JShapes: Biblioteca para apoio ao ensino da programação orientada por objectos

José Manuel de Campos Lages Garcia Simão

Equiparado a Assistente de 2º Triénio do
Instituto Superior de Engenharia de Lisboa

Submetido no âmbito do concurso de provas públicas para provimento de quatro lugares de professores-adjuntos do mapa de pessoal docente do Instituto Superior de Engenharia de Lisboa, publicado no Edital n.º 803/2009 do Diário da República n.º 146, Série II de 2009-07-30, e para cumprir o requisito da alínea b) do n.º 1 do artigo 25.º do Decreto -Lei n.º 185/81, de 1 de Julho

10 de Setembro de 2009

Conteúdo

Resumo

O ensino da programação orientada por objectos assenta nos conceitos de encapsulamento, herança e polimorfismo. Apesar de abstractos, estes conceitos, e as técnicas relacionadas, são propícios a serem apresentados através de exemplos interactivos, como são as aplicações com interface gráfica. Apesar da plataforma *Java* disponibilizar os recursos para a construção deste tipo de aplicações, o aluno não os consegue compreender na fase inicial do curso e por isso os exemplos tipicamente apresentados evitam este tipo de interacção.

Este documento discute as vantagens e dificuldades de utilizar a biblioteca pedagógica *JShapes* no ensino da programação orientada por objectos nas duas cadeiras de introdução à programação, nos cursos do Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores do Instituto Superior de Engenharia de Lisboa. Esta biblioteca divide-se em duas partes: *i)* Desenho de objectos gráficos e *ii)* Micro motor de jogo. São também discutidos aspecto de desenho e implementação da biblioteca *JShapes*.

A abordagem proposta consiste em três fases. Em cada uma delas o aluno explora um dos pilares da programação orientada por objectos. A biblioteca proporciona também um contacto inicial com o modelo de programação baseado em eventos.

Capítulo 1

Introdução

O desenvolvimento de *software* é uma actividade complexa. À medida que mais e melhores aplicações estão disponíveis, os utilizadores esperam que as próximas versões superem as anteriores em funcionalidade e qualidade de trabalho. Para atingir esse objectivo, a indústria e a comunidade de investigação, procuram formas inovadoras de resolver os problemas. Surgem assim modificações aos paradigmas de programação, alterações ao modelo de desenvolvimento, novas e mais complexas ferramentas. O ensino da informática está por isso sujeito a dois tipos de forças, por vezes opostas: se por um lado se quer transmitir os fundamentos das várias áreas, por outro também é objectivo que logo no final do primeiro ciclo de estudos o aluno possa desempenhar um papel relevante numa indústria de *software* em constante actualização tecnológica.

Transversal a estes aspectos são as expectativas que os alunos têm quanto aquilo que vão encontrar ao longo do curso, e em particular nas primeiras disciplinas. Nesta fase o aluno não conhece os desafios reais que tem pela frente mas já tem a experiência de ser utilizador de um mundo informático cada vez mais interactivo.

Nas três licenciaturas do Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores (DEETC) do Instituto Superior de Engenharia de Lisboa (ISEL) o ensino dos fundamentos da programação tem por base o modelo *imperative-first* tal como proposto pelo *IEEE - Joint Task Force on Computing Curricula* [1]. A linguagem de suporte é a linguagem *Java*™ [2]. Na unidade curricular Programação (PG) são apresentados os elementos da linguagem *Java* (e.g. expressões, estruturas de controlo de fluxo), e é introduzida a noção de classe e objecto. A unidade curricular Programação Orientada por Objectos (POO) explica os conceitos de herança e de polimorfismo para além de um conjunto de temas complementares (e.g. tratamento de excepções). Estas duas unidades curriculares são comuns às três licenciaturas do DEETC: Licenciatura em Engenharia Informática e de Computadores (LEIC), Licenciatura em Engenharia de Electrónica e Telecomunicações e de Computadores (LEETC) e Licenciatura em Engenharia de Redes de Comunicação e Multimédia (LERCM). A unidade curricular Algoritmos e

Estruturas de Dados da LEIC, apresenta estruturas de dados fundamentais (e.g. listas, árvores), algoritmos de pesquisa e métodos para análise da complexidade de algoritmos.

1.1 O ensino da programação imperativa no DEETC

A utilização na unidade curricular de Programação de uma linguagem amplamente utilizada na indústria, tem a vantagem de se conseguir ensinar os conceitos ao mesmo tempo que o aluno ganha experiência com uma linguagem utilizada em diferentes contextos. Contudo, e exactamente pelas características anteriores, a generalidade das bibliotecas da plataforma *Java* usa construções e impõe técnicas de utilização que o aluno não pode ainda compreender. Um exemplo simples mas paradigmático desde problema foi a recolha de valores (e.g. inteiros, reais, sequências de caracteres) até à versão 5.0. Os texto pedagógicos resolviam este assunto fornecendo classes de fachada. Walter Savitch, por exemplo, nas primeiras três edições do seu livro [3], usa nos exemplos a classe `SavitchIn` para recolha de valores do teclado. A classe Scanner, introduzida na versão 5.0, resolve em parte este problema. Em parte porque para a utilizar ainda é preciso que o aluno use o operador `new`, numa fase em que o operador `&&` e a instrução `if` ainda são um problema.

Os primeiros programas que o aluno de Programação consegue fazer lidam apenas com interacções em modo de texto com o ecrã, focando-o na concepção do algoritmo e consequentemente nas construções sintácticas da linguagem. É, contudo, inegável que o mundo informático a que o aluno está hoje exposto, tem cada vez mais conteúdos multimédia. Pelo facto dos resultados visíveis serem apenas caracteres na consola, faz com que os programas produzidos tenham pouco a ver com a experiência de utilização de computadores que o aluno tipicamente já tem antes de ingressar no curso.

O tipo de interacção descrito anteriormente não está limitado pelos serviços da plataforma *Java*, mas sim pela impossibilidade de, em Programação, não ser possível transmitir ao aluno todos os conhecimentos necessários para utilizar a biblioteca de interfaces gráficas da plataforma (i.e. a biblioteca *Swing*). Savitch [3] sugere a utilização de *applets*. Esta estratégia obriga a apresentar mais construções "mágicas" como é o caso das palavras reservadas `extends` e `implements`, para além do aspecto menor que seria a utilização de outra aplicação para visualizar as *applet* (e.g. *appletviewer*).

