grelha dividida por quadrados (ver figura 2.9), cujo objetivo é o de lhe acender a lâmpada que possuí na cabeça, em quadrados específicos (quadrados azuis).

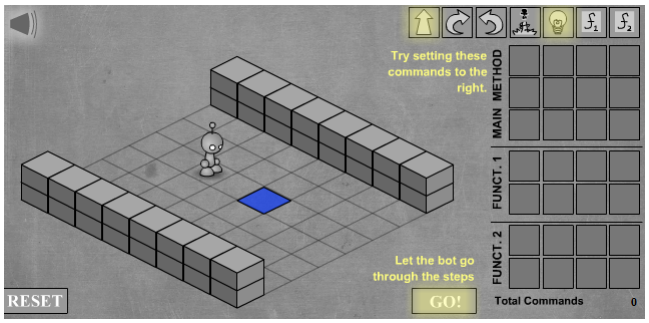


Figura 2.12: Imagem do jogo Light-Bot

Para tal, o jogador tem de criar o programa que controla o robô, colocando instruções numa lista, chamada "método principal". Ao seu dispor tem as seguintes instruções de controlo: andar para à frente, virar para a esquerda ou direita, saltar para a frente (podendo saltar para cima de um bloco ou para baixo), acender a lâmpada, assim como duas instruções especiais "f1"e "f2". Estas duas últimas instruções, são funções que

o jogador pode construir e utilizar para não repetir sequências de instruções, tornando-se essencial para resolver alguns dos níveis, visto que o número de instruções que se pode colocar na lista principal do robô é limitado.

A partir desta descrição e resolvendo alguns dos puzzles do jogo podemos determinar que o mesmo ensina:

- Que um programa é composto por um conjunto de instruções;
- Que o raciocínio para criar um programa exige a escrita das instruções a executar de forma resolver um determinado problema, no entanto, nem sempre o resultado obtido é o esperado (devido a um erro cometido);
- Que é importante observar o comportamento do programa para nos certificarmos de que o resultado é o pretendido;
- Reforça o uso de funções devido ao número limitado de instruções aceites pelo robô, especialmente em casos onde é necessário repetir várias vezes um conjunto de instruções;
- Que um problema pode ser resolvido de diversas formas.

Apesar de não se poder aplicar de forma prática no jogo, é possível criar funções recursivas. Como o jogo não contem instruções condicionais, não há maneira de parar uma função recursiva. Ao executar as funções, o robô fica preso num ciclo infinito.
Por fim, pode-se concluir que o jogo ensina alguns conceitos de programação importantes, dando maior destaque à construção e uso de funções, no entanto, não ensina o uso de variáveis de retorno de uma função. O uso de instruções condicionais, de variáveis, de ciclos e de *"arrays"* também não são ensinados, revelando-se uma lacuna na apresentação de conceitos essenciais para se iniciar a programar.

Jahooma's LogicBox

Tal como o Light-Bot, Jahooma's LogicBox [32] é um jogo de puzzle cujo grau de dificuldade vai aumentando em cada nível que for completado. O objetivo em cada nível é a construção de uma "caixa"(um programa), que recebe um valor input (uma "*string*") que terá de sofrer alterações de forma a cumprir um determinado objetivo. Para tal, são utilizadas outras caixas as quais recebem palavras e retornam o resultado consoante a direção das suas setas. Por exemplo, no primeiro nível o objetivo é a criação de uma caixa que apaga as primeiras 4 letras de uma determinada palavra e devolve o resultado (ver figura 2.10).

Cada caixa é representada por uma grelha, na qual se coloca outras caixas de forma a resolver cada puzzle.

À medida que o jogador vai avançando, as caixas criadas podem ser reutilizadas. Para cada nível, a maneira como se retorna o valor é dirigindo a palavra recebida para um dos 4 lados da caixa, onde cada lado é representado por uma cor diferente: verde, azul, amarelo e vermelho. Após construída a caixa, os lados utilizados correspondem às setas que estão dentro desse caixa, podendo-se deslocar as setas de forma a redirecionar a palavra ao longo do programa. Por exemplo, uma caixa construida num dos níveis com o nome "*Delete If Equal*"apaga a primeira letra de uma palavra com os primeiros caracteres repetidos e devolve o resultado para o lado verde, caso contrário, devolve para o lado vermelho. Essa caixa fica com

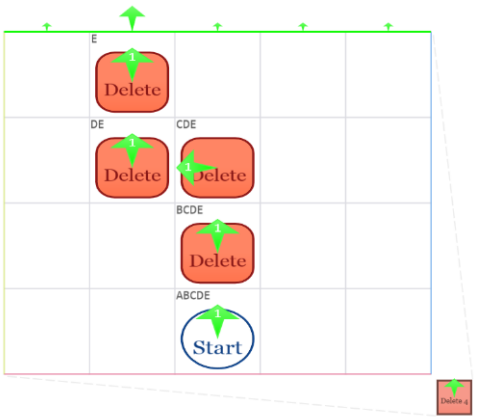


Figura 2.13: Primeiro nível do jogo Jahooma's LogicBox

duas setas: uma vermelha e outra verde que corresponde a esses lados. Simultaneamente, usando essas setas, é possível criar-se caixas condicionais em que cada seta corresponde a uma condição. Também é possível criar ciclos colocando duas caixas com as setas apontadas uma para a outra. Ao completar alguns dos níveis do jogo, determinou-se que o jogo ensina:

- Que um programa é composto por vários programas(programação modular);
- Que um programa recebe um valor e que após terminadas as suas tarefas devolve esse valor modificado;
- A utilizar instruções condicionais;
- Que um programa pode ser construído de diferentes formas;
- Que um programa pode utilizar ciclos para repetir várias vezes a execução de um conjunto de funções;
- O que é um ciclo infinito;
- A importância de testar o programa para que este tenha o comportamento desejado;
- A lógica utilizada para criar um programa.

A partir destes factos, podemos afirmar que o jogo ensina a maioria das bases da programação, destacando-se a criação e reutilização de programas feitos pelo jogador. Ensina também o conceito de funções condicionais e de ciclos, apesar de este último estar pouco explorado devido à estrutura do jogo. Os únicos aspetos que não são ensinados são o uso de variáveis e "*arrays*", o que faz com que o jogador só crie programas onde é utilizada apenas uma variável (especificamente, uma "*string*") desde o seu inicio e até ao fim da sua execução.

3

Proposta do jogo

3.1 Descrição da linguagem de programação do jogo

O objetivo deste trabalho é a criação de uma ferramenta auxiliar para jovens (3º ciclo), que consiga ensinar alguns dos conceitos básicos de programação e que transmita também uma ideia visual da estrutura e execução de um programa. Para tal, é necessário criar-se uma linguagem visual, acessível a qualquer utilizador, utilizando um paradigma familiar que proporcione uma ideia visual simples e clara da causa e efeito, durante a execução de um programa.
Ao analisarmos a forma como os programas de computador são executados, podemos constatar que grande parte das instruções dentro das funções, são geralmente lidas de forma linear (caso não tenham instruções que afetem o fluxo do código), percorrendo cada instrução linha a linha, até que se chegue à última. Por isso, a execução de um programa pode ser visto como a deslocação de um robô/pessoa numa casa, em que este entra pela porta de entrada e percorre um caminho ao longo desta casa, com a intenção de realizar uma ou um conjunto de atividades, utilizando os objetos presentes nesta e que ao terminar as suas tarefas sai da casa utilizando novamente a mesma porta. Tomando em consideração esta visualização como referência, podemos começar a construir uma linguagem de programação visual. Para determinar que tipo de objetos teremos de criar na nossa linguagem, é necessário identificar as componentes, instruções e os conceitos básicos utilizados num programa de computador:

- **Variáveis** - as variáveis num programa são essenciais para guardar, consultar, enviar e receber informação num programa. Este tipo de informação pode ser um número, uma letra, um conjunto de dados guardado numa lista, um valor booleano, etc. Visto que, se pretende ensinar as bases de programação, este trabalho apenas irá utilizar os tipos de dados mais comuns como "*int*", "*double*", "*boolean*", "*string*", "*char*"e a estrutura de dados "*array*";
- **Ciclos** - os ciclos são utilizados para repetir a execução de um conjunto de instruções várias vezes, até mesmo quando o número de repetições a executar é indeterminado. O controlo de execução destes é feita através de restrições impostas pelo programador, podendo as mesmas ser o número de vezes que o ciclo é executado ou quando ocorre algo especifico durante a execução das instruções;
- **Instruções condicionais** - são essenciais para possibilitar a execução de blocos de instruções em casos específicos, possibilitando a diversificação no modo como um programa é executado. Tal como nos ciclos, as instruções condicionais são controladas utilizando restrições. Por exemplo, uma restrição pode ser uma variável que contém um determinado valor ou uma expressão booleana;
- **Atribuição de valores e expressões** - apesar de atributo e de expressões serem dois conceitos completamente diferentes, ambos estão fortemente associados um ou outro. Quase todas as modificações feitas às variáveis num programa são realizadas através da atribuição (dado o nome em inglês de *assignment*) de novos valores ou de expressões. Uma expressão, dependendo do tipo de dados utilizado, pode ser um número ou uma expressão matemática, uma *"string"* ou uma concatenação de *"chars"* com outras *"strings"*, um valor booleano ou um expressão booleana ou até mesmo outra variável. Existem linguagens de programação em que o seu paradigma (nomeadamente o paradigma orientado a objetos) permite alterar valores num objeto, utilizando funções disponíveis no próprio chamados de métodos, não sendo necessário efetuar diretamente a atribuição de valores;
- **Funções e chamadas de funções** - as funções são essenciais para executar tarefas específicas dentro de um programa (ex: uma função que trata do cálculo de dois números ou uma que ordena os números guardados num *"array"*, etc.). Geralmente, uma função recebe um conjunto de variáveis de entrada a utilizar nesta, executa um conjunto de instruções e retorna um valor do resultado, após terminada a sua execução. Existem também casos de funções que não recebem ou nem retornam qualquer tipo de informação. Existindo uma função, também é necessário conseguir-se referenciar a mesma passando os parâmetros a utilizar nesta, ao que se dá o nome de chamada de função;
- **Valor de retorno da função** - o valor devolvido no final pela função, ou seja, é um elemento geralmente utilizado em funções, à exceção das que afetam de alguma forma o programa externamente (alterando variáveis globais como por exemplo).

3.1.1 Execução de um programa e descrição dos ícones da linguagem visual

Tendo em conta esta informação, podemos então começar a criar uma linguagem visual. Começando com as descrições da representação de um programa e de como este será executado visualmente, cada casa está ilustrada num plano 2D e cortada na vertical de forma a conseguir-se visualizar o seu interior(ver figura 3.1). O inicio da execução de um programa é representado por um robô, que entra pela porta principal da casa e a execução corresponde à deslocação deste pela casa. De forma a evitar ambiguidades na maneira como o programa é lido e executado, o robô apenas interage com os objetos da casa ao deslocar-se da esquerda para direita. Ao terminar o programa, desloca-se da direita para esquerda ignorando todo o conteúdo da casa até que volta a sair pela porta principal (a saída do programa). O percurso do robô depende dos objetos existentes na casa, sobre os quais terá de interagir durante o seu movimento. O fim da execução de um programa ocorre em duas situações: o robô não encontra mais objetos no seu caminho ou depara-se com um objeto que interrompe a execução.

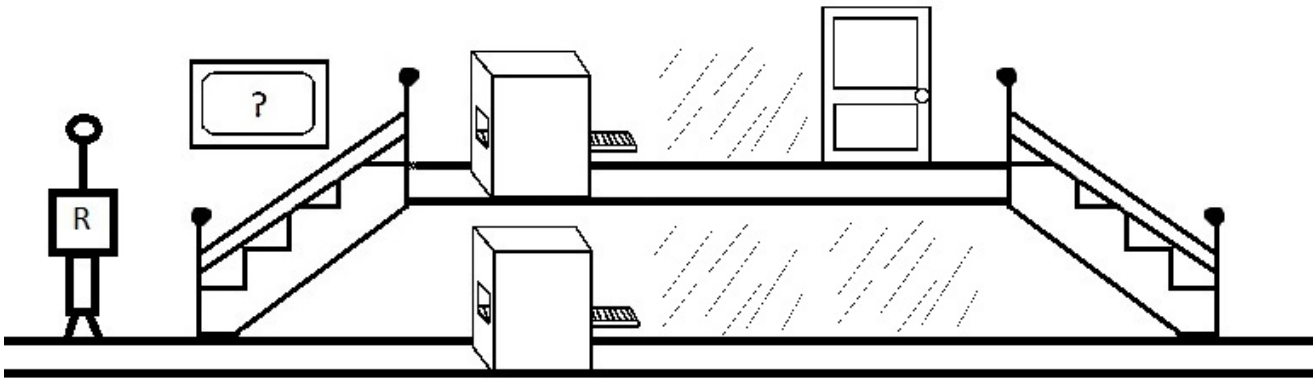


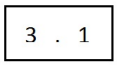
Figura 3.1: Exemplo de um programa casa simples

Estando definidas as regras de deslocação e de interação do robô, podemos então começar a definir os objetos que compõem a linguagem do jogo, começando com as variáveis.
Visto que o robô irá movimentar-se ao longo de uma casa, as variáveis têm de ser declaradas e mais tarde utilizadas, no resto do programa. Em termos visuais, a declaração de uma variável é fixa, mas o seu uso ao longo da casa terá de ser móvel. Portanto, uma variável pode ser representada por um objeto, que o robô apanha e guarda no seu inventário (declaração da variável) permitindo que este possa ser utilizado no resto do programa. Para evitar complicações na visualização de variáveis, não é possível utilizar variáveis declaradas numa casa dentro de outras funções, a não ser que estas sejam utilizadas como argumentos da função.
Os tipos de objetos que o robô pode apanhar também diferem consoante o seu tipo:

- *"Char"* - será representado por uma folha de papel contendo apenas uma letra/símbolo;

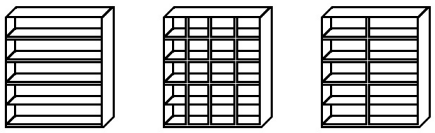
- *"String"* - será representada por um livro constituído por um conjunto de páginas com uma letra, ou seja, composto por vários *"Chars"*;

- *"Int"* ou número inteiro - será representado por um quadrado com um número registado dentro de si;



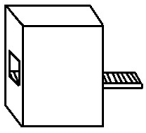
- *"Double"* ou número de virgula flutuante - será representado por um retângulo contendo dois dígitos numéricos e um ponto entre estes;

- *"Boolean"* - será representado por uma lâmpada, no qual apenas poderá guardar-se um de dois estados possíveis: ligado (*true*) e apagado (*false*);



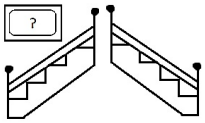
- *"Array"* - será representado por um armário. O *design* do armário pode ser utilizado para diferenciar o tipo de objetos que o mesmo guarda.

Após estarem definidos os objetos que o robô consegue apanhar e utilizar, é necessário também existir um objeto que altere a informação armazenada nestes objetos, ou seja, que faça a atribuição do resultado de uma expressão a um objeto escolhido. Esta ação pode ser representada por uma máquina (contendo uma expressão previamente configurada pelo programador), na qual o robô insere o objeto que pretende alterar e que, após terminado o seu funcionamento, obtêm-se de volta o mesmo objeto com um novo valor.



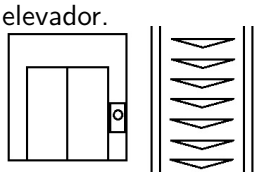
O tipo de máquina pode ser representada conforme o tipo de objeto a alterar. Por exemplo, no caso das variáveis *"Chars"*, utiliza-se uma máquina que carimba uma nova letra, no das *"Strings"* uma impressora, nos *"Ints"* é utilizado uma calculadora e nos *"Doubles"* um computador, etc. Dentro de cada máquina é utilizada uma expressão, construída com objetos já definidos na casa, (dependendo do tipo da máquina) com números, caracteres, letras, etc. As expressões definidas dentro de uma máquina são construídas de forma idêntica à das linguagens de programação, com a exceção das variáveis, que são representadas por objetos gráficos e não texto. A regra de utilização dos objetos disponíveis, ao construir expressões, é igual à regra das chamadas de função. Isto quer dizer, que apenas os objetos definidos e guardados anteriormente podem ser utilizados.