1.2 O ensino da programação orientada por objectos no DEETC

As unidades curriculares de Programação e Programação Orientada por Objectos contribuem em conjunto para o ensino dos conceitos e técnicas da programação orientada por objectos (i.e. abstracção, encapsulamento, herança e polimorfismo). Apesar de não ser actualmente seguida uma abordagem *object-first* [1], é convicção do autor que há mais

vantagens do que desvantagens em introduzir estes temas nas cadeiras introdutórias à programação. Esta convicção tem suporte na observação de que as plataformas de execução modernas (e.g. Java e .NET), as quais os futuros licenciados e mestres usaram directa e indirectamente no seu dia a dia profissional, são por natureza orientadas por objectos, desde o sistema de tipos, até às bibliotecas que servem de suporte ao desenvolvimento aplicacional. No próprio documento do IEEE [1] é referido que o atraso na introdução do estilo de programação orientada por objectos é uma fraqueza da aplicação estrita do modelo *imperative-first*.

A introdução ao objectos em Programação, é feita através de exemplos abstractos (e.g. Departamento, Turma) e sem qualquer preocupação com a sua representação visual. Também aqui o objectivo é que o aluno se concentre nos princípios básicos do encapsulamento, mas que tenha a capacidade de começar a ver os programas que produz como um conjunto de objectos que colaboram entre si. Esta ideia é complementada na unidade curricular Programação Orientada por Objectos, a qual apresenta a herança de implementação e de interface como suporte ao polimorfismo com base em sub-tipos. É ainda abordado o polimorfismo paramétrico, o qual serve de suporte aos contentores da biblioteca que acompanham a linguagem, e o modelo de programação baseado em eventos, tendo como suporte a biblioteca *Swing*.

Finalmente, na unidade curricular optativa Modelação e Padrões de Desenho, a qual os alunos poderão fazer no quarto ou sexto semestre da LEIC, são discutidas técnicas para construir sistemas mais genéricos e por isso mais flexíveis, designadamente através dos Padrões de Desenho descritos por Erich Gamma *et al.* [4].

1.3 Biblioteca de apoio ao ensino da programação

O ensino da programação orientada por objectos têm se mostrado propício a diferentes abordagens [5, 6, 7, 8]. Ao contrário do paradigma exclusivamente procedimental, a programação orientada por objectos assenta em aspectos mais abstractos como o encapsulamento e o polimorfismo. Na verdade, a qualidade de uma aplicação não se mede apenas pelo sucesso no uso de determinado algoritmo mas também (e cada vez mais) pela desenho e organização das suas classes e componentes. Este último aspecto tem de ser treinado desde cedo, tal como o desenho de um bom algoritmo.

Este documento discute as vantagens e dificuldades de utilizar a biblioteca pedagógica *JShapes* como suporte ao ensino da programação orientada por objectos nas duas cadeiras de introdução à programação, nos cursos do DEETC. A abordagem proposta tem por base dois padrões pedagógicos [9]: *Early Bird* e *Toy Box*. O padrão *Early Bird* defende que os assuntos mais importantes devem ser apresentados em primeiro lugar e várias vezes ao longo do caminho. O padrão *Toy Box* visa dar aos alunos a possibilidade de trabalhar com ferramentas (pedagógicas) as quais eles ainda não têm capacidade de construir, mas que

os ajudam a perceber o caminho que têm pela frente.

A biblioteca *JShapes* foi desenhada e implementada na sua versão protótipo pelo autor. Esta biblioteca divide-se em duas partes:

1. Desenho de objectos gráficos
2. Micro motor de jogo com animação de objectos cujos tipos são definidos pelo utilizador

O desenho de objectos gráficos inclui formas geométricas, mapas de pixeis e imagens carregadas a partir de ficheiro. Com estes elementos o aluno pode escrever programas que recorram ao desenho de formas geométricas para, por exemplo, animar modelos físicos ou pequenos jogos. Inicialmente a utilização da biblioteca consiste apenas em apresentar os objectos gráficos da janela da aplicação, mas quando for importante começar a ganhar mais experiência a tratar com objectos, o aluno poderá então tomar maior controlo sobre o objecto já criado e mudar, por exemplo, a sua localização e cores. O mapa de pixeis é uma fonte de exercícios com *arrays* já que é possível aceder e modificar os pixeis através de *arrays* de inteiros ou *arrays* de objectos representativos da cor.

Um segundo nível de utilização da biblioteca é definir as classes de objectos para interagirem num micro motor de jogos, o qual divide os objectos em duas categorias: estáticos e em movimento. Este ambiente precisa que os objectos saibam responder a algumas mensagens (e.g. qual a forma geométrica que os representa, o que fazer em caso de colisão com o objecto do tipo T), definindo a assinatura de um conjunto predefinido de métodos. Para que esta parte da biblioteca possa ser usada na unidade curricular Programação, não é preciso que os tipos definidos implementem qualquer interface ou estendam qualquer classe abstracta. A biblioteca gere os objectos independentemente do seu tipo concreto, chamando sobre eles as operações relevantes em cada momento. O aluno toma contacto pela primeira vez com o conceito de interface tendo de cumprir uma meta-descrição na definição do tipo que descreve o elemento do jogo, caso contrário o objecto será ignorado. A interacção entre a micro motor de jogo e os objectos é feita recorrendo à noção de evento. O código que o aluno tem de escrever não controla a totalidade do fluxo de execução, sendo chamado para reagir em função de algum acontecimento. É assim proporcionado um primeiro contacto com o modelo de programação baseado em eventos.

Finalmente a biblioteca pode ser utilizada na fase inicial do actual programa de Programação Orientada por Objecto, seguindo diferentes abordagens:

1. Mostrar algumas das fraquezas do modelo descrito anteriormente e com o qual os alunos entretanto já se familiarizaram. Uma das limitações é o difícil reaproveitamento de código já que os tipos que caracterizam os objectos não herdam comportamento de qualquer classe base (excepto `Object`). Outra é que a compatibilidade dos tipos só é determinada em tempo de compilação.

2. Explicar a necessidade do polimorfismo para, segundo as regras da linguagem, realizar este ou outro motor de jogo. O aluno já trabalhou indirectamente com este assunto e está em condições de compreender melhor a sua necessidade.

3. Estudar a relação hierárquica dos objectos gráficos dando ênfase à utilização de classes abstractas e métodos *template* [4].

1.4 Organização do documento

Os Capítulos 2 e 3 apresentam exemplos de utilização da biblioteca *JShapes* no contexto das unidades curriculares de Programção e Programação Orientada por Objectos, respectivamente. O Capítulo 4 apresenta a arquitectura e aborda alguns aspectos de implementação da biblioteca. As conclusões são apresentadas no Capítulo 5.

Capítulo 2

Utilização da biblioteca *JShapes* em PG

Neste capítulo são apresentadas as vantagens de usar a biblioteca *JShapes* na componente prática da unidade curricular Programação, a partir da 3ª semana. Nesta fase o aluno já tem experiência de utilização de tipos referência elementares como o `String` e o `Scanner` e sabe utilizar métodos de instância, os quais são apresentados como serviços prestados por determinado objecto.

2.1 A classe `BGraphics`

O primeiro contacto com a biblioteca é através da classe `BGraphics`. A Listagem 2.1 apresenta um troço de código onde são desenhados dois círculos centrados na área de desenho.

```
Scanner s = new Scanner(System.in);
int width = s.nextInt();
int height = s.nextInt();
int radius = s.nextInt();

BGraphics graphics = new BGraphics("Circles", width, height);
graphics.fillCircle(width/2-radius, height/2-radius, radius);
graphics.setFillColor(BColor.GREEN);
graphics.fillCircle(width/2-radius/2, height/2-radius/2, radius/2);
```

Listagem 2.1: Exemplo simples com a classe BGraphics

O aluno conhece o sistema de coordenadas a duas dimensões e por isso é preciso apenas apresentar os métodos disponíveis, referindo que as coordenadas indicadas no momento de criação dos diferentes objectos gráficos correspondem ao canto superior esquerdo de um rectângulo virtual que delimita o objecto. Esta forma de interagir é igual às *frameworks* para interfaces gráficas com as quais o aluno irá mais tarde trabalhar. É ainda preciso

explicar que o ponto (0,0) está no campo superior esquerdo da área gráfica; o equivalente ao 4º quadrante mas onde o eixo vertical cresce no sentido descendente.

No momento em que forem introduzidos os ciclos será possível realizar animações simples. O exemplo da Listagem 2.2 mostra a animação do movimento de um projéctil segundo as leis da física.

```
int x0 = 50, y0 = 250, v0 = 50, a0 = 60;
double g = 9.8, x, y;
double angle0 = a0 * Math.PI/180;
for (double t=0; t<20; t+=0.5) {
   x = x0 + v0 * Math.cos(angle0) * t;
   y = y0 - (v0 * Math.sin(angle0) * t - 0.5 * g * t*t);
   graphics.fillCircle((int)x, (int)y, 5);
   graphics.delay(250);
}
```

Listagem 2.2: Trajectória de projéctil

A classe `BGraphics` é composta por um conjunto de métodos fábrica, os quais criam objectos para serem desenhados no ecrã gráfico. A família de objectos inclui formas geométricas, mapa de pixeis e imagens carregadas a partir de ficheiro.

Durante aproximadamente seis semanas o aluno é utilizador de objectos. A biblioteca *JShapes* enriquece essa experiência de uma forma intuitiva, dando controlo sobre as figuras criadas com os métodos fábrica da classe `BGraphics`. Por exemplo, a Listagem 2.3 apresenta um troço de código cujo objectivo é idêntico ao exemplo da Listagem 2.2 mas onde existe apenas um objecto círculo a simular o projéctil.

```
...
BCircle projectil = graphics.fillCircle(x0, y0, 5);
for (double t=0; t<10; t+=0.5) {
  x = ...;  y = ...;
  projectil.move((int)x, (int)y);
  graphics.delay(250);
}
```

Listagem 2.3: Trajectória de projéctil

Com este tipo de experiências o aluno vai ganhando a percepção de objecto como algo que agrega estado e funcionalidade. Nesta fase da aprendizagem é difícil usar o mesmo tipo de analogia com a classe `Scanner` ou a classe `PrintStream` (classe da variável estática `System.out`, sobre a qual nem sequer é habitual falar). Apesar de as usar desde o primeiro programa que recolheu dados do teclado ou os enviou para ecrã, o estado associado a estes objectos é algo difuso. No caso das figuras criadas através da classe `BGraphics` é notório que o seu estado inclui o par (x, y) a partir do qual é desenhada a figura. O método `move` modifica estes dados.

Qualquer figura criada através da classe `BGraphics` pode ser movida. No caso das figuras geométricas também poderão ser alteradas outras propriedades do seu aspecto (e.g. raio, cor de preenchimento). A cor, nas suas componentes de vermelho, verde e azul, é representada por objectos do tipo `BColor` mas também pode ser alterada através da passagem de um valor inteiro, onde as três componentes referidas são indicadas nos três *bytes* de menor peso.

2.2 Utilização do micro motor de jogo em PG

No programa actual de PG, a segunda parte do semestre começa com a definição de classes e respectivos métodos. Depois do aluno ter utilizado alguns tipos elementares, é agora altura de definir as suas próprias classes.

Com o modelo de objectos apresentado anteriormente já é possível o aluno realizar diferentes tipos de aplicações interactivas. Os jogos são para muitos alunos um desafio interessante. O processo pelo qual passam tem o potencial de os fazer assimilar os conceitos e as técnicas que fazem parte dos objectivos da disciplina, ao mesmo tempo que produzem algo com uma componente lúdica.

A segunda fase de utilização da biblioteca *JShapes* consiste na definição de um conjunto de classes com o objectivo de representarem os elementos de um jogo. O objectivo é concentrar o aluno apenas na definição dos tipos relevantes para o jogo em causa. Em cada jogo podem existir duas categorias de objectos: parados ou em movimento. Os elementos do jogo e respectivas classes têm de se enquadrar numa destas duas categorias. A biblioteca trata de acções comuns como animar os objectos que se movem, detectar colisões ou reagir a teclas, tudo em função de código definido pelas classes que representam os elementos de jogo.

A classe `GameEnviromentPG` é a fachada para interagir com esta parte da biblioteca. Através dela é possível configurar o jogo (e.g. adicionar níveis, adicionar elementos, remover elementos) e obter referência para um elemento através do seu nome.