Para a definição de blocos de instruções condicionais podemos representa-las como caminhos alternativos pela casa, controlados por expressões que ditam os casos em que o robô utiliza estes caminhos.
Para a condição *"if"*, no caso da sua expressão condicional ser verdadeira, o robô tem de utilizar um caminho diferente de forma a executar as instruções contidas dentro do *"if"* e, caso contrário, segue o caminho pré-definido (o *"else"*). Após terminado o *"if"* e o *"else"* ambos os caminhos juntam-se e continuam o caminho original da casa.
Este comportamento pode ser representado por um par de escadas, nas quais o robô sobe para percorrer um caminho alternativo pela casa, conforme uma restrição, tendo que descer de novo outras escadas, para retornar ao caminho original da casa. Caso contrário, ignora as mesmas utilizando o caminho normal, podendo inclusive definir um segundo caminho por baixo do "*if*", que representa o "*else*".



Uma vantagem de utilizar esta representação gráfica é que ao definir-se o caminho do *"if"*, o caminho do *"else"* é definido em simultâneo. A distinção entre um e outro resume-se apenas ao andar em que estes se encontram na casa, um em relação ao outro, sendo os limites do *"if"* e do *"else"* marcados por um par de escadas (uma escada para subir e outra para descer). A condição que define o comportamento do robô é dado através de uma expressão escrita num cartaz afixado perto das escadas que informa se este poderá subir.

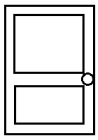
Partindo deste conceito de andares, o *"switch"* pode ser representado por múltiplos andares, onde um elevador transporta o robô para um dos andares que cumpre uma condição do *"switch"*. No final deste andar, terá de retornar ao caminho de origem utilizando um escorrega. A placa que identifica o andar em cada piso, em vez de conter um número, guarda a expressão da condição do *"switch"* daquele piso. O rés-de-chão corresponde à condição *"default"* do *"switch"* que executa sempre caso o robô não utilize o elevador.



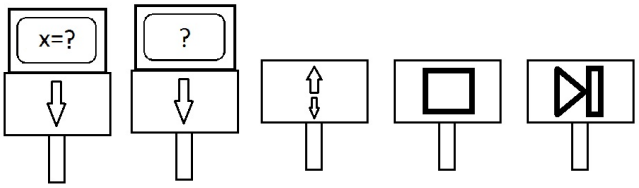
O *"return"* da função pode ser definida por uma parede (vista lateralmente) que previne a continuação do caminho do robô e faz com que este mude de direção, regressando assim, à porta inicial da função. Opcionalmente, esta parede pode conter informação do objeto que o robô tem de levar consigo ao sair da função, sendo este objeto o valor de retorno da função.

Para as chamadas de função, uma simples porta na parede pode ser usada para representar esta ação.
O programador pode utilizar uma casa construída anteriormente e aplicá-la como uma função. Para tal, é necessário que o programador selecione os objetos que pretende que o robô transporte para dentro desta porta, ficando o resto dos objetos detidos temporariamente, até que o mesmo retorne. Isto faz com que o robô só possa transportar objetos que tenham sido apanhados antes de ter alcançado a porta. No caso dos objetos transportados (os parâmetros) serem alterados de alguma forma dentro da porta da função, estes têm de ter os mesmos valores, à priori, à entrada e à saída da porta. Em termos visuais, este acontecimento

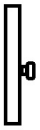
chamado de passagem por valor, é mostrado ao programador através da animação do robô a levar cópias destes objetos pela porta, descartando as cópias, ao terminar a função chamada. Após terminada a função, o robô pode guardar o valor de retorno desta, caso exista, num dos objetos guardados no seu inventário e continuar o seu caminho pela casa.



Para representar o processo dos ciclos visualmente, escolheu-se um íman suspenso que se desloca ao longo de uma linha de carris, com a capacidade de elevar e transportar o robô. Isto faz com que o íman consiga colocar o robô no principio ou no fim do ciclo (sendo este delimitado pela linha de carris) independentemente da sua posição. A razão da escolha deste mecanismo está relacionado com as instruções especiais *"break"* e *"continue"* que interrompem e avançam o ciclo, respetivamente, não sendo fácil representar estas ações no mundo real. Ou seja, o robô em qualquer posição dentro do ciclo poderia de imediato interromper e sair deste ou retornar à posição inicial do mesmo, ignorando os restantes objetos. Para representar estas duas instruções especiais escolheu-se dois sinais com setas a indicar o transporte do robô, um com uma seta em direção ao inicio do ciclo e outra para o final. A expressão que controla a execução do ciclo é representada por um ecrã contendo esta informação, conforme o robô passa por este é feita a verificação da condição do ciclo.



Para identificar a casa construída e a posição inicial do robô ao executar o programa, representa-se por uma porta de entrada, sendo esta vista lateralmente ao contrário das portas de chamadas de função já descritas. Neste objeto poderá-se também definir os objetos que o robô terá no inicio da execução do programa, os argumentos, podendo estes serem utilizados dentro da casa. E finalmente, também é possível determinar que tipo de objeto (só o tipo) o robô terá de trazer consigo ao terminar o programa visto que o robô deverá sempre regressar e sair da casa utilizando esta porta.



Estando quase todos os ícones da linguagem visual definidos, é necessário definir objetos vazios na casa de forma a não forçar o programador a preencher cada andar com objetos (usáveis pelo robô) e permitir um número diferente de objetos em andares diferentes (exemplo: ver as escadas da figura 3.1). Para representar um objeto vazio escolheu-se uma parede riscada de forma a indicar que não existe um objeto nesse espaço da casa. Estes ícones correspondem aos locais da casa em que é possível colocar um objeto (tendo um piso para tal).

Com os ícones criados o dicionário de ícones da linguagem está representada na figura 3.2, contendo as imagens dos ícones e os nomes identificadores correspondentes, e a gramática de ícones da linguagem está representada na figura 3.3, em que faz uso da mesma notação utilizada no livro "Visual Languages and Visual Programming"para descrever uma gramática de ícones.

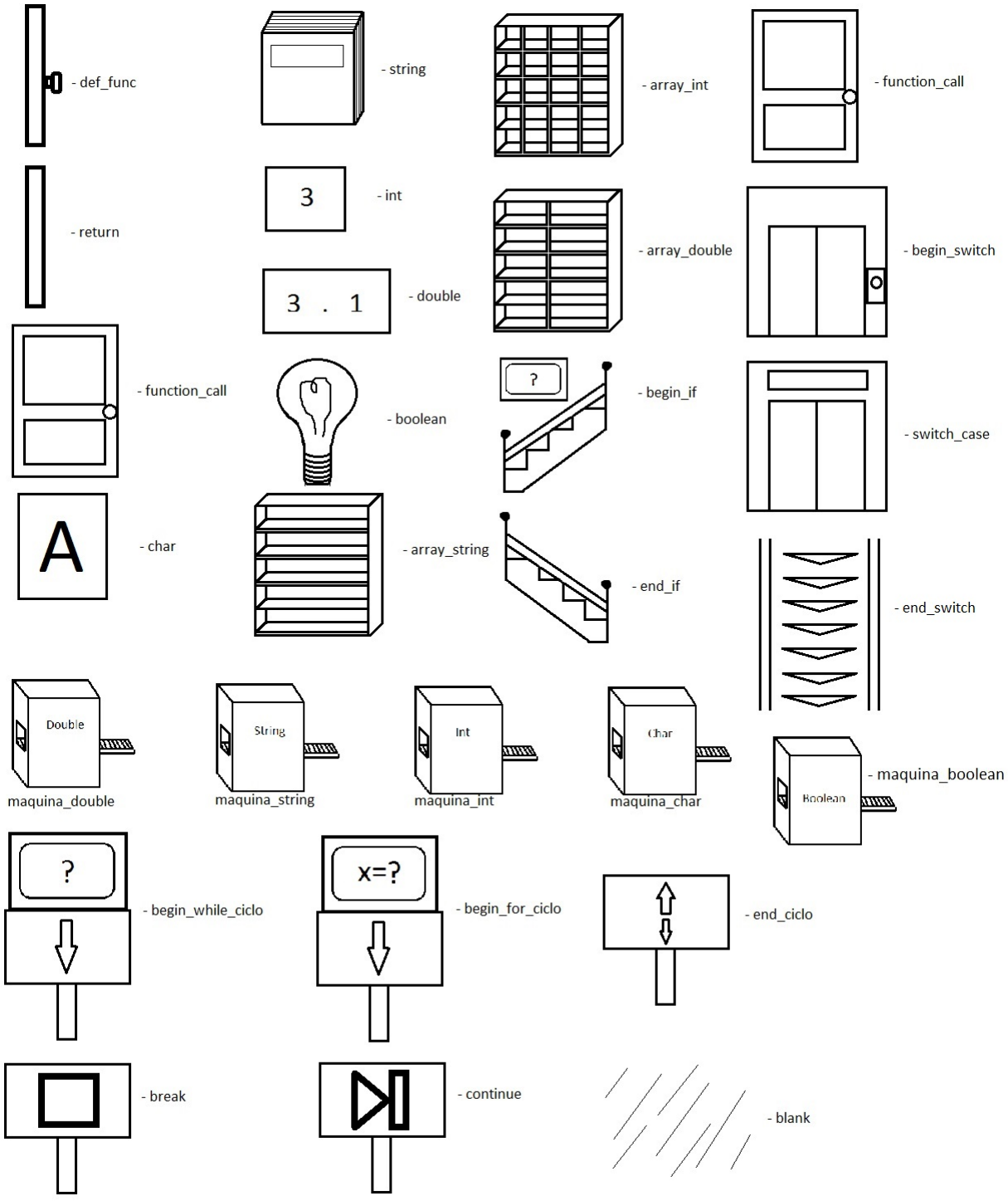


Figura 3.2: Dicionário de ícones da linguagem visual do jogo

```
<casa>            ->  start_function & <objectos>

<objectos>        ->  <objecto> & <objectos> | <objecto> | <objectos_condic>
<objecto>         ->  <declarar_var> | <atribuir_var> | <objectos_ciclo> |
                        function_call | blank | return | break | continue

<objectos_condic> ->  <if> | <switch>
<objectos_ciclo>  ->  <for_ciclo> | <while_ciclo>

<declarar_var>    ->  int | double | char | string | boolean | array_string | array_int
                        | array_double
<atribuir_var>    ->  maquina_double | maquina_string | maquina_int |
                        maquina_char | maquina_boolean

<if>              ->  <if_else_objectos> & end_if
<if_else_objectos>->  <objecto> & <if_else_objectos> & <objecto> | <separador_if>
<separador_if>    ->  <objecto> ^ begin_if & <objecto>


<switch>          ->  switch_case & <switch_corpo> & switch_end
<switch_corpo>    ->  <objecto> & <switch_corpo> & <objecto> | <switch_default> |
                        <objecto> & switch_end ^ <switch_case>
<switch_case>     ->  switch_case & <objecto> & <switch_corpo>
<switch_default>  ->  <objecto> & switch_end ^ begin_switch & <objecto>

<for_ciclo>       ->  begin_for_ciclo & <objectos> & end_ciclo
<while_ciclo>     ->  begin_while_ciclo & <objectos> & end_ciclo
```

Figura 3.3: Gramática de ícones da linguagem visual do jogo

(NOTA: Esta gramática contém alguns problemas quanto às combinações de objetos permitidos, assim como algumas observações que se destacam de outras linguagens de programação:

1. Problemas:
   (a) Esta gramática permite utilizar os objetos *"continue"* e *"break"* fora dos ciclos;
   (b) Força um número igual de objetos nos blocos *"if"/"else"*, mas no *"switch"* isto já não acontece para todos os casos.
2. Outras observações:
   (a) Força utilizar utilizar o objeto "start_function"no inicio, tal como pretendido;
   (b) É possível inserir qualquer tipo de objetos dentro de ciclos, tal como pretendido;
   (c) É possível construir blocos de objetos vazios (só com objetos "blank"), tal como pretendido;
   (d) Objetos *"if"/"else"* não podem ser colocados dentro de blocos *"if"/"else"*, tal como pretendido(ver secção 5.1 para a justificação).

Uma solução para mitigar os dois problemas enunciados é fazer-se uso de um editor que restrinja a colocação de objetos de forma a evitar estes problemas.)

3.2 Construção de um programa e leitura dos objetos

Após a descrição da linguagem visual do jogo, é necessário determinar-se de que forma se irá implementar a escrita de um programa. Visto que o jogador terá de colocar objetos dentro de uma casa de forma

a assegurar a legibilidade do programa, queremos que os objetos estejam distanciados entre si e não "atafulhados"sem que se consiga identificar quais os objetos guardados dentro da casa. Uma resolução para este problema é dividir a casa numa grelha em quadrados, onde cada quadrado corresponde a uma possível posição para colocar um objeto. Para identificar cada quadrado serão utilizadas coordenadas X e Y em que X identifica o número da coluna de quadrados da casa e Y o número da linha da casa, por exemplo, as coordenadas X=0 e Y=0 representam o quadrado mais abaixo e mais à esquerda da casa sendo a única posição onde é colocada a porta de entrada da casa(ver figura 3.4).

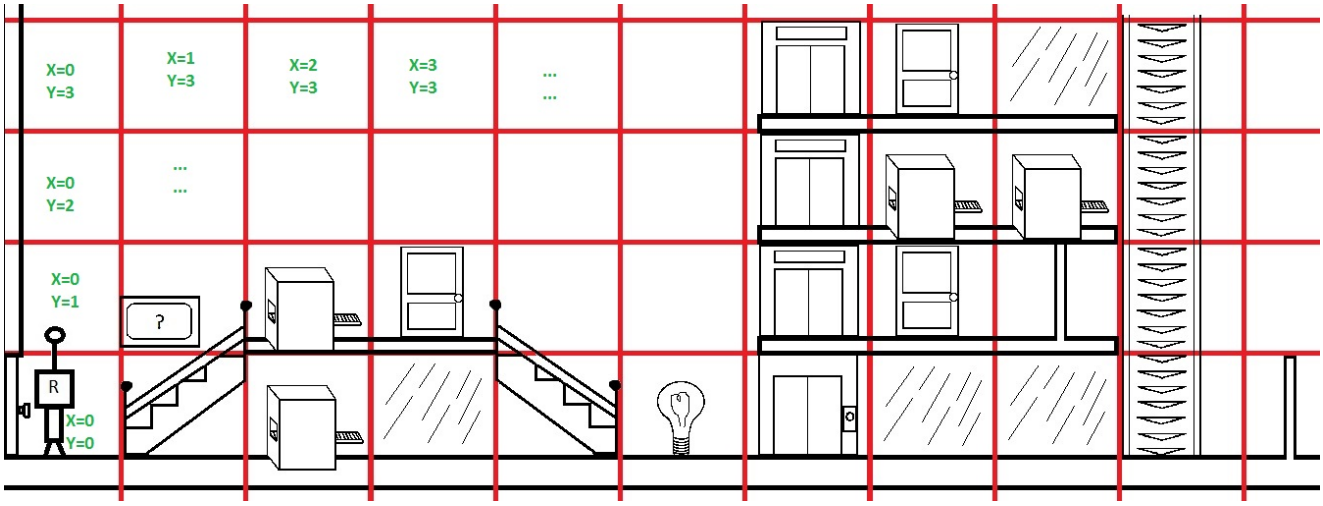


Figura 3.4: Exemplo de um programa visual do jogo

A escrita de um programa é feita através da colocação de objetos a partir de uma lista, arrastando um destes até a uma posição válida na casa, ficando a imagem do objeto nesse quadrado. Do ponto de vista do jogo, o número de objetos disponíveis em cada nível pode ser restringido conforme o tópico que se pretende ensinar ao jogador. Cada objeto contém informação dependendo do seu tipo, podendo esta ser modificada e consultada por um elemento da interface do jogo (uma janela ou um *pop-up*, p.ex.). Assim sendo, isto implica que portas, escadas, elevadores, paredes, máquinas ou qualquer outro tipo de objeto contenha informação não visível graficamente. A razão desta escolha deve-se ao espaço limitado ocupado por cada objeto na casa. Em alguns casos é possível mostrar graficamente a informação guardada no objeto, mas isso implica também limitar o tamanho da informação (mais especificamente o número de caracteres) que o jogador consegue guardar em cada objeto de forma a cumprir o espaço atribuído a cada posição da casa.

Tendo uma casa construída para conseguir-se executar o programa que esta descreve, o primeiro passo necessário é efetuar a leitura dos objetos gráficos presentes nesta. A leitura dos objetos de uma casa é feita na direção horizontal (seguindo as coordenadas X) é sempre feita da esquerda para à direita, com a exceção dos objetos que estão dentro dos blocos "*if*"/"*switch*", nomeadamente:

- *"Else"* - que regressa ao ícone inicial do "*if*"para ler os ícones por baixo das escadas;
- *"If-Else"* - primeiro lê os objetos no andar de cima e só depois os objetos que estão dentro do bloco *"else"*;
- *"Switch"* - lê o ícone de inicialização do *"switch"*, avança imediatamente para o último andar do *"switch"* prosseguindo a leitura aos "ziguezagues";

- *"Switch Case"* - que ao ler cada andar retorna à coluna do inicio do *"switch"*;

Na direção vertical (coordenadas Y) só dentro dos blocos *"if-else"* e *"switch"* é que é feita a deslocação. Utilizando o programa da figura 3.4, podemos ver a sua leitura dos ícones enumerada na figura 3.5.
A leitura dos ícones de um programa termina assim que for encontrado um quadrado vazio no rés-do-chão da casa. Um exemplo do código na linguagem de programação Python correspondente à casa da figura 3.5 seria:

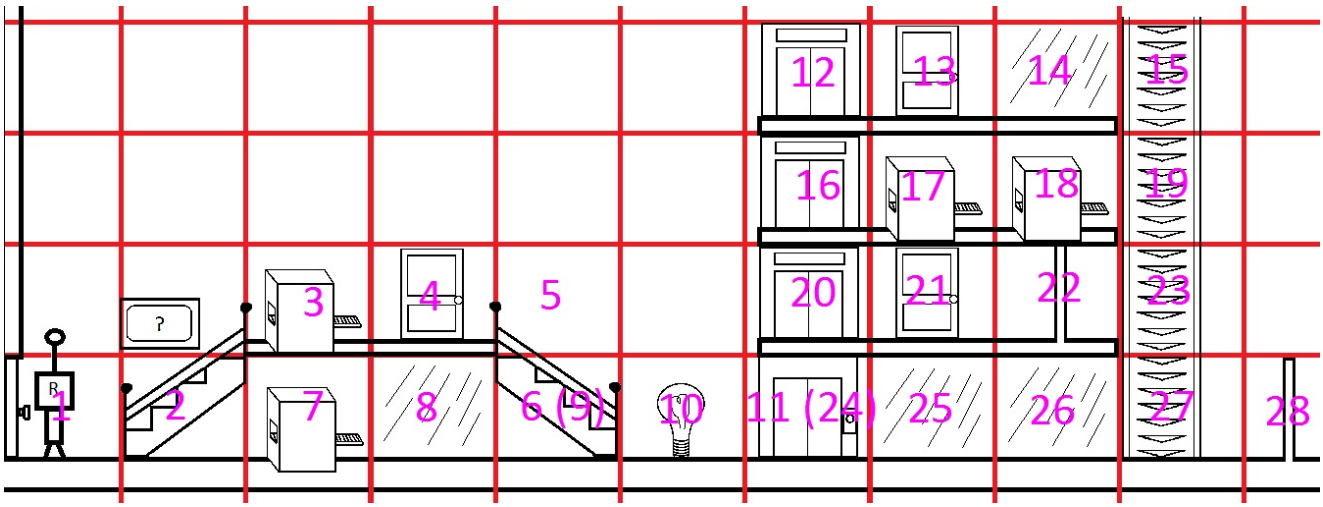
```
def foo(a,b):
  if a>b:
    a=a-b
    foo(a,b)
  else:
    a=a*b

  igual= a==b

  if igual:
    foo2(a,b)
  elif a<b:
    a=a*b*b
    b=a-b
  elif a>b:
    foo2(b,a)
    return a

  return b
```

(NOTA: A linguagem de Python não contém declarações *"Switch"* tendo sido utilizado como substituto *"ifs"* encadeados (a notação *"if-elif...elif-else"*) na geração do código Python.)



Os números dentro de parêntesis correspondem a ícones que foram lidos duas vezes.

Figura 3.5: Ordem da leitura do programa da figura 3.4

4

Implementação do protótipo do jogo

Para a implementação do protótipo, optou-se utilizar uma linguagem de programação orientada a objetos, visto que a linguagem visual do jogo é composta por objetos familiares ou comuns. De todas as linguagens existentes dentro deste paradigma, escolheu-se Java, visto ser a linguagem que o autor desta dissertação tem mais experiência em utilizar. A linguagem de programação escolhida para a geração de código foi Python, devido à sua simplicidade na escrita e na leitura de programas.
Conforme as descrições da linguagem visual feitas no capitulo anterior e considerando-se todas as características desejadas no jogo, o protótipo do mesmo terá de ter:

- Uma interface visual - Com o intuito de ver os objetos existentes dentro de uma casa, onde será feita a construção do programa, assim como, a animação da execução do programa;
- A casa - Estrutura onde se irá guardar os objetos, regras, testes, e outros;
- Os objetos da casa - São os objetos que compõem o programa, que guardam informação distinta conforme o tipo de cada objeto;
- As regras da casa - Restringem a colocação dos objetos na casa de forma a evitar erros de compilação, problemas de leitura do programa e limitações gráficas da linguagem;

- O editor da casa - Permite colocar os objetos na casa consoante as regras estabelecidas e alterar a informação contida dentro destes;
- O leitor de casas que gera código - Lê todos os objetos presentes dentro da casa e faz simultaneamente a geração do código de uma linguagem de programação textual, Python neste caso;
- O interpretador do código gerado - Executa o código gerado pelo leitor da casa;
- A animação da execução do programa - Mostra as posições da casa conforme os valores dados à execução do programa, com o objetivo de observar a execução do mesmo, ao longo da casa;
- A capacidade de efetuar testes a uma casa - Para testar se a casa construída cumpre um determinado objetivo estabelecido;
- A capacidade de abrir e guardar casas - Guarda casas anteriormente criadas, podendo estas serem referenciadas por outras casas, a partir das chamadas de funções já existentes.

A estrutura ou arquitetura do protótipo pode ser vista no diagrama da figura 4.1. Neste capitulo será feita a descrição detalhada do trabalho realizado na construção do protótipo do jogo, das suas componentes, das bibliotecas utilizadas e dos algoritmos/sistemas criados para o funcionamento correto do mesmo.

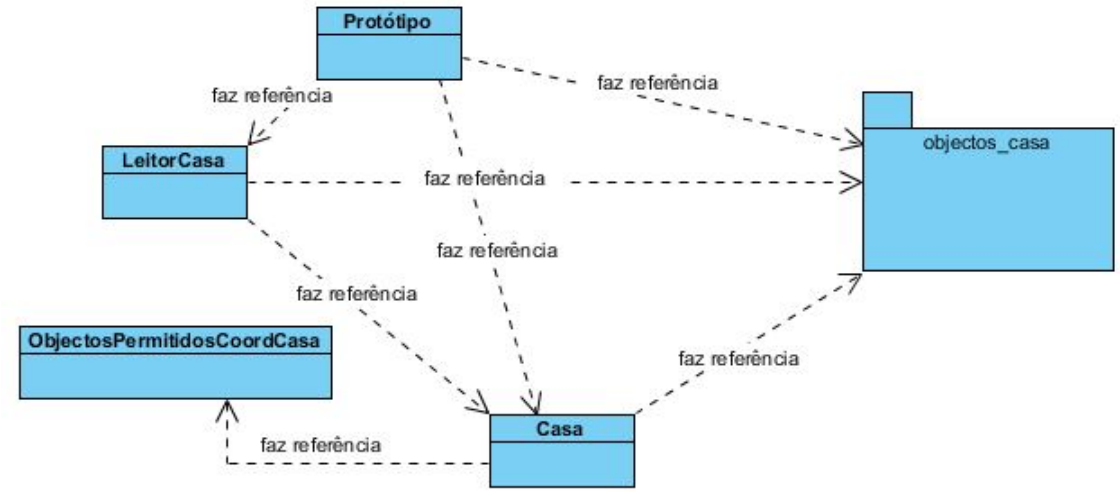


Figura 4.1: Arquitetura do protótipo do jogo

4.1 Pacote objectos_casa

Em primeiro lugar, é necessário definir devidamente as características de todos os objetos utilizados na construção de um programa. Este pacote contém a implementação de todas as classes de objetos da LVP do jogo, assim como, uma pasta "imagens"com todas as imagens dos ícones dos objetos, a componente gráfica da linguagem. As relações existentes dentro do pacote **objectos_casa** pode ser visto no diagrama da figura 4.28, localizada no fim deste capítulo.

4.1.1 ObjectoCasa

A classe **ObjectoCasa** é a implementação abstrata de um objeto da casa, não estando associado a um ícone gráfico específico. Em termos funcionais, este objeto corresponde a um espaço vazio da casa. Esta classe é necessária para se conseguir guardar qualquer tipo de objeto dentro da casa, aproveitando-se assim o polimorfismo de objetos disponibilizado na linguagem Java. Todos os outros objetos fazem extensão a esta classe, o que significa que herdam também as variáveis e métodos. O resumo das variáveis e métodos presentes na classe **ObjectoCasa** pode ser vista na figura 4.2 seguido da descrição de cada item.

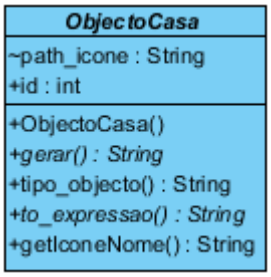
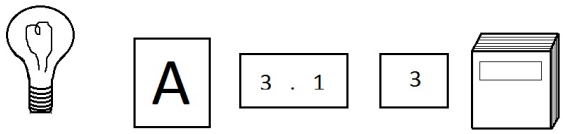


Figura 4.2: Diagrama da classe **ObjectoCasa**

- Variáveis:
  - String **path_icone** - variável String onde será guardado o caminho da imagem gráfica do objeto;
  - int **id** - variável com o número único identificador do objeto disponível/declarado dentro da casa;
- Métodos:
  - **ObjectoCasa()** - o construtor do ObjectoCasa, inicializa as variáveis descritas anteriormente;
  - **gerar()** - método abstrato que devolve a String do código Python correspondente ao objeto da casa;
  - **tipo_objecto()** - o método que devolve a String com o tipo de objeto, para permitir a identificação dos diversos objetos guardados na casa. Este método não é abstrato com o propósito de se conseguir identificar os espaços vazios da casa, sendo estes identificados com a String **"null"**.
  - **to_expressao()** - método abstrato que devolve a String, com a informação do objeto, para que esta possa ser utilizada dentro de uma expressão;
  - **getIcone()** - o método que devolve a String com o nome da imagem do ícone do objeto. No caso do ObjectoCasa devolve uma imagem totalmente branca.

4.1.2 Boolean_casa, Char_casa, Double_casa, Int_casa e String_casa



As classes dos objetos referidos em cima, representam respetivamente os tipos de dados descritos no capitulo anterior. Estas classes contêm juntamente com o seu nome "_casa"de forma a evitar conflitos e/ou enganos com a linguagem Java, visto que esta também utiliza os mesmos nomes, para os mesmos tipos de dados presentes nesta.
Todas têm os mesmos métodos e variáveis (à exceção do **Boolean_casa** que em vez de **setValor()** tem **switchValor()** que ao ser chamado troca o valor Booleano existente para o valor oposto) como se pode observar na figura 4.3.

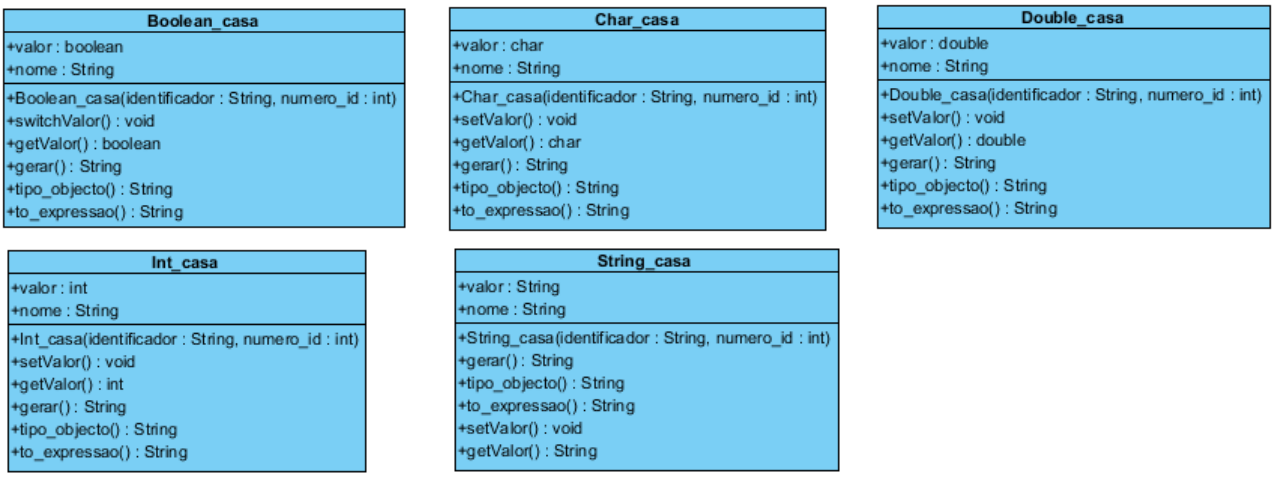
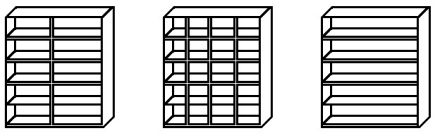


Figura 4.3: Diagrama das classes **Boolean_casa**, **Char_casa**, **Double_casa**, **Int_casa** e **String_casa**

- Variáveis:
  - (boolean / char / Double / int / String) **valor** - variável que irá armazenar o valor do objeto conforme o seu tipo;
  - String **nome** - a String com o identificador da variável objeto;
- Métodos:
  - **Boolean_casa()** / **Char_casa()** / **Double_casa()** / **Int_casa()** / **String_casa()** - inicializam as variáveis dos objetos conforme o tipo de dados que representam. É necessário fornecer a String do identificador e o int do número **id** do objeto;
  - **setValor()**/**switchValor()** - altera o valor guardado no objeto para o valor fornecido ao método. No caso da classe **Boolean_casa**, não é necessário fornecer um valor ao método **switchValor()**, visto que o seu uso faz a troca do valor booleano existente, para o seu oposto;
  - **getValor()** - Devolve o valor guardado na variável **valor** do objeto;

- **gerar()** - Devolve a String do código Python no seguinte formato:
  nome = **valor**
- **tipo_objecto()** - Devolve a String com o nome do tipo do objeto, neste caso **"boolean"** / **"char"** / **"double"** / **"int"** / **"string"**;
- **to_expressao** - Devolve a String com o nome do objeto para que este possa ser referenciado dentro de uma expressão.

4.1.3 Array_double, Array_int e Array_string



Os objetos dos arrays da linguagem têm variáveis e métodos semelhantes entre si, tendo apenas o tipo de dados utilizado diferente (ver figura 4.4).

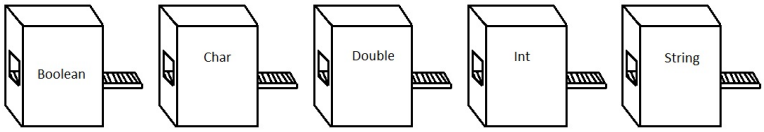
Array_double	Array_int	Array_string
+nome : String +lista : ArrayList<Double> +tamanho : int	+nome : String +lista : ArrayList<Integer> +tamanho : int	+nome : String +lista : ArrayList<String> +tamanho : int
+Array_double(identificador : String, numero_id : int) +Array_double(indentificador : String, numero_id : int, size +addElem() : void +addElem(d : double) : void +getElem(pos : int) : double +removeElem(pos : int) : void +setElem(d : double, pos : int) : void +printLista() : void +gerar() : String +tipo_objecto() : String +to_expressao() : String	+Array_int(identificador : String, numero_id : int) +Array_int(indentificador : String, numero_id : int, size : int) +addElem() : void +addElem(i : int) : void +getElem(pos : int) : int +removeElem(pos : int) : void +setElem(i : int, pos : int) : void +printLista() : void +gerar() : String +tipo_objecto() : String +to_expressao() : String	+Array_string(identificador : String, numero_id : int) +Array_string(indentificador : String, numero_id : int, size : int) +addElem() : void +addElem(s : String) : void +getElem(pos : int) : String +removeElem(pos : int) : void +setElem(s : String, pos : int) : void +printLista() : void +gerar() : String +tipo_objecto() : String +to_expressao() : String

Figura 4.4: Diagrama das classes **Array_double**, **Array_int** e **Array_string**

- Variáveis:
  - String **nome** - a String com o nome do array;
  - ArrayList<Double> / ArrayList<Integer> / ArrayList<String> **lista** - a estrutura de dados onde irá guardar todos os valores armazenados dentro do array;
  - int **tamanho** - o número do tamanho do array;
- Métodos:
  - **Array_double()** / **Array_int()** / **Array_string()** - Construtores dos objetos que inicializam as variáveis já descritas. É necessário fornecer o int do **id** do objeto, a String do identificador e opcionalmente o tamanho do array.
  - **addElem()** - adiciona um novo elemento ao array;
  - **getElem()** - devolve o elemento guardado no array dando o número da sua posição no mesmo;
  - **removeElem()** - remove o elemento no número da posição do array dado;

- **setElem()** - altera o valor guardado no array, sendo necessário fornecer o valor e a posição a alterar no array;
- **printLista()** - faz print de todos os valores guardados no array na consola (para debugging);
- **gerar()** - gera o código Python da declaração array conforme os valores existentes na *lista* e devolve a String deste código. O código gerado segue o seguinte formato: **nome**=[elemento1,elemento2,...]
- **tipo_objecto()** - devolve a string com o nome do tipo do objeto, neste caso **"array_double"** / **"array_int"** / **"array_string"**;
- **to_expressao()** - devolve a String do nome do array.