2.2.1 Modelo baseado em eventos

A utilização do motor de jogo implica a utilização de um modelo de programação baseado em eventos. Para o aluno este tipo de interacção é uma novidade. Desde os primeiros programas que é o aluno quem controla o rumo dos acontecimentos, sabendo que o seu programa tem um ponto de entrada e que a partir daí é o seu código que toma a iniciativa de realizar todas as acções. A abordagem sugerida implica a inversão do controlo no rumo das acções, passando a ser a biblioteca que determina em que momento determinado método é chamado.

Esta mudança de paradigma é, contudo, apresentada de uma forma simplifica, deixando

para mais tarde no decorrer do curso a discussão dos padrões de desenho [4] e das técnicas (e.g. *delegates* da plataforma .NET [10]) que estão subjacentes a este tipo de programação nas plataformas virtuais de execução modernas.

Hoje em dia os profissionais na área do desenvolvimento de *software* têm de lidar com plataformas e bibliotecas onde a inversão de controlo é uma realidade cada vez mais presente. Ao longo dos anos as expectativas que os utilizadores têm das aplicações tem vindo sempre a aumentar. Para responder a esta desafio o desenvolvimento de *software* é cada vez mais baseado em bibliotecas extensíveis (i.e. *frameworks*), as quais fornecem os mecanismos comuns a um domínio do problema. O programador tem de conhecer os pontos de extensibilidade da biblioteca para criar uma aplicação completa. Estas *frameworks* estão presentes desde os ambientes para *desktop* (sendo o exemplo mais paradigmático as bibliotecas para interfaces gráficas) até às aplicações *web* e empresariais onde existe um variedade de contentores aplicacionais que tratam problemas comuns deste tipo de aplicações.

2.2.2 Definição de classes

Uma vez que a biblioteca é independente do jogo (apesar de facilitar o desenvolvimento de uns mais do que outros), são à partida definidas regras para as duas categorias de objectos que podem existir. Não são impostas regras ao nome da classe, mas em função da categoria do objecto terão de ser definidos métodos com determinada assinatura.

Tomemos por base o caso de estudo de definir as classes que caracterizam os elementos do jogo *Arkanoid* [11]. A Listagem 2.4 apresenta a definição da classe `Brick` a qual caracteriza um tijolo no jogo.

```
public class Brick {
   private BRectangle rectangle;
   private boolean oneHit;
   public Brick(int x, int y, int w, int h) {
      oneHit = false;
      this.rectangle = new BRectangle(x, y, w, h,BColor.RED,BColor.BLUE,1);
   }
   public void onCollision(OnCollisionParams params, Ball b) {
      if (oneHit)
         GameEnviromentPG.removeOjectFromLevel(this);
      else {
         oneHit = true;
         rectangle.setFillColor(BColor.GREEN);
      }
   }
   public BRectangle getShape() { return rectangle; }
}
```

Listagem 2.4: Definição da classe `Brick`

Conceptualmente o ambiente de jogo não é composto por objectos gráficos mas sim por objectos que representam os diferentes elementos do jogo (i.e. bola, tijolo). Cada classe tem por isso de definir qual o aspecto do elemento e qual o seu comportamento, promovendo a separação entre os aspectos visuais e a lógica subjacente ao elemento. A Secção 2.2.4 descreve a forma como o jogo é preenchido com os diferentes objectos.

A Tabela 2.1 apresenta os métodos que têm de ser definidos em função da categoria do elemento de jogo.

Categoria	Métodos	Opcional
Estático	`Shape getShape()`	
	`void onCollision(OnCollisionParams params, Object object)`	√
Com movimento	*métodos da categoria "Estático"*	
	`void onMove(Point currentPoint)`	√
	`void onKey(Point currentPoint, Key keyPressed)`	√
	`void onKeyUp(Point currentPoint)`	√
	`onKeyDown`, `onKeyLeft`, `onKeyRight`	√

Tabela 2.1: Métodos que definem a categoria do elemento do jogo.

Cada categoria define um conjunto de métodos os quais podem ser obrigatórios ou opcionais. Por exemplo, para que um objecto se enquadre na categoria "Com movimento", a classe da qual o objecto é instância terá de definir os métodos da categoria "Estático" e pelo menos um dos métodos opcionais.

Com esta abordagem as conceitos de interface e polimorfismo são informalmente apresentados. No que respeita à noção de interface, o enquadramento de um objecto numa categoria leva a que diferentes objectos tenham obrigatoriamente em comum uma parte da sua interface pública. O polimorfismo está também presente porque, mesmo pertencendo à mesma categoria, os objectos reagem de maneira diferente ao mesmo conjunto de eventos. Estas ideias chave podem assim ser abordadas informalmente sem questões sintácticas subjacentes.

Apesar do tipo definido na Listagem 2.4 estender directamente de `Object`, a biblioteca trata de verificar, em tempo de execução, se os objectos que estão a ser adicionados obedecem às regras descritas anteriormente. Aos objectos cujo tipo não se enquadra numa destas categorias a biblioteca não fornece o serviço em causa (e.g. exibir objecto gráfico, enviar tecla premida). A definição da classe precisa por isso de ser compatível com a meta descrição do elemento de jogo, independentemente da implementação.

Reacção a colisões

A reacção a colisões é fortemente tipificada. Apesar da Tabela 2.1 apresentar na assinatura do método `onCollision` o nome *Object*, tal não corresponde à classe `Object` da plataforma *Java*. Em vez disso, significa que qualquer tipo pode ser indicado na definição concreta de um elemento de jogo. Desde que o nome do método seja `onCollision` e o primeiro parâmetro seja do tipo `OnCollisionParams`, o tipo do segundo parâmetro pode ser qualquer um. É trabalho da biblioteca verificar, em tempo de execução, se existe algum método `onCollision` que corresponda aos elementos envolvidos na colisão.

Na Listagem 2.4 a especificação do código de colisão é feita indicando o tipo `Ball` como segundo parâmetro do método `onCollision`. Caso haja outro tipo de objecto a colidir com o tijolo, este método não será chamado pelo biblioteca.

Aspectos a ter em conta na definição dos métodos

Para cada elemento do jogo terá de, no mínimo, ser definido qual é o objecto gráfico que o representa, retornando-o no método `getShape`. Os objectos que aqui podem ser retornados são aqueles com os quais o aluno já ganhou experiência na primeira fase de utilização da biblioteca (e.g. `BCircle`, `BRectangle`, `BBitmap`) O objecto gráfico retornado tem de ser sempre o mesmo já que é responsabilidade da biblioteca adicionar e remover estes objectos do ecrã quando o elemento do jogo (e.g. Brick) é adicionado ou removido.