4.1.4 Maquina_boolean, Maquina_char, Maquina_double, Maquina_int e Maquina_string



As maquinas representam atribuição que são feitos ao longo da execução de um programa. O resumo das variáveis e métodos presentes nestas classes está representado no diagrama da figura 4.5.

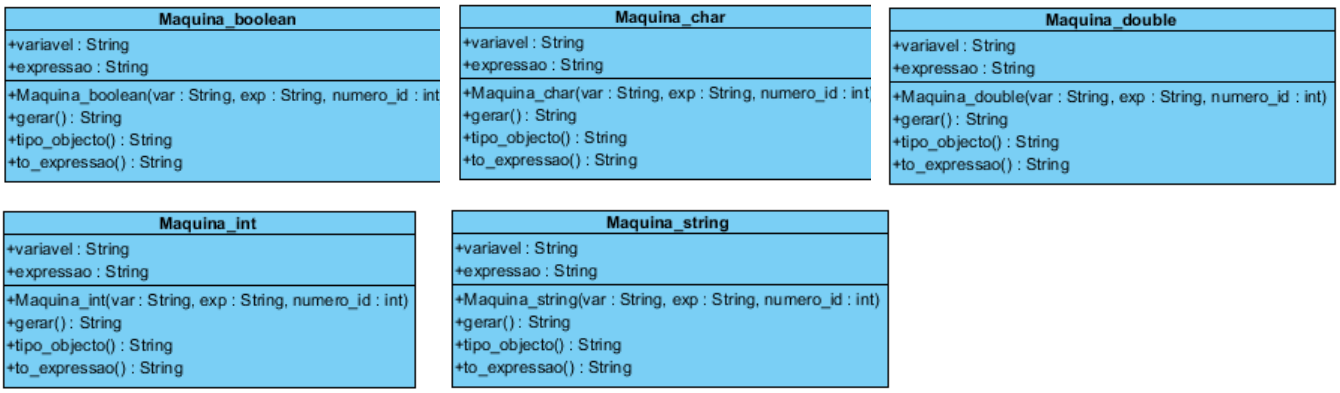


Figura 4.5: Diagrama das classes **Maquina_boolean**, **Maquina_char**, **Maquina_double**, **Maquina_int** e **Maquina_string**

- Variáveis:
  - String **variavel** - nome da variável a guardar o resultado da expressão;
  - String **expressao** - expressão a utilizar na atribuição da variável;
- Métodos:
  - **Maquina_boolean()** / **Maquina_char()** / **Maquina_double()** / **Maquina_int()** / **Maquina_string()** - os construtores que inicializam as variáveis já descritas;

- **gerar()** - gera a String do código Python associado à maquina. O formato do código gerado é o seguinte:
  variavel = **expressao**
- **tipo_objecto()** - devolve a String com o nome do tipo do objeto, neste caso devolve **"maquina_boolean"** / **"maquina_char"** / **"maquina_double"** / **"maquina_int"** / **"maquina_string"**;
- **to_expressao()** - devolve **null** visto não ser um objeto acessível durante a construção de expressões.

4.1.5 Blank

Este objeto representa um objeto vazio. Do ponto de vista de uma linguagem de programação textual, representa os espaços vazios, parágrafos e qualquer outro elemento de formatação do texto de um programa. Este objeto, é necessário para que o jogador não seja forçado a preencher totalmente todas as posições disponíveis/válidas da casa com objetos, para permitir a leitura correta de uma casa. O diagrama da classe **Blank** está representado na figura 4.6.

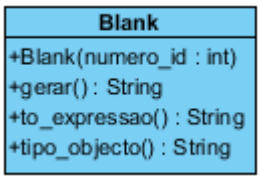
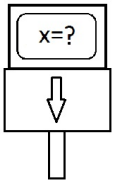


Figura 4.6: Diagrama da classe **Blank**

- Variáveis: só contém as variáveis herdadas da classe **ObjectoCasa**.
- Métodos:
  - **Blank()** - o construtor do objeto, que recebe o int do número **id**. No protótipo do jogo, todos os objetos Blank têm sempre o valor **id** igual a zero;
  - **gerar()** - devolve uma String vazia;
  - **to_expressao()** - devolve **null** visto não ser um objeto acessível durante a construção de expressões;
  - **tipo_objeto()** - devolve a String com o nome do tipo do objeto, neste caso devolve **"blank"**.

4.1.6 Begin_for_ciclo



Este é o único objeto do pacote **objectos_casa** que guarda outro objeto dentro de si (o iterador do ciclo). Encontra-se na figura 4.7 o resumo do conteúdo desta classe.

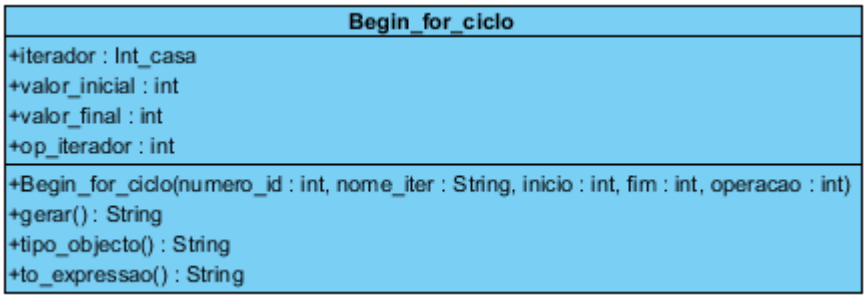
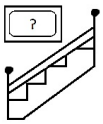


Figura 4.7: Diagrama da classe **Begin_for_ciclo**

- Variáveis:
  - Int_casa **iterador** - o objeto da variável iteradora, apenas acessível dentro do ciclo. O iterador do ciclo "for"é um objeto **Int_casa** de forma a que este possa ser referenciado dentro do ciclo por outros objetos da casa;
  - int **valor_inicial** - o valor inicial que a variável contém no inicio do ciclo;
  - int **valor_final** - o valor final que a variável deverá ter no final do ciclo;
  - int **op_iterador** - o número int, positivo ou negativo, a somar ou subtrair ao valor do **iterador** em cada passo do ciclo;
- Métodos:
  - **Begin_for_ciclo()** - o construtor do objeto, sendo necessário fornecer o nome do iterador, três números int do valor inicial, do valor final e da operação a realizar neste e finalmente o int do **id** do objeto;
  - **gerar()** - devolve a String do código Python com o seguinte formato:
    for **interador.nome** in range(**valor_inicial**, **valor_final**, **op_iterador**):
  - **tipo_objeto()** - devolve a String que identifica o tipo de objeto, neste caso será **"begin_for_ciclo"**;
  - **to_expressao()** - devolve **null**, visto que este objeto não é acessível na construção de expressões.

4.1.7 Begin_if



O diagrama da classe **Begin_if** está representada na figura 4.8.

- Variáveis:

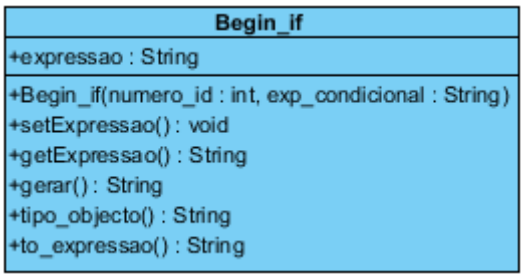
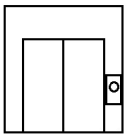


Figura 4.8: Diagrama da classe **Begin_if**

– String **expressao** - a String com a expressão condicional do "if";

- Métodos:
  - **Begin_if()** - construtor do objeto, sendo necessário o número int do **id** do objeto e a String da expressão;
  - **setExpressao()** - define uma nova expressão dada uma String;
  - **getExpressao()** - devolve a String da expressão condicional;
  - **gerar()** - gera e devolve a String do código Python, consoante a seguinte estrutura:
    if **expressao**:
  - **tipo_objeto()** - devolve a String "begin_if"para identificar o objeto dentro da casa;
  - **to_expressao()** - devolve **null** visto não ser acessível no construtor de expressões.

4.1.8 Begin_switch



Este objeto, do ponto de vista gráfico, representa o inicio de um bloco "switch", mas no contexto de uma linguagem textual representa o "default"ou "else"do conjunto de condições possíveis num bloco "switch". O diagrama desta classe está representada na figura 4.9.

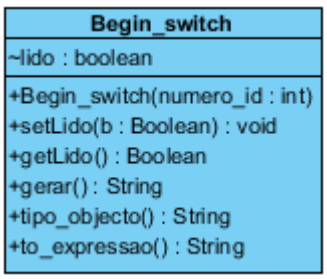


Figura 4.9: Diagrama da classe **Begin_switch**

- Variáveis:

– boolean lido - variável utilizada durante a leitura da casa, na qual indica se este objeto já foi lido conforme o seu valor booleano. Este boolean é necessário para se fazer a leitura correta da casa, assim como efetuar a geração do código Python na ordem correta(ver LeitorCasa para mais detalhes);

- Métodos:

– Begin_switch() - o construtor do objeto, ao qual é necessário fornecer o número **id** a atribuir ao objeto;

– **setLido()** - atribui ao valor da variável **lido** o boolean fornecido a este método;

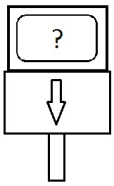
– **getLido()** - devolve o boolean guardado em **lido**;

– **gerar()** - gera e devolve a String do código Python deste objeto, nomeadamente:
else:

– **tipo_objeto()** - devolve a String **"begin_switch"**;

– **to_expressao()** - devolve **null**, uma vez que não é suposto este objeto ser usado dentro de uma expressão.

4.1.9 Begin_while_ciclo



O diagrama desta classe está representada na figura 4.10.

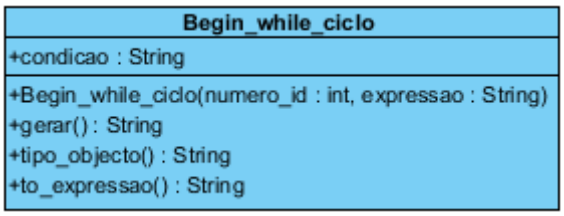


Figura 4.10: Diagrama da classe **Begin_while_ciclo**

- Variáveis:

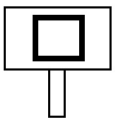
– String **codicao** - a String com a expressão da condição que controla a repetição do ciclo;

- Métodos:

– **Begin_while_ciclo()** - construtor do objeto, recebe o int **id** do objeto e a String da condição do ciclo;

- **gerar()** - devolve a String do código Python com o seguinte formato:
  while **condicao**:
- **tipo_objeto()** - devolve a String **"begin_while_ciclo"**;
- **to_expressao()** - devolve **null**, visto tratar-se de um objeto não disponível dentro de uma expressão.

4.1.10 Break_casa



O diagrama desta classe está representada na figura 4.11.

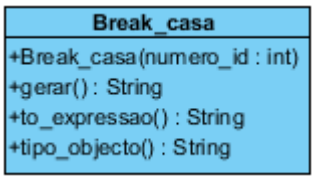
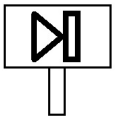


Figura 4.11: Diagrama da classe **Break_casa**

- Variáveis: só contém as variáveis herdadas da classe **ObjectoCasa**.
- Métodos:
  - **Break_casa()** - o construtor do objeto, só necessita do número int **id**;
  - **gerar()** - gera a String do código Python que corresponde a String:
    break
  - **to_expressao()** - devolve **null**, visto não ser um objeto disponível dentro de expressões;
  - **tipo_objeto()** - devolve a String **"break"**.

4.1.11 Continue_casa



O diagrama desta classe está representada na figura 4.12.

- Variáveis: só contém as variáveis herdadas da classe **ObjectoCasa**.
- Métodos:

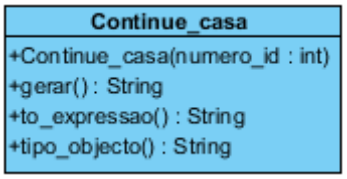
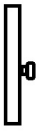


Figura 4.12: Diagrama da classe **Continue_casa**

- **Continue_casa()** - o construtor do objeto, só necessita do número int **id**;
- **gerar()** - gera a String do código Python que corresponde à String:
  continue
- **to_expressao()** - devolve **null**, visto não estar acessível dentro de expressões;
- **tipo_objeto()** - devolve a String **"continue"**.

4.1.12 Def_func



Este objeto só pode estar presente na primeira posição da casa, representa a definição da casa como função. O diagrama desta classe está representada na figura 4.13.

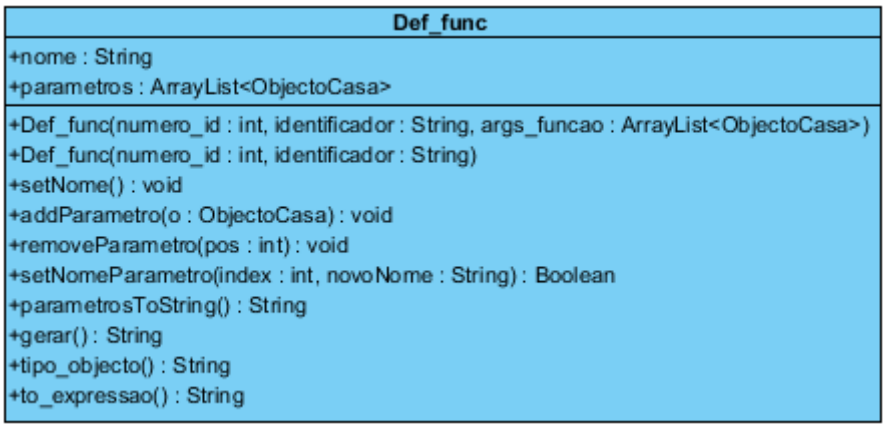
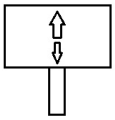


Figura 4.13: Diagrama da classe **Def_func**

- Variáveis:
  - String **nome** - a String com o nome da função, nome esse que tem de ser igual ao da casa;
  - ArrayList <ObjetoCasa> **parametros** - contém a lista dos parâmetros da função. A lista contém objetos do tipo ObjetoCasa para que possam ser referenciados dentro da casa;
- Métodos:

- **Def_func()** - o construtor do objeto, necessita do número **id**, a String do nome da função e opcionalmente uma lista **ObjetoCasa** com os parâmetros da função;
- **setNome()** - dada uma String, altera o nome da função para esse valor;
- **addParametro()** - dado um **ObjetoCasa**, adiciona esse à lista de parâmetros da função;
- **removeParametro()** - remove um parâmetro da lista, dado o número int da posição do objeto a remover;
- **setNomeParametro()** - altera o nome do parâmetro guardado na lista, dado o número da posição e a String com o novo nome;
- **parametrosToString()** - devolve a String com os parâmetros guardados na lista **parametros**;
- **gerar()** - devolve a String com o código Python, a estrutura deste código é a seguinte:
  def **nome** (parâmetro1, parâmetro2, ...):
- **tipo_objeto()** - devolve a String **"def_func"**;
- **to_expressao()** - devolve **null**, o objeto **Def_func** é apenas utilizado para definir a função da casa.

4.1.13 End_ciclo



O diagrama da classe do objeto que marca o fim de um ciclo (for ou while) está representado na figura 4.14.

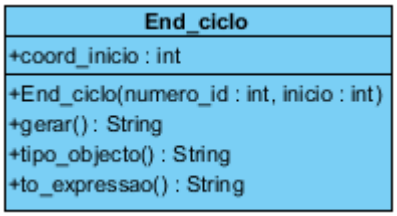
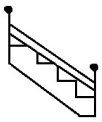


Figura 4.14: Diagrama da classe **End_ciclo**

- Variáveis:
  - int **coord_inicio** - o número int da coordenada x onde marca o inicio do ciclo. Como o par de objetos que forma um ciclo têm de estar sempre no mesmo andar da casa não é necessário registar o valor da coordenada y;
- Métodos:
  - **End_ciclo()** - construtor do objeto, que recebe o int da coordenada x do inicio do ciclo e o número **id** do objeto;
  - **gerar()** - devolve uma String vazia;
  - **tipo_objeto()** - devolve a String **"end_ciclo"**;
  - **to_expressao()** - devolve o **null**, uma vez que não é possível utilizar ciclos dentro de expressões.