O método `onMove` e o conjunto de métodos `onKey` recebem o ponto actual do actor. Durante a execução destes métodos, este ponto pode ser modificado, reflectindo a estratégia de movimento do objecto ou a reacção à tecla premida. O nova localização só é definitiva se o objecto não colidir com qualquer outro objecto para o qual tenha definido um método `onCollision`. Caso ocorra colisão, a nova localização será determinada pelo código do método `onCollision` que vier a ser chamado.

Os métodos `onCollision` recebem um objecto do tipo `OnCollisionParams`, o qual tem descreve três informações: *i)* a posição do objecto gráfico, *ii)* uma *flag* indicando se esta é a última colisão e *iii)* o rectângulo com os dados da intersecção dos objectos gráficos (posição, altura e largura). Esta informação tem como principal destinatário os objectos com movimento mas também é passada aos restantes.

Quebra de encapsulamento

Na definição da classe `Brick` anteriormente apresentada, o método `getShape` retorna uma referência para estado interno do objecto. Esta não é uma boa prática e representa uma quebra de encapsulamento, promovida pelo facto dos tipos terem de retornar um objecto gráfico no método *getShape*. Outras opções poderiam ter sido seguidas para que o objecto não pudesse guardar uma referência para o objecto gráfico:

- Configuração em ficheiro. Neste caso os objectos gráficos associados a cada elemento do jogo seriam descritos à parte do código, sendo geridos exclusivamente pela biblioteca;

- Anotar a classe do elemento com o tipo de objecto gráfico que o representa. Esta opção evita a utilização do ficheiro, mas obriga a que seja apresentado um aspecto marginal nesta fase da aprendizagem;

- Fábrica de objectos gráficos. Em vez de um objecto gráfico seria retornado uma instância de uma fábrica especializada em determinado tipo de objecto gráfico.

A opção seguida simplifica o uso da biblioteca e a implementação de alguns jogos. Por exemplo, no caso de estudo seguido, o objecto que representa o tijolo não precisa de ser outro pelo facto de se pretender mudar a cor de preenchimento. Se qualquer uma das abordagens anteriores fosse escolhida, seria necessário remover o objecto e adicionar um de outro tipo para o qual o objecto gráfico já teria a cor pretendida.

2.2.3 Mais tipos do *Arkanoid*

Um dos elementos principais do *Arkanoid* é a bola. A Listagem 2.5 apresenta uma possível definição para este elemento do jogo.

```
public class Ball {
   private LinearMove movingStrategy;
   private BCircle shape;
   public Ball(int x, int y) {
      this.movingStrategy = new LinearMove();
      this.shape = new BCircle(x,y,10,BColor.GREEN,BColor.RED,1);
   }
   public void onCollision(OnCollisionParams params, Wall w) {
      if (w == GameEnviromentPG.getObjectByName("Floor"))
         GameEnviromentPG.actorDied();
      else
         onCollisionInternal(params);
   }
   public void onCollision(OnCollisionParams params, Brick brick) {
      onCollisionInternal(params);
   }
   public void onCollision(OnCollisionParams params, Racket racket) {
      onCollisionInternal(params);
   }
   public void onMove(GEPoint newPosition) {
      movingStrategy.onMove(newPosition);
   }
   public BCircle getShape() { return shape; }
   private void onCollisionInternal(OnCollisionParams params) {
```

```
        if (!params.isFinalCollision) return;
        if (params.intersectionInfo.getWidth() >=
            params.intersectionInfo.getHeight()) {
            movingStrategy.stickY(params);
            movingStrategy.invY();
        } else {
            movingStrategy.stickX(params);
            movingStrategy.invX();
}   }   }
```

Listagem 2.5: Definição da classe `Ball`

Para além do método `getShape` são definidas várias sobrecargas do método `onCollision` para os diferentes tipos de colisão que é relevante tratar.

Mesmo com um exemplo simples, como é o jogo *Arkanoid*, o aluno tem oportunidade de ganhar sentido crítico sobre os caminhos que pode seguir na implementação. No exemplo apresentado na Listagem 2.5 é utilizada a classe `LinearMove`, a qual seria definida no contexto do desenvolvimento do jogo. Ela tem informação sobre o incremento horizontal e vertical por cada movimento do elemento. Para além disso tem métodos que "colam" o elemento de jogo que colidiu com o outro. Este aspecto não é específico do elemento *Ball*. Os bónus (não apresentados) também podem usufruir dos serviços desta classe.

Esta não seria provavelmente a primeira solução de alguém que ainda se está a familiarizar com a programação em geral e com o conceito de classe e objecto em particular. Tal como se deve estimular o sentido crítico sobre os algoritmos que o aluno desenvolve, seja do ponto de vista de recursos (estimados) que ocupa (i.e. memória e capacidade de processador), também é importante discutir desde cedo estes aspectos de desenho/organização para que o aluno adquira sentido de estética e funcionalidade sobre as soluções a que chega.

2.2.4 Configuração e início do Jogo

A configuração do jogo consiste em definir tipos que se insiram numa das duas categorias da Tabela 2.2. A categoria "Jogo" representa a classe principal de cada jogo. É ela que tem a responsabilidade de adicionar objectos ao ambiente de jogo, os quais terão de cumprir as regras da categoria "Nível".

Categoria	Métodos	Opcional
Jogo	`void onGameSetup()`	
	`void onCollision()`	√
Nível	`int onLevelLoad()`	
	`void onNewLife()`	√

Tabela 2.2: Métodos dos tipos para configuração do jogo.

A Listagem 2.6 apresenta o ponto de entrada do jogo cujos tipos têm vindo a ser definidos.

```
public class Arkanoid {
   public void onLevelsSetup() {
      GameEnviromentPG.addLevelToGame(new LevelOne());
   }
   ...
   public static void main(String[] args) {
      GameEnviromentPG.run(30, 500, 500, "Arkanoid", new Arkanoid());
   }
}
```

Listagem 2.6: Classe principal do jogo Arkanoid

O controlo do fluxo de execução é passado à biblioteca *JShapes* através da chamada ao método `run` da classe `GameEnviromentPG`. O tipo do objecto a utilizar no método `run` tem de definir os métodos obrigatórios da categoria "Jogo" da Tabela 2.2. A Listagem 2.7 apresenta um exemplo de configuração de um nível no contexto do caso de estudo.

```
public class LevelOne  {
   public int onLevelLoad() {
      GameEnviromentPG.addObjectToLevel(new Wall(0, 0, 25, 500));
      ...
      int brickWidth = 64, brickHeight = 30, startX = 27, startY = 50;
      for (int lin=0; lin<5; ++lin)
         for (int col=0; col<7; ++col)
            GameEnviromentPG.addObjectToLevel(new
               Brick(startX+col*brickWidth, startY+lin*brickHeight,
                     brickWidth, brickHeight));
      // return value is the number of lives for this level
      return 3;
   }
   public void onNewLife() {
      Message.show("Get Ready!!");
      GameEnviromentPG.addObjectToLevel(new Racket(), "MyRacket");
      GameEnviromentPG.addObjectToLevel(new Ball(250,400));
   }
}
```

Listagem 2.7: Configuração dos elementos de jogo

Quando um novo objecto é adicionado é possível atribuir-lhe um nome. O nome é utilizado durante o decorrer do jogo para obter uma referência para o objecto que lhe está associado. Este aspecto é utilizado na classe `Ball` para determinar se a parede com a qual colidiu é o chão da arena de jogo.

O diagrama da Figura 2.1 mostra a dependência da classe `GameEnviromentPG` com as classes anteriormente definidas.

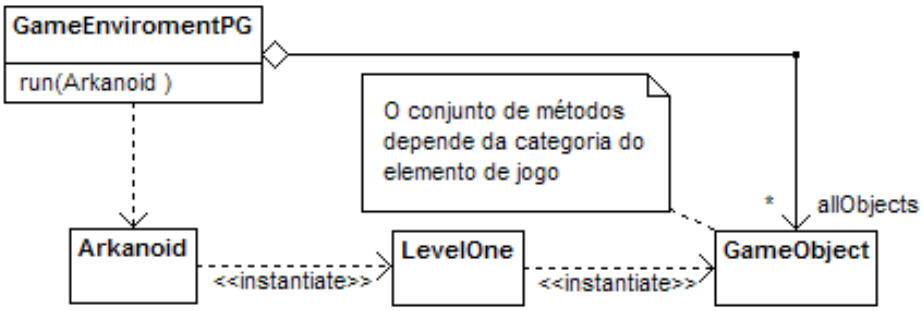

Figura 2.1: Tipos para configuração dos elementos de jogo

Capítulo 3

Utilização da biblioteca *JShapes* em POO

Este capítulo apresenta de que forma a biblioteca *JShapes* pode ser usada unidade curricular de Programação Orientada por Objectos para introduzir os conceitos de herança e polimorfismo, inclusive a sua ordem de apresentação. É abordada a reutilização de código e o problema da reacção a colisões usando o despacho dinâmico simples suportado pela linguagem.

3.1 Lições da utilização da biblioteca em PG

Passada a barreira inicial de ver um programa como um conjunto de objectos que interagem entre si, tendo por base a definição de uma classe, o aluno tem agora de aprender a utilizar as restantes técnicas da programação orientada por objectos. A experiência de utilização da biblioteca *JShapes* na unidade curricular de Programação deve permitir-lhe reflectir sobre os seguintes aspectos:

1. Difícil reutilização de código;
2. Compatibilidade das classes com as categorias pré-definidas só é determinada em tempo de execução;

O primeiro aspecto que deve ser realçado é o facto de existir comportamento e estado igual, ou semelhante, entre diferentes elementos do jogo. Por exemplo, a definição de um conjunto de classes para representar os diferentes tipos de tijolos do jogo *Arkanoid* é propícia à repetição de código.

A definição de uma classe que não se enquadre nas categorias pré-definidas resulta na não prestação de serviços ao objecto em causa. Isto acontecerá porque está em falta um ou mais métodos ou porque apesar do nome estar correcto, a quantidade ou tipo dos parâmetros não coincide com a meta-descrição da categoria. Apesar do problema principal

ser detectável em tempo de compilação, só em tempo de execução é que as consequências do erro são notórias.

3.2 Herança de implementação

Um ambiente onde não existe a relação de herança de implementação leva à repetição de código. No caso de estudo que tem vindo a ser apresentado, a definição de classes para representar diferentes tipos de tijolos levanta este problema. Considere-se um cenário onde existem três tipos de tijolos: inquebráveis, quebráveis ao fim de n impactos e os que quando quebram adicionam ao jogo um bónus que pode ser capturado pela raquete. As classes da Listagem 3.1 são uma possível concretização deste enunciado.

```
class Brick {
   private BRectangle rectangle;
   public Brick(int x, int y, int w, int h, BColor fillColor) {
      this.rectangle = ...;
   }
   public BRectangle getShape() { return rectangle; }
}
class HitBrick extends Brick {
   private int hitCounter;
   ...
   public void onCollision(OnCollisionParams params, Ball b) {
      if (hitCounter-- == 0)
         GameEnviromentPG.removeOjectFromLevel(this);
      else
         getShape().setFillColor(colors[hitCounter]);
   }
   public int getHitCounter() { return hitCounter; }
}
class BonusBrick extends HitBrick {
   ...
   public void onCollision(OnCollisionParams params, Ball b) {
      super.onCollision(params, b);
      if (getHitCounter() == 0)
         GameEnviromentPG.addObjectToLevel(new Bonus(...));
   }
}
```

Listagem 3.1: Diferentes tipos de tijolos

Tal como nos exemplos de Programação, as instâncias destas classes passam a participar no jogo através do uso da classe `GameEnviromentPG`.

Na Listagem 3.1 podem-se observar várias consequências resultantes da utilização da relação de herança. A classe `Brick` é usável por si só. Não define métodos de colisão

e por isso representa um tijolo inquebrável. A classe `HitBrick` assenta na definição anterior, acrescentando estado e comportamento. Finalmente a classe `BonusBrick` redefine comportamento, tirando partido da definição anterior.

Em Java a relação de herança dá suporte à reutilização de código mas também ao tratamento polimórfico dos objectos e das suas operações. A utilização da classe `GameEnviromentPG` permite que se separe claramente estes dois aspectos, apresentando ao aluno as vantagens de definir tipos com base noutros com o objectivo central de reutilizar código.