4.1.14 End_if



O diagrama desta classe está representada na figura 4.15.

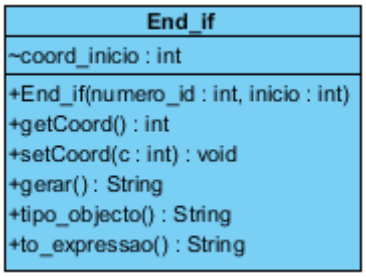
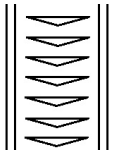


Figura 4.15: Diagrama da classe **End_if**

- Variáveis:
  - int **coord_inicio** - o número int da coordenada x do objeto que delimita o inicio do "if". Esta variável é utilizada durante a leitura da casa, para certificar que o conteúdo debaixo das escadas "if"(dentro do "else") é também lido;
- Métodos:
  - **End_if()** - construtor do objeto, recebe o int **id** e o número int da posição x do inicio do "if";
  - **getCoord()** - devolve a coordenada x do inicio do bloco "if";
  - **setCoord()** - atribui à variável **coord_inicio** o valor dado o número int;
  - **gerar()** - retorna a String do código Python:
    else:
  - **tipo_objeto()** - retorna a String com o texto "end_if";
  - **to_expressao()** - retorna **null**.

4.1.15 End_switch



Este objeto delimita o final de um andar/secção do "switch", assim como, também é utilizado para determinar a largura do bloco "switch". O diagrama desta classe está representada na figura 4.16.

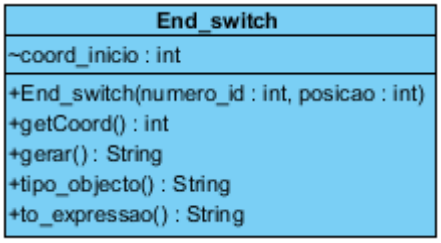
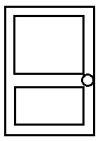


Figura 4.16: Diagrama da classe **End_switch**

- Variáveis:
  - int **coord_inicio** - número da coordenada x do objeto **Switch_case** ou **Begin_switch** associado a este. Esta variável é necessária durante a leitura dos andares de um "Switch"aos ziguezagues, desde o último andar até ao primeiro. No caso do **Begin_switch**, a coordenada deve ser sempre igual a 0, pois não podem existir andares abaixo do rés-de-chão da casa;
- Métodos:
  - **End_switch()** - construtor do objeto, recebe o int **id** do objeto e o int da coordenada x do inicio do bloco "Switch";
  - **getCoord()** - devolve o int da coordenada x do inicio do "Switch";
  - **gerar()** - devolve a String vazia;
  - **tipo_objeto()** - devolve a String: **"end_switch"**;
  - **to_expressao()** - devolve **null**.

4.1.16 Function_call



Este objeto representa a chamada de uma função da casa. Para utilizar uma função de uma casa existente, é necessário utilizar um objeto deste tipo conforme as características definidas na função da casa que se pretende referenciar. O diagrama desta classe está representada na figura 4.17.

- Variáveis:
  - String **nome** - o nome da função a chamar;
  - String **tipo_objeto_retorno** - o tipo de valor que esta função devolve. Esta variável é necessária para fazer a filtragem dos tipos de variáveis e funções disponíveis na construção de uma expressão;
  - ArrayList <String> **argumentos** - lista com as Strings das variáveis de passagem a utilizar na função;

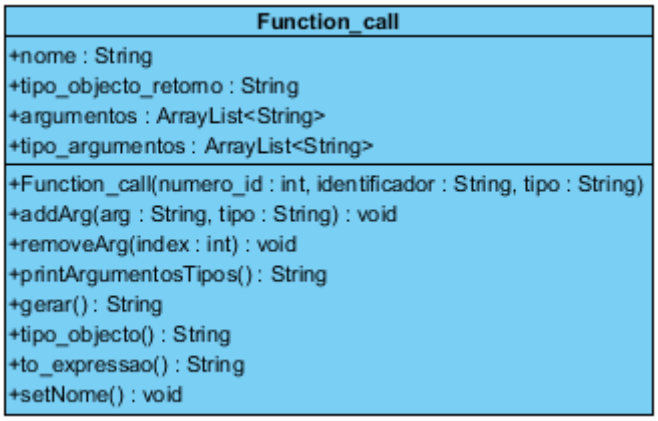


Figura 4.17: Diagrama da classe **Function_call**

– ArrayList <String> **tipo_argumentos** - o tipo necessário em cada argumento da função;

- Métodos:

– **Function_call()** - construtor do objeto, requer o int **id** do objeto, a String com o nome da função e a String do tipo do valor de retorno desta;
– **setNome()** - dada uma String, altera o nome do objeto para esse valor;
– **addArg()** - adiciona um argumento à função, fornecendo as Strings do nome e tipo do argumento dados;
– **removeArg()** - remove um argumento, dada a posição do número int;
– **printArgumentosTipos()** - devolve a String com a lista dos argumentos e tipos guardados no objeto;
– **gerar()** - gera e devolve a String do código Python com o seguinte formato:
nome(argumento 1, argumento 2,...)
– **tipo_objeto()** - devolve a String **"function_call"**;
– **to_expressao()** - devolve a String do nome da função.

4.1.17 Return_casa

O diagrama desta classe está representada na figura 4.18.

- Variáveis:

– String **expressao_valor_retorno** - a String da expressão do valor de retorno da casa;

- Métodos:

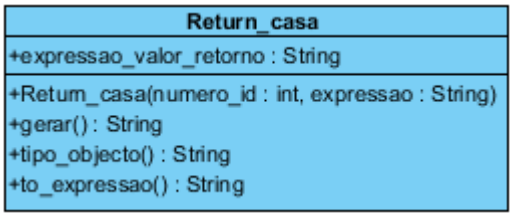
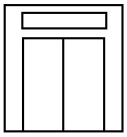


Figura 4.18: Diagrama da classe **Return_casa**

- **Return_casa()** - construtor do objeto que necessita do int **id** e da String da expressão;
- **gerar()** - devolve a String do código Python com o formato:
  return *expressao*
- **tipo_objeto()** - devolve a String **"return"**;
- **to_expressao()** - devolve **null**.

4.1.18 Switch_case



Este objeto representa um dos casos possíveis, dentro de um "Switch". O diagrama desta classe está representada na figura 4.19.

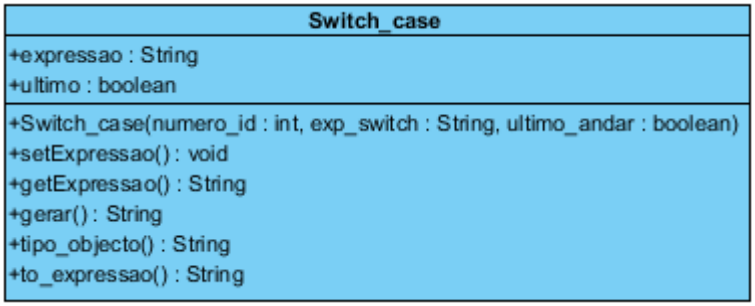


Figura 4.19: Diagrama da classe **Switch_case**

- Variáveis:
  - String **expressao** - a String da expressão a aplicar neste caso da "Switch";
  - boolean ultimo - o boolean a indicar se é o último andar do "Switch". Esta variável é utilizada na função **gerar()**, para certificar a geração correta do código Python;
- Métodos:
  - **Switch_case()** - construtor do objeto, onde é necessário fornecer o int **id**, a String da expressão e o boolean que indica se é o último andar do bloco "Switch";

- **setExpressao()** - altera a expressão condicional para a String dada, a este método;
- **getExpressao()** - devolve a String da expressão do caso "Switch";
- **gerar()** - gera e retorna a String do código Python, dependendo do valor booleano em **ultimo**, com as seguintes estruturas:
  Se **ultimo**=true gera a String:
  if **expressao**:
  Caso contrário gera:
  elif **expressao**:
- **tipo_objeto()** - retorna a String **"switch_case"**;
- **to_expressao()** - devolve **null**.

4.2 ObjetosPermitidosCoordCasa

Esta classe representa as regras dos objetos permitidos numa posição da casa. Juntamente com a estrutura de dados responsável pelo armazenamento dos objetos na casa, haverá ainda outra composta por estes objetos, em simultâneo. À medida que o programador adiciona e remove objetos na casa, as regras correspondentes a cada posição serão também modificadas. Durante a construção do sistema editor de casas determinou-se as seguintes restrições:

- **Regra nº1** - Não são permitidos objetos a flutuar na casa. Um objeto tem de ser sempre colocado numa superfície. Isto faz com que, na maioria das vezes, as posições válidas para novos objetos sejam no rés-de-chão da casa. Apenas os objetos condicionais **if_begin** e **switch_case** é que podem criar novos andares na casa;
- **Regra nº2** - Não é permitido colocar objetos condicionais dentro de um bloco condicional já existente, isto significa que não é possível colocar objetos **if_begin**, **end_if**, **begin_switch**, **switch_case** e **end_switch** dentro de um bloco "if"ou "Switch"já existente. A razão desta restrição deve-se aos possíveis conflitos do espaço horizontal entre os objetos (ver figura 4.20);
- **Regra nº3** - Não é permitido colocar os objetos continue e break fora de um ciclo, visto tratarem-se de instruções que manipulam a execução de um ciclo;
- **Regra nº4** - O objeto **def_func** só pode ser colocado nas coordenadas (x=0,y=0) da casa;
- **Regra nº5** - Objetos **switch_case** só podem ser colocados em cima de um **begin_switch** e/ou de outro **switch_case**, para restringir o uso de **switch_case** exclusivamente em locais onde se inicia um bloco de instruções "Switch";
- **Regra nº6** - Objetos **end_switch** só podem ser colocados em cima de outros **end_switch**, assim como no **switch_case**, esta regra restringe o uso de **switch_end** no final de um bloco "Switch";
- **Regra nº7** - Não é permitida a sobreposição parcial dos blocos condicionais com ciclos e vice-versa, isto que dizer, que não se pode colocar o inicio de um bloco condicional fora de um ciclo, tendo o marcador final dentro do ciclo e vice-versa(ver figura 4.21).

Durante a construção de uma casa, o editor deverá permitir a colocação de objetos que cumpram estas regras, considerando isto, o diagrama correspondente a esta classe está representado no diagrama da figura 4.22.

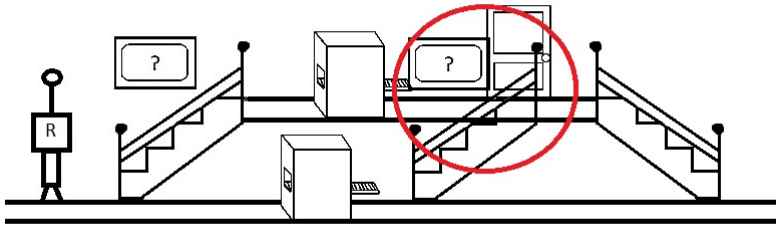


Figura 4.20: Exemplo de conflitos com outros objetos no espaço horizontal

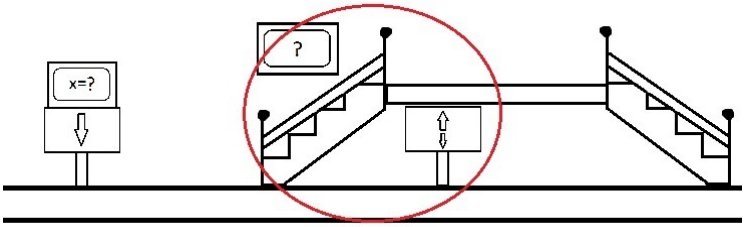


Figura 4.21: Exemplo de sobreposição de blocos condicionais

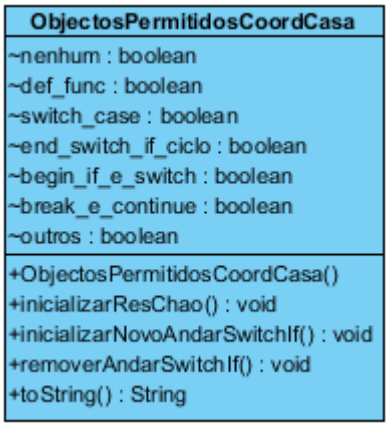


Figura 4.22: Diagrama da classe **ObjetosPermitidosCoordCasa**

- Variáveis:
  - boolean **nenhum** - boolean que indica que nenhum objeto é permitido na respetiva posição da casa. Esta variável é necessária para evitar objetos "flutuantes"na casa, cumprindo assim a regra nº1;
  - boolean **def_func** - boolean que indica se o **def_func** é permitido. Esta variável é utilizada para cumprir a regra nº4;

- boolean **switch_case** - indica se o objeto **switch_case** é permitido, esta variável é utilizada para cumprir a regra nº5;
- boolean **end_switch_if_ciclo** - indica que os objetos terminais dos blocos **switch**, if e ciclos são permitidos e é utilizada para cumprir as regras nº2, nº6 e a nº7;
- boolean **begin_if_e_switch** - indica que os objetos iniciais do "if"e do "Switch"são permitidos, esta variável é utilizada para cumprir as regras nº2 e º7;
- boolean **break_e_continue** - indica se é possível colocar os objetos continue e break, este boolean é utilizado para cumprir a regra nº3;
- boolean **outros** - este boolean restringe a colocação de todos os outros objetos que não estejam especificamente indicados na lista de regras, à exceção da primeira regra. Este boolean indica em que locais é possível colocar variáveis, máquinas(atribuições), chamadas de funções e "returns";

- Métodos:

- **objetosPermitidosCoordCasa()** - construtor do objeto, não recebe nenhum valor para inicializar. Coloca todas as variáveis booleanas com o valor *false* à exceção da variável **nenhum** que contém o valor *true*;
- **inicializarResChao()** - método utilizado para inicializar as regras no rés-do-chão da casa, colocando os valores booleanos de **variavel**, **begin_if_e_switch** e **outros** *true* e colocando **nenhum** com o valor *false*;
- **inicializarNovoAndarSwitchIf()** - cada vez que se gera um novo andar à casa este método é utilizado, alterando as regras para restringir a colocação de objetos condicionais nesse mesmo andar e permitir outros objetos;
- **removerAndarSwitchIf()** - este método é chamado quando é removido um andar da casa, altera os booleans de forma a proibir a colocação de objetos;
- **toString()** - retorna a String com um código de zeros(*false*) e uns(*true*) a indicar os valores dos booleans. O significado de cada digito corresponde à seguinte variável boolean:
  Nº1-**nenhum**; Nº2-**def_func**; Nº3-**switch_case**; Nº4-**end_switch_if_ciclo**; Nº5-**begin_if_e_switch**; Nº6-**break_e_continue**; Nº7-**outros**;

4.3 Casa

A classe Casa representa a estrutura de dados cuja informação de um programa é armazenado e representa também um nível de um puzzle do jogo. Visto que a casa está dividida por uma grelha, escolheu-se utilizar um array bidimensional para armazenar os objetos introduzidos dentro desta. Esta classe é também responsável pela modificação e introdução de objetos dentro da casa, modificando as regras de cada posição e certificando que não são colocados objetos em locais inválidos. O diagrama desta classe está representada na figura 4.23.