Apesar da relação de herança de implementação ser nesta fase apenas utilizada para reutilizar código, este exemplo serve também para continuar a enfatizar a importância da interface, introduzindo informalmente as diferenças entre a relação "is-a" e "has-a". A classe `BonusBrick`, por exemplo, poderia agregar um objecto `HitBrick`, mas isso não a qualificaria para ser usada no ambiente de jogo já que essa agregação não lhe fornece a implementação dos métodos `getShape` e `onCollision`.

3.3 Polimorfismo

Durante a utilização da classe `GameEnviromentPG` o aluno constata que a assinatura de métodos como `addObjectToLevel` e `removeObjectToLevel` recebem um parâmetro do tipo `Object`. Na unidade curricular de POO, e no seguimento da discussão sobre herança de implementação, o aluno terá oportunidade de perceber porque motivo o compilador não dá qualquer mensagem de erro na chamada a estes métodos. Não será contudo possível perceber como é que realmente os métodos são chamados já que os mecanismos de reflexão não podem ainda ser apresentados.

Para que a biblioteca continue a servir de bancada de teste e de aprendizagem em POO, a classe `GameEnviromentPOO` tem métodos com o mesmo nome da classe `GameEnviromentPG` mas os quais recebem parâmetros do tipo da interface `IGameObject`, apresentada no diagrama UML da Figura 3.1. A definição dos elementos do jogo terá por isso de ser feita implementando as interface `IGameObject` ou `IMovingObject` em função de se tratar de um objecto estático ou com movimento.

Apesar de em ambos os tipos de utilização o aluno perceber que existe um tratamento polimórfico dos objectos que representam os elementos do jogo, é preciso realçar a diferença fundamental entre utilizar a classe `GameEnviromentPG` e `GameEnviromentPOO`. No primeiro caso é a biblioteca que tem o ónus de verificar a compatibilidade do objecto. No segundo caso é o compilador que realiza essa verificação, não sendo possível compilar um programa que não obedeça às regras do sistema de tipos. Deve-se também referir que o código produzido é diferente não sendo apenas uma questão da altura em que a verificação é feita. Na verdade, usando a classe `GameEnviromentPG`, as chamadas aos métodos das classes que representam elementos do jogo têm um custo aproximadamente 100 vezes superior às que são realizadas sobre os objectos cujas classes implementam uma das interfaces.

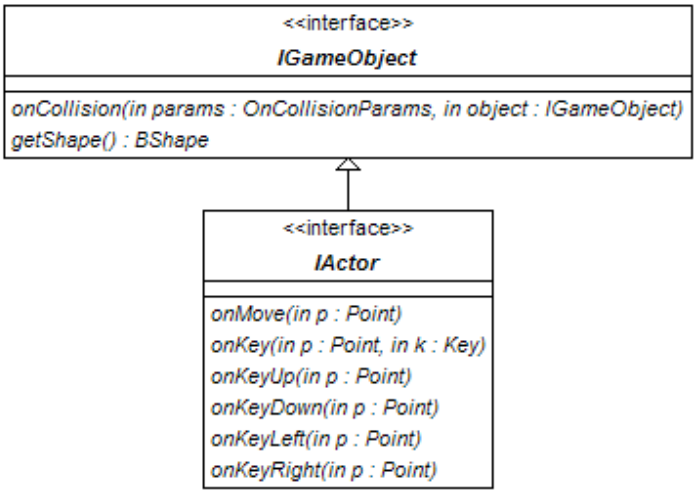

Figura 3.1: Interfaces definidas pela biblioteca JShapes

3.3.1 Classes abstractas

As classes cuja definição foi apresentada na Secção 2.2 podem omitir os métodos opcionais da Tabela 2.1. A utilização da classe `GameEnviromentPOO` impede este tipo de prática. Este problema tem como causa um problema recorrente no desenho de hierarquias de classes que é querer ter um comportamento por omissão para determinadas operações. O facto do aluno já ter trabalhado num ambiente onde isto era uma característica base, leva-o a perceber melhor a necessidade desta técnica. Usando as regras do sistema de tipos é possível obter um resultado semelhante através da definição de classes abstractas que implementam a interface pretendida e dão corpo vazio a cada um dos métodos. Por exemplo, para os elementos com movimento, os quais têm de definir a reacção a diferentes eventos, poderia ser definida a classe abstracta `AbstractMovingObject`.

Na definição original da classe `Ball`, presente na Listagem 2.5, é sugerido que o código que determina a nova localização da bola esteja definido numa classe própria, com o objectivo de reutilizar essa classe noutros elementos de jogo. Em POO, conhecendo a herança de implementação, e a possibilidade de criar classes abstractas, uma solução aparentemente mais simples seria definir os métodos da classe `LinearMove` na classe abstracta `AbstractMovingObject`, a qual seria a base de qualquer objectos com movimento. Esta solução dificultaria a alteração da estratégia de movimentação, obrigando a escrever código com vários testes, ou a retirar do ambiente de jogo objectos, adicionando outros com estratégias de movimentação diferentes. A aplicação ficaria por isso mais difícil de perceber, no caso dos vários testes, ou menos eficiente, no caso de ter de constantemente adicionar e remover object. A apresentação da técnica descrita pela padrão de desenho *Strategy* [4] serve para voltar a discutir sobre as vantagens e desvantagens de usar a relação de herança de implementação *versus* a de agregação, recorrendo a um problema cujo domínio é

familiar ao aluno e sobre qual ele já tem experiência prática.

3.3.2 Reacção a colisões

Na Listagem 2.4, a classe `Brick` define a mudança de estado de um tijolo quando a bola embate nele. Na Listagem 2.5, a classe `Ball` define a mudança de estado da bola quando embate num tijolo. A biblioteca chama cada um destes métodos quando os objectos envolvidos são do tipo exacto indicado nos métodos `onCollision`. Por este motivo, o código que define a reacção à colisão entre os elementos do jogo é aquele onde terão de ser feitas maiores alterações quando comparada a utilização das classes `GameEnviromentPG` e `GameEnviromentPOO`. É também através da resolução deste problema que o aluno é exposto às limitações do despacho dinâmico simples existente na linguagem Java, e que é semelhante a outras linguagens de grande utilização (e.g. $C++$, $C\#$).