- Variáveis:

- String **nome_casa** - a String que guarda o nome da casa, essencial para guardar o ficheiro da casa e para que possa ser chamada por outras casas. O **Def_func** (se presente) deverá sempre ter o mesmo nome que a casa;
- String **tipo_valor_retorno** - a String com o tipo de valor que a casa deverá devolver;

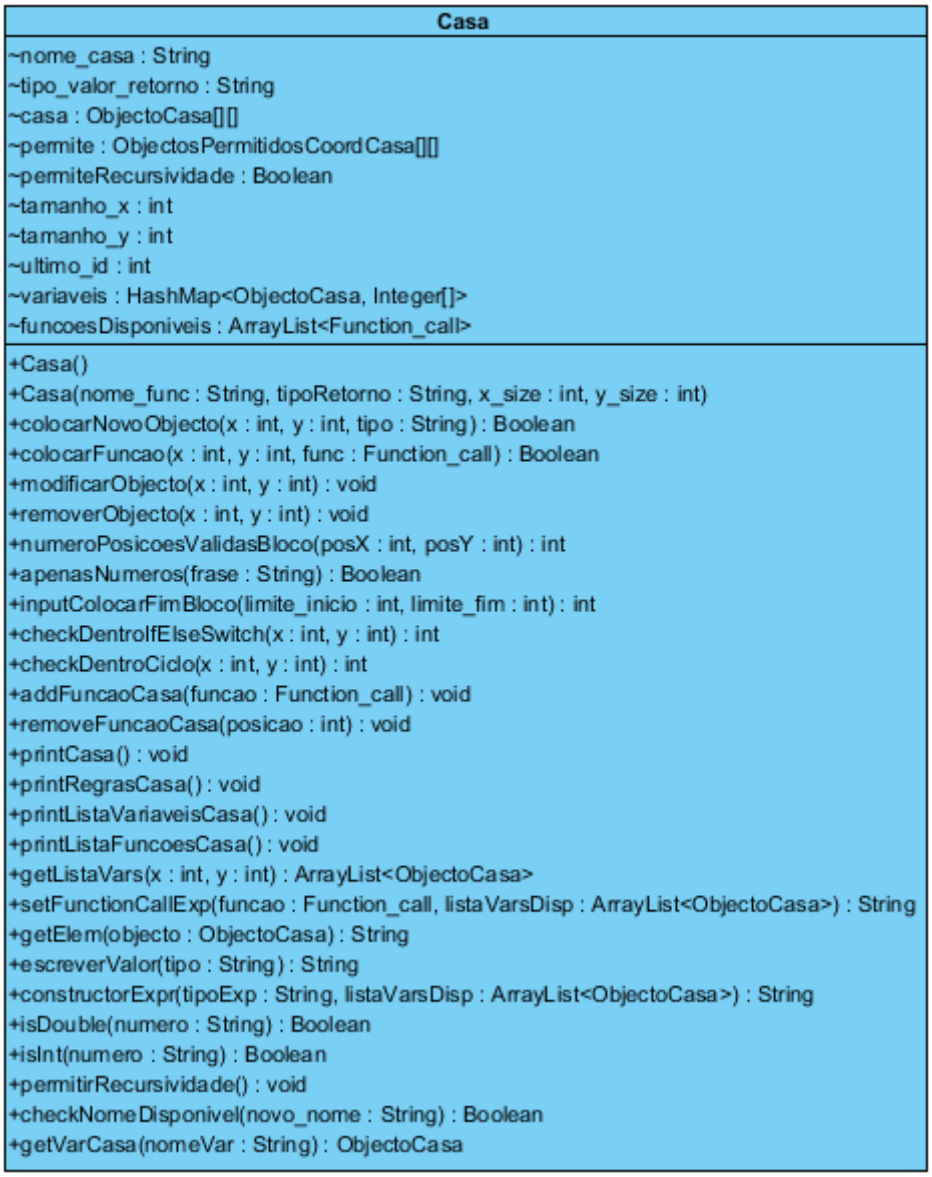


Figura 4.23: Diagrama da classe **Casa**

- ObjetoCasa [][] **casa** - o array bidimensional em que todos os objetos da casa são guardados;
- ObjetosPermitidosCoordCasa[][] **permite** - array bidimensional com as regras de cada coordenada da casa. Este array tem de ter as mesmas dimensões que o array **casa**;
- Boolean **permiteRecursividade** - boolean a indicar se é permitido usar a própria função da casa dentro desta;
- int **tamanho_x** - int com o número do tamanho de espaços no eixo dos x;
- int **tamanho_y** - int com o número do tamanho de espaços no eixo dos y;
- int **ultimo_id** - int com o número do **id** do objeto mais recentemente adicionado;
- HashMap<objetoCasa,Integer[]> variaveis - a tabela de "Hash"que guarda a lista de variáveis disponíveis e declaradas dentro da casa. Cada entrada contém o objeto da variável e um array com quatro valores int que delimitam a área das coordenadas da casa, onde é permitido o uso

destas variáveis ("scope"ou escopo das variáveis). Cada número da posição do array guarda a seguinte informação:

* Posição 0 - a primeira coordenada válida do eixo dos x, do uso da variável, os valores x que se seguem a esta são também válidas até ao limite;
* Posição 1 - última coordenada válida no eixo dos x, os valores x a seguir a esta, já não permitem o uso da variável na casa;
* Posição 2 - primeira coordenada válida do eixo dos y, os valores que se seguem a este são válidos;
* Posição 3 - última coordenada válida no eixo dos y, os valores a seguir a este não são coordenadas válidas no uso da variável.

Isto significa que todos os valores x entre a Posição 0 e a Posição 1, incluindo os mesmos, são valores de x em que se pode utilizar a variável. Aplica-se a mesma lógica aos valores de y guardados na Posição 2 e 3 do array;

– ArrayList<Function_call> **funcoesDisponiveis** - Esta lista guarda todos os objetos das chamadas de funções disponíveis na casa;

▪ Métodos:

– **Casa()** - construtor da classe **Casa**, inicializa todas as variáveis. Pode-se também criar uma casa já com o nome, o tipo de valor de retorno e as dimensões do x e y, dados estes valores ao construtor;

– **colocarNovoObjeto()** - coloca um novo objeto da casa dada a String do tipo do objeto que se pretende colocar e os números int x e y da posição deste. Esta função, identifica o tipo de objeto a colocar, utilizando o seu método **tipo_objeto()** e verifica se este pode ser colocado na posição indicada da casa. Se o objeto cumprir as regras estabelecidas naquela posição e se existir espaço (no caso de objetos de blocos "Switch"/"if"/ciclo) coloca-o nesse local.

A declaração de novas variáveis na casa requer que estas sejam adicionadas à lista de variáveis da mesma, para permitir o uso destas em expressões. Também é necessário saber qual a zona da casa em que estas podem ser utilizadas. Regra geral, uma nova variável pode ser utilizada nas coordenadas x, a seguir ao objeto até ao final da casa e a partir da coordenada y, do mesmo andar do objeto, até ao limite da dimensão y da casa. As únicas exceções são quando o objeto é colocado dentro de um bloco "if"/"Switch"/ciclo, só podendo ser referenciada dentro destas zonas da casa. Para calcular estes limites são utilizadas as funções **checkDentroIfElseSwitch()** e **checkDentroCiclo()** para determinar se a variável colocada na casa se encontra dentro de um destes blocos de instruções.

Os objetos condicionais e de ciclos são forçosamente colocados aos pares para evitar erros de compilação. Ao colocar o objeto do inicio do bloco, o jogo mostra as coordenadas válidas para a colocação do final do bloco, calculadas com a função **numeroPosicoesValidasBloco()** e utilizando esta informação recebe o input onde colocar o objeto terminal do bloco com a função **inputColocarFimBloco()**.

Visto que as chamadas de função têm de estar definidas na lista de funções disponíveis da casa, para serem utilizadas dentro desta, não é possível colocar objetos do tipo **Function_call** utilizando este método, existindo outra função como alternativa;

– **colocarFuncao()** - coloca um objeto *Function_call* já existente na lista de funções da casa, sendo necessário o objeto da função a chamar e as coordenadas a colocar o objeto. Tal como na função **colocarNovoObjeto()** esta também verifica a chamada de função, que pode ser colocada nas coordenadas dadas;

- **modificarObjeto()** - modifica um objeto nas coordenadas dadas x e y da casa. Esta função mostra a informação guardada dentro do objeto e as opções disponíveis para alterar o objeto;
- **removerObjeto()** - remove o objeto guardado nas coordenadas x e y dadas à função, nos casos em que o objeto a remover faça parte da estrutura de um bloco "if", "Switch"ou ciclo, toda a informação contida neste bloco será apagada;
- **numeroPosicoesValidasBloco()** - função que devolve o int com o número de posições válidas para colocar o final de um bloco "Switch", "if"e ciclo, dadas as coordenadas do inicio deste bloco. Este algoritmo conta o número de espaços com objetos vazios e pára quando encontra uma posição ocupada. Esta função é utilizada dentro da função **colocarNovoobjeto()**;
- **apenasNumeros()** - função utilizada no input do utilizador que devolve o boolean a indicar se a string dada contém apenas dígitos de números. Esta função certifica que nos casos em que é pedido um número ao jogador, apenas os números válidos introduzidos são aceites;
- **inputColocarFimBloco()** - função que recebe os limites válidos na colocação do objeto final de um bloco e recebe o input do utilizador para colocar este objeto conforme as restrições calculadas;
- **checkDentroIfElseSwitch()** - dadas as coordenadas x e y, verifica se esta posição se encontra dentro de um bloco if/switch/else e devolve um valor booleano com o resultado deste teste;
- **checkDentroCiclo()** - dadas as coordenadas de um objeto, devolve um valor booleano a indicar que este se encontra dentro de um ciclo;
- **addFuncaoCasa()** - adiciona uma nova função à lista de chamadas de funções disponíveis da casa, sendo necessário fornecer o objeto **Function_call** desta;
- **removeFuncaoCasa()** - remove uma função guardada na lista de funções da casa;
- **printCasa()** - imprime na consola os objetos da casa em formato de texto;
- **printRegrasCasa()** - imprime na consola as regras impostas em cada coordenada da casa, utilizando o código descrito no método **toString** da classe **ObjetosPermitidosCoordCasa**;
- **printListaVariaveisCasa()** - imprime na consola a lista de variáveis disponíveis dentro da casa, juntamente com o array a guardar os limites destes;
- **printListaFuncoesCasa()** - imprime na consola a lista de funções que podem ser utilizadas dentro da casa;
- **getListaVars()** - dada uma posição da casa, devolve uma lista de variáveis que possam ser utilizadas nessa coordenada da casa. Esta função utiliza os valores do array presente em cada objeto da lista de variáveis da casa, para filtrar as variáveis permitidas naquele espaço;
- **setFunctionCallExp()** - esta função é utilizada para devolver a String de uma chamada de função para uso dentro de uma expressão, mostrando os argumentos necessários na função e recebendo o input do jogador até que seja preenchido todos os valores de passagem necessários. Esta função recebe a lista com todas as variáveis disponíveis no local onde a expressão é utilizada e o objeto da **Function_call** a adicionar à expressão;
- **getElem()** - esta função é utilizada para devolver a String a referenciar um elemento dentro de um "array"ou de uma String para que possa ser utilizada dentro de uma expressão;
- **escreverValor()** - dada a String do tipo de dados, esta função permite que o utilizador digite um valor do tipo indicado e devolva a String deste input. Esta função é utilizada para introduzir valores dentro de uma expressão;
- **constructorExpr()** - função responsável pela construção de expressões presentes em alguns dos objetos da casa. O uso desta função requer a String do tipo de expressão a construir e a lista de todas as variáveis disponíveis onde a expressão será criada. A partir do tipo da expressão,

é realizada a seleção de todas as funções e variáveis que possam ser utilizadas dentro desta expressão. A interface mostra esta informação e pede ao utilizador as ações que este pretende efetuar, de forma a construir a expressão pretendida. Terminada a construção desta, a função devolve a sua String;

- **isDouble()** - dada a string de um número, devolve o valor booleano a indicar se esta representa um número double. As regras da expressão regular utilizadas na verificação da String foram retiradas da documentação do Java 6, presente no método **valueOf()** do objeto Java **Double**;
- **isInt()** - dada a string de um número, devolve o valor booleano a indicar se este representa um número int;
- **permitirRecursividade()** - função utilizada para permitir recursividade na casa. Verifica se já existe um objeto def_func definido na casa e utiliza a informação existente para criar e adicionar um novo **Function_call** à lista de variáveis disponíveis na casa;
- **checkNomeDisponivel()** - dada a String de um nome de uma variável ou função devolve um valor booleano a indicar se este nome está presente dentro das listas das variáveis e das funções da casa;
- **getVarCasa()** - dada a String do nome de uma variável, caso exista, devolve o objeto dessa variável.

4.4 Jython

Após feita a pesquisa acerca do problema de executar código Python dentro de um programa Java, encontrou-se duas resoluções:

- O código é compilado e executado fora do programa Java, sendo necessário ter Python instalado no sistema;
- O código é compilado e executado dentro do programa, utilizando uma implementação de Python em Java chamada Jython;

De forma a evitar problemas na execução do protótipo do jogo, independentemente da configuração do sistema, decidiu-se utilizar a biblioteca de Jython mais recente (especificamente a versão 2.7.0). O uso do .jar desta biblioteca na execução de código Python será feita na classe Prototipo, o qual utiliza a classe **PythonInterpreter** que permite executar código Python guardado em "Strings"(ver a secção da classe Prototipo para mais detalhes).

4.5 LeitorCasa

Esta classe é responsável pela leitura de casas, assim como também pela geração do código Python correspondente à casa lida. Visto que Python requer a formatação do texto de forma a delimitar os blocos de código, o algoritmo de leitura requer um símbolo de indentação e um número de símbolos a adicionar em cada linha do texto, de forma a construir o código devidamente indentado. Tal como foi mostrado na imagem do capitulo anterior, a leitura da casa é feita da esquerda para direita e no caso de existirem múltiplos andares, a leitura será feita no sentido descendente.
Considerando toda esta informação, o diagrama desta classe corresponde ao da figura 4.24.

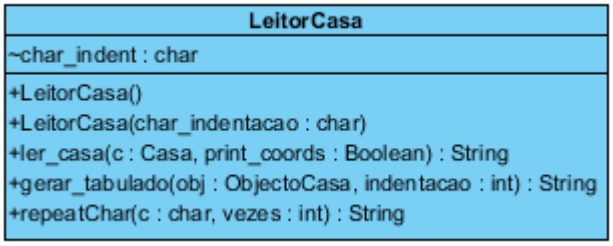


Figura 4.24: Diagrama da classe **LeitorCasa**

- Variáveis:
  - char **char_indent** - o símbolo a utilizar na indentação do código Python;
- Métodos:
  - **LeitorCasa()** - construtor do leitor de casas, que pode receber opcionalmente o símbolo a utilizar na indentação do código gerado, caso contrário utiliza o símbolo "\t"utilizado na tabulação de texto;
  - **repeatChar()** - função que recebe um char e um número int e devolve a String com esse símbolo repetido, conforme o número dado. Esta função é utilizada na função **gerar_tabulado()**;
  - **gerar_tabulado()** - função que recebe um objeto lido na casa e o número a aplicar na indentação. Utiliza o método **gerar()** do objeto lido para obter o seu código Python e devolve a String com o código indentado. Esta função é utilizada pelo método **ler_casa()**;
  - **ler_casa()** - método responsável pela leitura e construção da String do código Python dada uma casa (e consoante o boolean dado) este devolve o código Python com prints das coordenadas da casa para serem usadas na animação da execução do programa. O funcionamento do algoritmo de leitura da casa está descrito na figura 4.27 localizada no final deste capítulo.

O algoritmo descrito deverá na maior parte dos casos, gerar código Python executável. Existem casos onde tal não acontece, tais como:

- A casa contém um andar, criado pelo bloco if, sem objetos;
- A casa contém apenas um bloco "Switch"do rés-de-chão sem nenhuns casos de "Switch"criados;
- A casa contém um andar **switch_case** sem objetos; Nestes três casos específicos, o interpretador Python deverá devolver um erro de compilação, não podendo executar o código da casa.

4.6 Prototipo

Esta classe representa a classe principal do jogo, onde os restantes componentes implementados estão interligados num só sitio, formando assim o jogo. Esta classe contém funções para abrir e guardar casas, assim como para executar e testar uma casa aberta, conforme um conjunto de valores de teste fornecidos ao protótipo. Foi utilizada a biblioteca gráfica "JavaFX"(disponível no "Java Development Kit 8") na construção da interface gráfica. O diagrama desta classe está representada na figura 4.25. A imagem da interface gráfica do protótipo está representada na figura 2.26 no final desta secção.

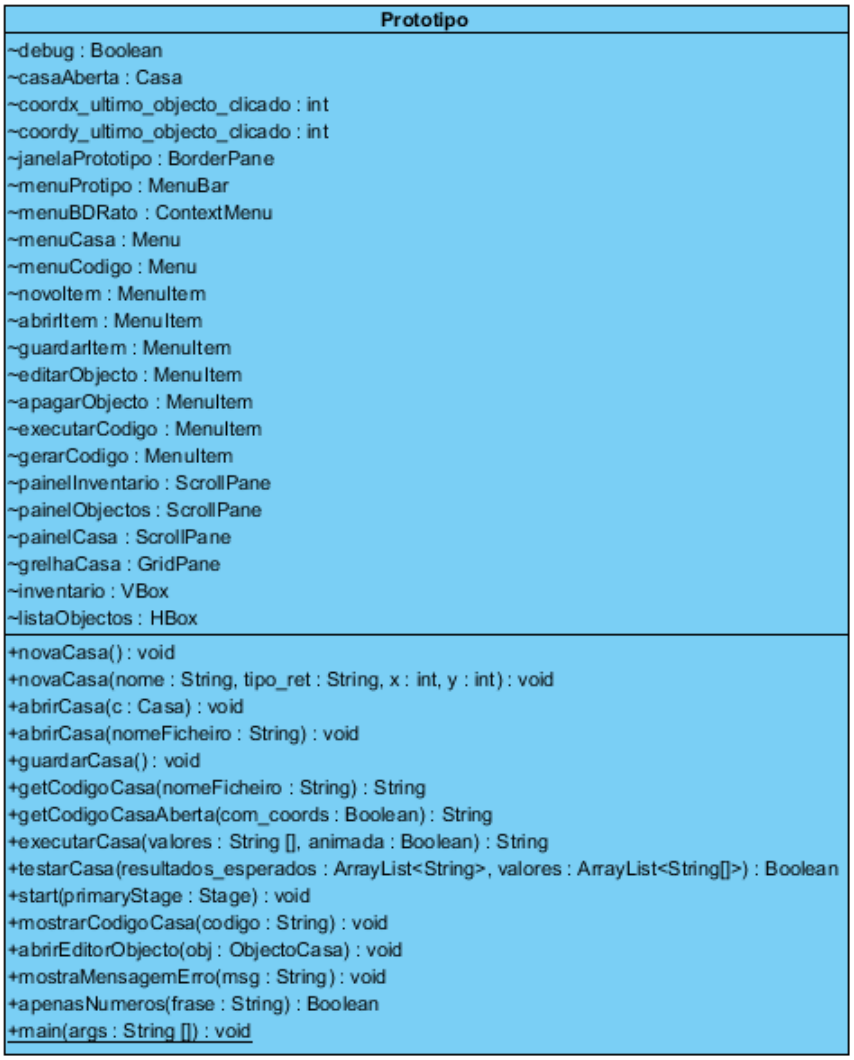


Figura 4.25: Diagrama da classe **Prototipo**

- Variáveis:
  - Boolean **debug** - esta variável controla a inicialização do protótipo no modo de testes;
  - Casa **casaAberta** - esta variável representa a casa aberta no protótipo, podendo ser vista, editada e executada pelo jogador;
  - int **coordx_ultimo_objecto_clicado** - esta variável guarda a coordenada x da grelha casa onde foi clicado com o rato;
  - int **coordy_ultimo_objecto_clicado** - esta variável guarda a coordenada y da grelha casa onde foi clicado com o rato;
  - BorderPane **janelaPrototipo** - objeto da interface gráfica que define a estrutura da janela do protótipo;
  - MenuBar **menuProtipo** - objeto da interface gráfica que define a barra de menus, este elemento está contido dentro da **janelaPrototipo**;
  - ContextMenu **menuBDRato** - objeto da interface gráfica que define a o menu de contexto ao clicar com o botão direito num objeto da casa. Adicionalmente, ao clicar num objeto regista

as coordenadas do objeto clicado e guarda-as nas variáveis **coordx_ultimo_objecto_clicado** e **coordy_ultimo_objecto_clicado**;

- Menu **menuCasa** - objeto da interface gráfica que define o menu "Casa"contido dentro da barra de menu **menuProtipo**;
- Menu **menuCodigo** - objeto da interface gráfica que define o menu "Código"contido dentro da barra de menu **menuProtipo**;
- MenuItem **novoItem** - objeto da interface gráfica que define o item de menu "Novo"contido dentro do menu "Casa";
- MenuItem **abrirItem** - objeto da interface gráfica que define o item de menu "Abrir"contido dentro do menu Casa;
- MenuItem **guardarItem** - objeto da interface gráfica que define o item de menu "Guardar"contido dentro do menu "Casa";
- MenuItem **editarObjecto** - objeto da interface gráfica que define o item de menu "Editar"contido dentro do menu de contexto **menuBDRato**. Ao clicar neste item abre a janela editora do objeto que se pretende editar, fazendo-se uso das variáveis **coordx_ultimo_objecto_clicado** e **coordy_ultimo_objecto_clicado** para identificar o objeto a alterar;
- MenuItem **apagarObjecto** - objeto da interface gráfica que define o item de menu "Apagar"contido dentro do menu de contexto **menuBDRato**;
- MenuItem **executarCodigo** - objeto da interface gráfica que define o item de menu "Executar"contido dentro do menu "Código";
- MenuItem **gerarCodigo** - objeto da interface gráfica que define o item de menu "Gerar"contido dentro do menu "Código";
- ScrollPane **painelInventario** - objeto da interface gráfica que define o espaço da lista de objetos guardados no inventario do robô, este elemento está contido dentro da **janelaPrototipo**;
- ScrollPane **painelObjectos** - objeto da interface gráfica que define o espaço da lista de objetos colocáveis na casa, este elemento está contido dentro da **janelaPrototipo**;;
- ScrollPane **painelCasa** - objeto da interface gráfica que define o espaço da grelha da casa, este elemento está contido dentro da **janelaPrototipo**;
- GridPane **grelhaCasa** - objeto da interface gráfica onde mostra uma grelha com os objetos guardados na casa aberta, este elemento está contido dentro do **painelCasa**;
- VBox **inventario** - objeto da interface gráfica que define a lista gráfica dos objetos guardados no inventário do robô, este elemento está contido dentro do **painelInventario**;
- HBox **listaObjectos** - objeto da interface gráfica que define a lista gráfica dos objetos colocáveis na casa, este elemento está contido dentro **painelCasa**;

▪ Métodos:

- **novaCasa()** - cria uma nova casa vazia;
- **abrirCasa()** - abre uma casa no protótipo, dado o nome do ficheiro ou um objeto **Casa** já existente;
- **guardarCasa()** - guarda a casa atualmente aberta no protótipo num ficheiro com o nome da casa com a extensão de ficheiro ".casa";
- **getCodigoCasa()** - devolve a String do código da casa guardada dada a String do nome do ficheiro. Esta função é utilizada para exportar o código de outras casas, permitindo a execução de chamadas de funções;

- **getCodigoCasaAberta()** - devolve a String do código da casa aberta no protótipo, com ou sem prints das coordenadas conforme o boolean dado;
- **executarCasa()** - executa a casa aberta, dado um array de Strings (contendo os valores dos argumentos a utilizar nesta) e um boolean, devolve uma String com o resultado da execução da casa e conforme o boolean dado, as coordenadas percorridas durante a sua execução;
- **testarCasa()** - dadas duas listas com os resultados esperados e os valores de teste, executa o código da casa e compara os resultados obtidos utilizando a primeira lista. Se nenhum dos testes falhar, devolve o valor booleano *true*, caso contrário, devolve *false*;
- **start()** - função que define a inicialização dos objetos da interface gráfica;
- **mostrarCodigoCasa()** - função que mostra a janela com o código gerado a partir da casa aberta;
- abrirEditorObjecto() - função que mostra a janela editora de um objeto casa;
- **mostraMensagemErro()** - função que mostra uma janela com uma mensagem de erro, utilizado quando ocorre algum erro no protótipo;
- **apenasNumeros()** - função que devolve o boolean a indicar se a string dada contém apenas dígitos de números. Esta função é utilizada na verificação do input do utilizado na interface textual do protótipo.
- **main()** - a função principal da classe e a primeira ser executada no protótipo. Esta função utiliza como opção adicional dois argumentos utilizados para testar o protótipo:
  * **debug_testes** - utilizado para testar diretamente as funcionalidades do protótipo;
  * **interface_texto** - utilizado para executar o protótipo com a interface textual;

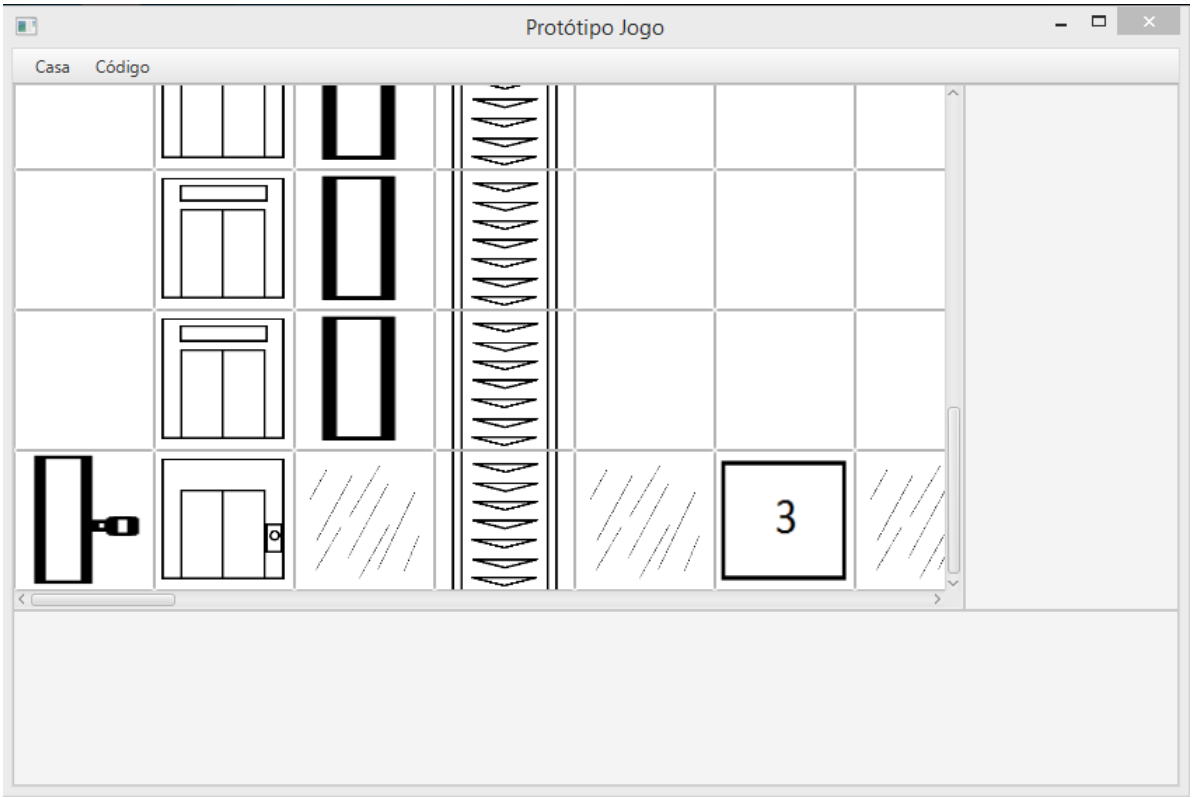


Figura 4.26: Imagem da interface gráfica da classe **Prototipo**

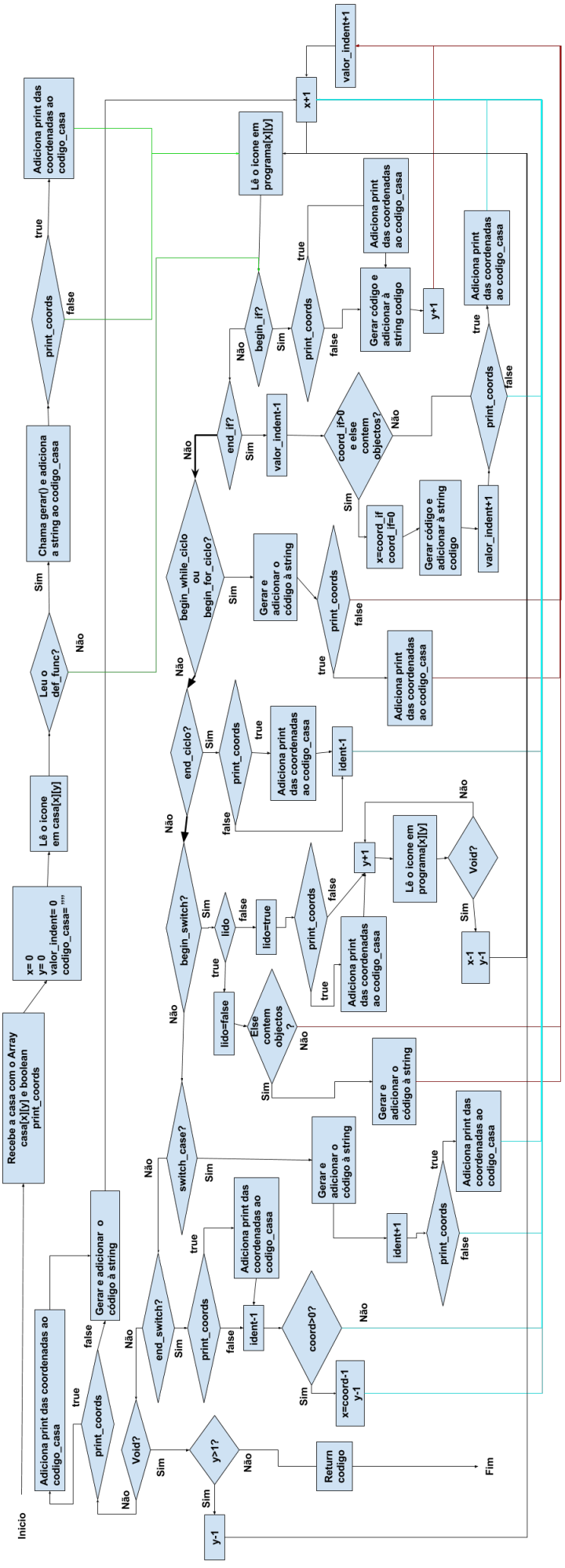


Figura 4.27: Algoritmo de leitura de casas

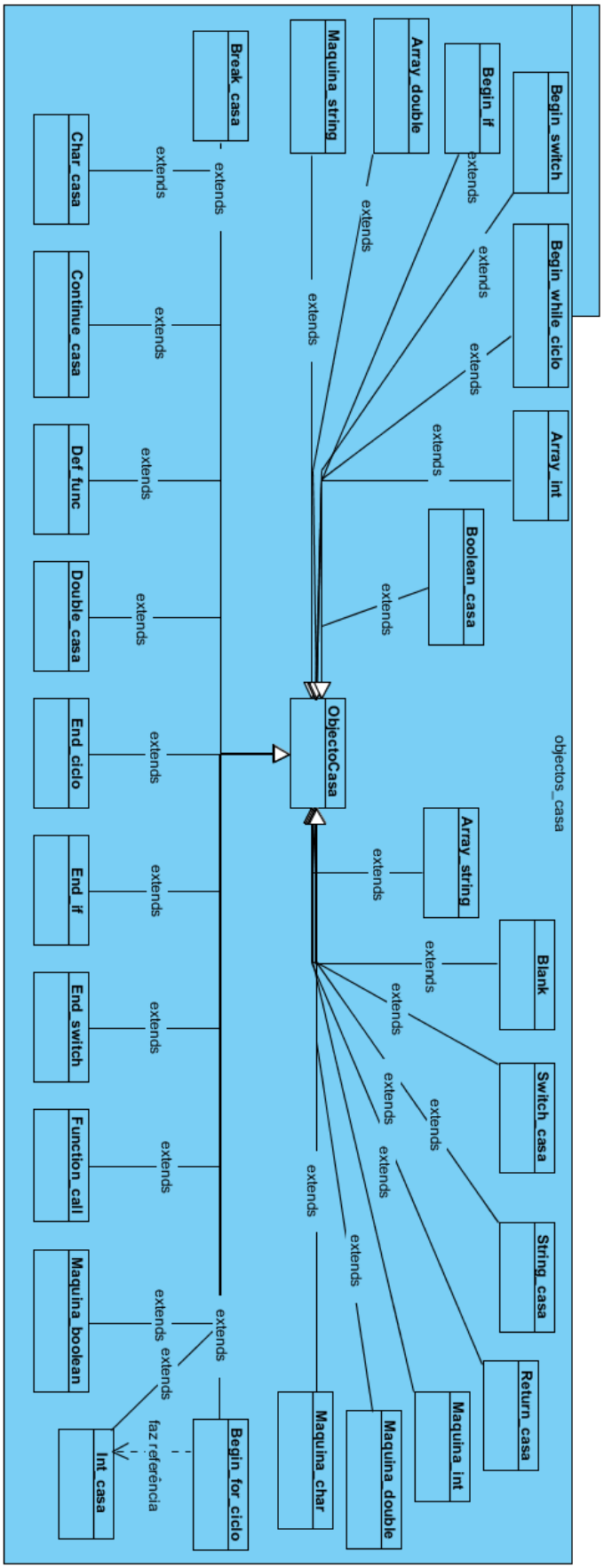


Figura 4.28: Diagrama das classes contidas dentro do pacote **objectos_casa**

5

Avaliação do jogo

Neste capitulo serão analisadas as limitações e os problemas encontrados durante a construção e implementação do protótipo, como também será feita a comparação com os jogos estudados e analisados no capitulo 2.

5.1 Problemas e limitações da linguagem do jogo

Durante a implementação da linguagem do jogo observou-se diversos problemas associados ao design da linguagem:

- Não é possível colocar *"ifs/switches"* dentro de *"ifs"* ou *"switches"*;
- Não é possível criar um novo bloco *"switch"/"if"*/ciclo contendo objetos já existentes, sendo necessário primeiro colocar o bloco *"switch"/"if"*/ciclo e só depois adicionar objetos dentro deste;
- Remover um objeto pertencente à estrutura de um bloco *"switch"/"if"*/ciclo apaga também todos os objetos dentro do mesmo;

- Não é feito tratamento de erros de compilação (incluindo a verificação de expressões e *"ifs"/"switches"*/ciclos vazios);
- Não é possível implementar objetos que façam "*print*", devido à forma como o sistema devolve as coordenadas da execução da casa;
- Podem ser adicionadas novas linguagens de programação, no entanto, isto requer implementar novas classes e funções para permitir a leitura de casas, geração de código, construção de expressões, e por fim, a execução do código.

5.2 Comparação do protótipo com outros jogos

Uma vez que o protótipo não contem puzzles e que as componentes características de um jogo não estão terminadas, não é possível fazer uma avaliação exata do que este consegue ensinar relativamente aos jogos estudados no capítulo 2. Considerando isto, apenas se irá comparar cada jogo com o protótipo relativamente às potencialidades de programação e funcionalidades visuais que cada um apresenta.

5.2.1 Toontalk

Tal como foi dito no capítulo 2, a linguagem visual Toontalk permite o uso de ciclos, instruções condicionais, *"assigns"* e chamadas de funções. Em relação às variáveis, o jogador só tem acesso a *"arrays"* que consegue guardar qualquer tipo de informação, tais como números, texto e outros objetos. Isto faz com que a linguagem seja fracamente tipada em *"arrays"*, não sendo possível impor restrições no uso e identificação de *"arrays"* com objetos de um determinado tipo.
Os objetos utilizados num programa não são identificados por um nome. Para referenciar de forma correta objetos ou funções, a linguagem requer que os mesmos estejam contidos num *"array"* em que utilizam a posição destes no *array"* para evitar ambiguidades, no caso de existirem dois ou mais objetos com o mesmo tipo.
A execução do programa é representada visualmente pela deslocação dos agentes computacionais (os robôs) e pela sua interação e manipulação dos objetos disponíveis numa função. Infelizmente, a colocação dos objetos e a deslocação dos robôs não necessitam de ser muito organizadas.
Tendo em conta esta informação, as diferenças entre o protótipo e o Toontalk são:

- Toontalk utiliza *"arrays"* como uma estrutura de dados que armazena qualquer tipo de objeto. No protótipo é necessário fazer a distinção dos tipos de dados a guardar dentro de *"arrays"*;
- Toontalk não utiliza "booleans"ou qualquer tipo de valor booleano na construção de um programa, mas aplica lógica computacional de forma oculta com as balanças e a execução dos robôs. O protótipo utiliza "booleans"diretamente podendo ser aplicados em expressões ou armazenados dentro de variáveis;
- O posicionamento dos objetos numa programa em Toontalk é irrelevante, à exceção em tarefas que utilizem as posições de um *"array"* no padrão de tarefas a executar por um robô. Já no protótipo, o posicionamento dos objetos é extremamente importante para assegurar que o código é executável;
- Caixas presentes no Toontalk são os objetos mais semelhantes a variáveis dentro da linguagem do jogo, já que estes podem guardar informação, alterar informação guardada dentro deles e ser referenciada utilizando as posições disponíveis. O protótipo para além de *"arrays"*, utiliza também variáveis que guardam números inteiros, números de virgula flutuante, *"Strings"*, caracteres e "booleans";

- Toontalk consegue abrir e guardar objetos no disco rígido, o que permite criar programas que utilizam imagens. O protótipo só pode utilizar os tipos de objetos definidos na linguagem visual, não existindo um objeto que guarde a informação de uma imagem, visto não ter acesso a instruções que permitam abrir ou guardar ficheiros no disco rígido diretamente.
- Toontalk utiliza a metodologia de "programação por exemplo"para construir programas, enquanto que o protótipo utiliza uma abordagem baseada em gramáticas, utilizada em quase todas as linguagens de programação por texto;
- Toontalk não disponibiliza a capacidade de visualizar o código dos seus programas numa linguagem de programação textual. O protótipo consegue mostrar os seus programas em código Python.

5.2.2 Lightbot

Os programas construidos neste jogo estão diretamente relacionados com a deslocação de um robô, com o intuito de concretizar um único objetivo: deslocar o robô para posições específicas no mapa de cada nível e acender a sua lâmpada nestas posições. Considerando estes factos, a comparação que se pode fazer do Lightbot com o protótipo é a seguinte:

- Lightbot não utiliza variáveis e por consequência não utiliza *"arrays"*. O protótipo utiliza variáveis e *"arrays"*;
- Lightbot não utiliza ciclos para repetir um conjunto de instruções "n"vezes. O protótipo permite o uso de ciclos *"while"* e *"for"*;
- Lightbot não utiliza instruções condicionais. O protótipo permite o uso de instruções condicionais *"if"* e *"switch"*;
- Não é possível utilizar funções recursivas no Lightbot por falta de instruções condicionais. O protótipo permite utilizar funções recursivamente;
- Lightbot utiliza funções sem argumentos para executar um conjunto de instruções, mas não permite reutilizar programas criados noutros níveis. O protótipo permite o uso de funções, com ou sem argumentos, para executar programas criados ou já existentes;
- Lightbot não consegue gerar código dos programas construídos em cada nível numa linguagem de programação textual. O protótipo consegue converter os programas construídos em código Python;
- Lightbot separa a área onde é feita a construção de um programa da área onde a execução visual do programa é realizada. Os objetos que compõem um programa no protótipo, são colocados na mesma zona gráfica onde ocorre a animação da execução do código;
- Lightbot utiliza apenas ícones gráficos na programação. No protótipo a colocação dos objetos gráficos não é o suficiente, sendo necessário o preenchimento da informação guardada em alguns dos objetos da linguagem visual do jogo.

5.2.3 Jahooma's LogicBox

No jogo "Johooma's LogicBox"cada programa é construído com o objetivo de receber uma palavra (uma *"string"*) e realizar uma transformação ou modificação específica da mesma, sendo o puzzle de cada nível

o objetivo a atingir. A programação neste jogo é feita colocando caixas com setas (funções) nas quais irão receber uma palavra que será alterada e enviado o resultado obtido na direção de uma das suas setas. As diferenças entre este jogo e o protótipo são as seguintes:

- LogicBox não utiliza variáveis ou *"arrays"*. O protótipo usa;
- LogicBox não permite o uso de funções recursivamente, ao contrário do protótipo;
- LogicBox não gera código equivalente aos programas criados numa linguagem textual de programação, enquanto o protótipo gera código Python;
- LogicBox só utiliza os objetos gráficos para programar, sendo por vezes necessário alterar a direção da(s) seta(s) que cada caixa contém. O protótipo não utiliza apenas objetos gráficos para programar, requer por vezes texto;
- No LogicBox os argumentos das funções estão restritas a receberem apenas uma palavra. No protótipo as funções podem receber um ou mais argumentos ou nenhum;
- No LogicBox os programas nunca utilizam números. No protótipo podem ser utilizados números, letras, etc.;
- LogicBox permite o uso de ciclos, mas não contem instruções que permitam controlar especificamente a execução destes, ao contrário do protótipo.

6

Trabalho futuro e conclusões

Tendo em conta os problemas observados no capítulo anterior e as comparações feitas, o protótipo tem o potencial de utilizar a maioria dos conceitos básicos de programação na construção de casas, no entanto, ainda existem alguns problemas por resolver, abrindo a possibilidade para realizar melhorias e/ou criar novas funcionalidades no protótipo, nomeadamente:

1. Criar um compilador para permitir a execução direta dos programas gráficos e, por conseguinte, adicionar funcionalidades para o tratamento de erros de compilação (saber quais os objetos responsáveis pelo erro, outras informações para debugging, etc.) assim como também, facilitar a passagem do código de casas para outras linguagens;
2. Fazer alterações na linguagem para permitir a visualização da informação guardada dentro destes e permitir o uso de *"ifs"/"switches"* dentro de *"ifs"/"switches"*. Uma solução (não muito elegante) é efetuar a gestão do espaço da casa automaticamente (subir andares) à medida que são colocados novos blocos *"if"/"switch"* dentro de um bloco *"if"/"switch"*;
3. Criar uma interface gráfica adequada ao jogo, que permita arrastar objetos para a casa, editá-los e visualizar a informação contida dentro destes de forma prática;

4. Testar o jogo com novos programadores para certificar que os mesmos estão a aprender os conceitos de programação e conforme o *feedback* efetuar alterações à linguagem ou ao jogo;
5. Adicionar novos ícones à linguagem;
6. Criar um editor de casas mais flexível e acessível (ver ponto 2);
7. Criar um gerador de código destacado por cores, semelhante ao utilizado no Flowgorithm, para visualizar melhor quais são os objetos ou áreas da casa correspondentes ao código gerado.

Bibliografia

[1] Website da linguagem visual scratch. https://scratch.mit.edu/. Acedido em: 3/3/2016.

[2] Página oficial da linguagem de programação visual forms/3. http://web.engr.oregonstate.edu/ burnett/Forms3/forms3.html. Acedido em: 21/8/2016.

[3] Marc-Alexander Najork. Programming in three dimensions. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.27.1619&rep=rep1&type=pdf, 1994. Artigo acerca de uma linguagem visual em 3D com uma boa introdução em linguagens visuais.

[4] Shi-Kuo Chang, editor. *Visual Languages and Visual Programming*. Springer, 1990.

[5] Marat Boshernitsan and Michael S. Downes. Visual programming languages: a survey. Technical Report UCB/CSD-04-1368, EECS Department, University of California, Berkeley, Dec 2004.

[6] *Reflections on the Teaching of Programming: Methods and Implementations (Lecture Notes in Computer Science / Programming and Software Engineering)*. Springer, 2008.

[7] Página oficial da linguagem visual flowgoritm. http://www.flowgorithm.org/. Acedido em: 14/3/2016.

[8] Kirsten N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, 8(1):109–142, 1997.

[9] Kang Zhang. *Visual Languages and Applications*. Springer, 2007.

[10] Toontalk and logo. http://el.media.mit.edu/logo-foundation/resources/papers/pdf/toontalk.pdf. Acedido em: 21/6/2016.

[11] Margaret Burnett, John Atwood, Rebecca Walpole Djang, Herkimer Gottfried, James Reichwein, and Sherry Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. ftp://ftp.cs.orst.edu/pub/burnett/ForJFP/JFP.fordistrib.pdf, 2001. Acedido em: 25/11/2015.

[12] Página da wikipédia acerca da linguagem de programação visual flowgorithm. https://en.wikipedia.org/wiki/Flowgorithm. Acedido em: 21/8/2016.

[13] Lista de funções matemáticas ou funcionais disponíveis no flowgorithm. http://www.flowgorithm.org/documentation/intrinsic-functions.htm. Acedido em: 14/3/2016.

[14] Página da wikipédia acerca do jogo de programação toontalk. https://en.wikipedia.org/wiki/ToonTalk. Acedido em: 21/8/2016.

[15] Página da wikipédia acerca da linguagem de programação visual scratch. https://en.wikipedia.org/wiki/Scratch_(programming_language). Acedido em: 21/8/2016.

[16] Página da wikipédia acerca de programação de computadores. https://en.wikipedia.org/wiki/Computer_programming. Acedido em: 4/7/2016.

[17] Página da wikipédia acerca de um programa de computador. https://en.wikipedia.org/wiki/Computer_program. Acedido em: 4/7/2016.

[18] Página da wikipédia acerca de linguagens de programação. https://en.wikipedia.org/wiki/Programming_language. Acedido em: 4/7/2016.

[19] Kirsti Ala-Mutka. Problems in learning and teaching programming - problems in learning and teaching programming. https://www.cs.tut.fi/ edge/literature_study.pdf, 2004.

[20] Orivaldo de Lira Tavares, Crediné Silva de Menezes, and Rosane Aragon de Nevado. Pedagogical architectures to support the process of teaching and learning of computer programming. In *2012 Frontiers in Education Conference Proceedings*, pages 1–6. IEEE, 2012.

[21] Melisa Koorsse, Charmain Cilliers, and André Calitz. Programming assistance tools to support the learning of it programming in south african secondary schools. *Computers & Education*, 82:162–178, 2015.

[22] Shirley Gibbs, Patricia Anthony, and Stuart Charters. Reflection on teaching it for non-computing students. *Procedia-Social and Behavioral Sciences*, 186:790–799, 2015.

[23] Douglas A Kranch. Teaching the novice programmer: A study of instructional sequences and perception. *Education and Information Technologies*, 17(3):291–313, 2012.

[24] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. In *ACM SIGCSE Bulletin*, volume 37, pages 14–18. ACM, 2005.

[25] Trevor D Collins and Pat Fung. A visual programming approach for teaching cognitive modelling. *Computers & Education*, 39(1):1–18, 2002.

[26] Página da wikipédia acerca da linguagem de programação apl. https://en.wikipedia.org/wiki/APL_(programming_language). Acedido em: 19/8/2016.

[27] Página da wikipédia acerca da linguagem de programação blockly. https://en.wikipedia.org/wiki/Blockly. Acedido em: 16/7/2016.

[28] Página da wikipédia acerca do website code.org. https://en.wikipedia.org/wiki/Code.org. Acedido em: 16/7/2016.

[29] Ken Kahn. A computer game to teach programming. 1999.

[30] Página com todos os puzzles do toontalk e descrição do que cada um ensina ao jogador. http://www.toontalk.com/English/puzlearn.htm. Acedido em: 21/6/2016.

[31] Introdução do jogo light-bot. http://www.willamette.edu/ fruehr/141/lightbotintro.html. Acedido em: 22/6/2016.

[32] Página oficial do jogo jahooma's logicbox. https://logicbox.jahooma.com/. Acedido em: 22/6/2016.

[33] Página da wikipédia acerca da série de jogos "incredible machine". https://pt.wikipedia.org/wiki/The_Incredible_Machine. Acedido em: 15/9/2015.

[34] Margaret Burnett. Software engineering for visual programming languages. *Handbook of Software Engineering and Knowledge Engineering*, 2:77–92, 2001.

[35] Gennaro Costagliola, Vincenzo Deufemia, and Giuseppe Polese. Visual language implementation through standard compiler–compiler techniques. *Journal of Visual Languages & Computing*, 18(2):165–226, 2007.

[36] Pavel Grigorenko, Ando Saabas, and Enn Tyugu. Cocovila–compiler-compiler for visual languages. *Electronic Notes in Theoretical Computer Science*, 141(4):137–142, 2005.

[37] Filiz Kalelioğlu. A new way of teaching programming skills to k-12 students: Code. org. *Computers in Human Behavior*, 52:200–210, 2015.

[38] Mikael Kindborg and Kevin McGee. Visual programming with analogical representations: Inspirations from a semiotic analysis of comics. *Journal of Visual Languages & Computing*, 18(2):99–125, 2007.

[39] José María Rodríguez Corral, Antón Civit Balcells, Arturo Morgado Estévez, Gabriel Jiménez Moreno, and María José Ferreiro Ramos. A game-based approach to the teaching of object-oriented programming languages. *Computers & Education*, 73:83–92, 2014.

[40] Hai-Ning Liang, Jim Morey, and Kamran Sedig. Using visual tiling patterns to support the teaching of programming concepts. In *Teaching, Assessment and Learning for Engineering (TALE), 2012 IEEE International Conference on*, pages W1B–5. IEEE, 2012.

[41] Eiji Nunohiro, Kotaro Matsushita, Kenneth J Mackin, and Masanori Ohshiro. Development of game-based learning features in programming learning support system. *Artificial Life and Robotics*, 17(3-4):373–377, 2013.

[42] Cagin Kazimoglu, Mary Kiernan, Liz Bacon, and Lachlan Mackinnon. A serious game for developing computational thinking and learning introductory computer programming. *Procedia-Social and Behavioral Sciences*, 47:1991–1999, 2012.

UNIVERSIDADE DE ÉVORA
INSTITUTO DE INVESTIGAÇÃO
E FORMAÇÃO AVANÇADA

Contactos:
Universidade de Évora
Instituto de Investigação e Formação Avançada — IIFA
Palácio do Vimioso | Largo Marquês de Marialva, Apart. 94
7002 - 554 Évora | Portugal
Tel: (+351) 266 706 581
Fax: (+351) 266 744 677
email: iifa@uevora.pt