A interface `IGameObject`, tal como apresentada na Figura 3.1, define a assinatura do método `onCollision`, o qual tem um segundo parâmetro do tipo `IGameObject`. Mesmo que existam várias versões do método `onCollision`, não será possível chamar a versão correcta já que em Java o despacho dinâmico é feito apenas com base no tipo concreto do objecto sobre o qual o método é chamado. Os vários métodos são por isso sobrecargas do nome `onCollision` e não a redefinição do método para cada par de tipos.

Uma solução para este problema é testar todas as situações de colisão usando o operador `instanceof`. Apesar de ser possível chegar a uma solução funcionalmente correcta por esta via, o uso do operador `instanceof` é mais propício a erros e resulta em código que é mais difícil de perceber [12]. A Figura 3.2 apresenta o diagrama UML de uma solução que tem por base a ideia de duplo despacho. Apesar de evitar testes exaustivos a todas as situações, tem a potencial de quebrar a modularidade porque é preciso alterar a interface de todas as classes quando novos elementos são acrescentados ao jogo.

Com esta solução, a forma de definir as classes que representam os elementos do jogo (e.g. `HitBrick`, `Ball`) assemelha-se ao processo que o aluno experimentou durante a utilização da classe `GameEnviromentPG`. Todos as classes que derivam de `AbstractGameObject` ou `AbstractMovingObject` implementam `ICollision`. Quando dois elementos colidem a biblioteca chama sobre ambos o método `onCollision`. Através da técnica de duplo despacho, a operação de colisão continua a ser fortemente tipificada. Nas classes concretas é apenas preciso especificar a implementação dos métodos que recebem um tipo de objecto sobre o qual é relevante tratar a colisão.

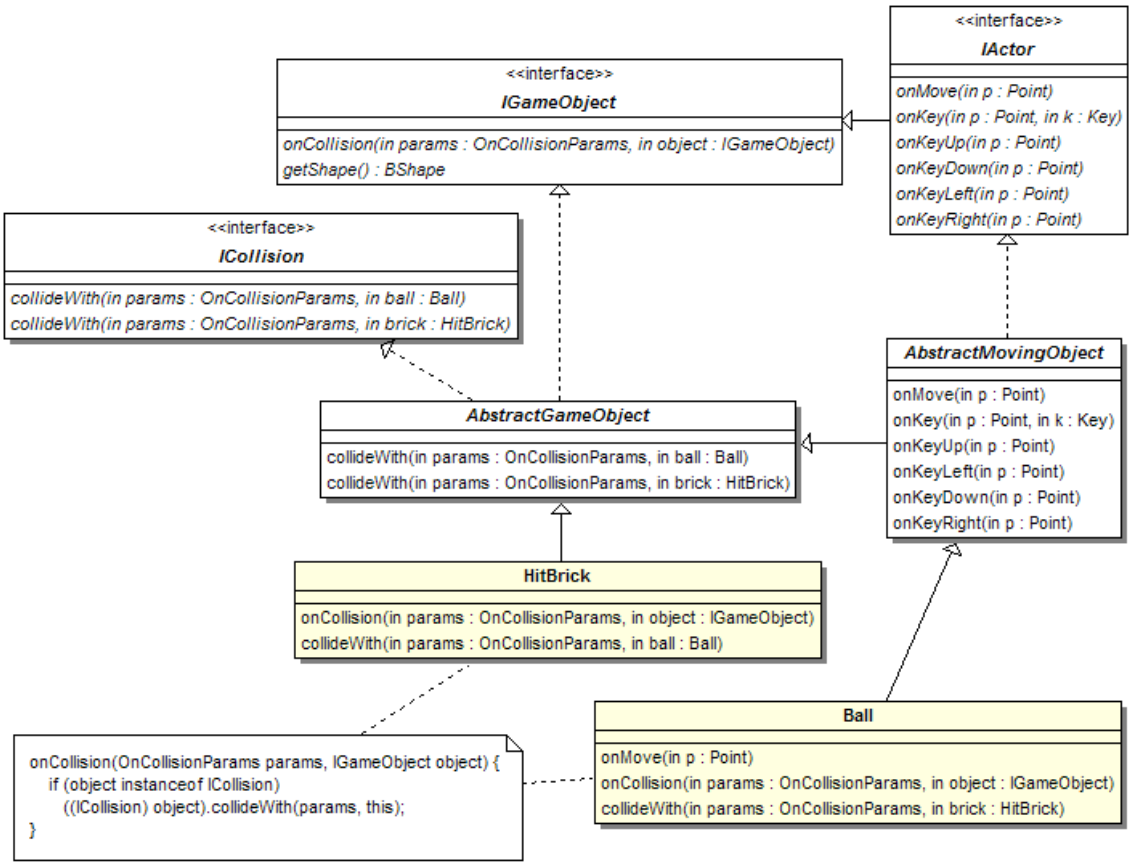

Figura 3.2: Reacção a colisões com duplo despacho

3.4 Ordem de apresentação dos temas herança de implementação e polimorfismo

O tema herança de implementação é habitualmente apresentado antes do tema polimorfismo, já que o primeiro serve de base ao segundo (no caso do polimorfismo baseado em sub-tipos). Esta ordem pode contudo ser invertida, sendo esta alteração facilitada pela utilização da biblioteca *JShapes*. Desde a definição dos primeiros elementos do jogo, o aluno tem indirectamente lidado com os ideias mais importantes: objectos e polimorfismo. Tudo no jogo são objectos e eles comportam-se de maneira diferente para operações com o mesmo nome (e.g. `onMove`). As bases estão por isso lançadas para se explicar o que suporta este comportamento polimórfico - a herança de interface e o despacho dinâmico.

Esta abordagem sugere que o aluno seja em primeiro lugar estimulado a perceber como é possível implementar algo semelhante ao micro motor de jogo que utilizou, ou seja, que mecanismos na linguagem dão suporte à gestão heterogénea de tipos. Neste caso seria utilizada desde o início de POO a classe `GameEnviromentPOO`. Seguir este caminho tem a vantagem de se poder falar de técnicas que tiram partido destes mecanismos no contexto da reutilização de código. O exemplo da definição de diferentes tipos de tijolos (Listagem 3.1) poderia então ser apresentado como na Listagem 3.2, tirando partido da técnica *template method* [4].

```
class HitBrick extends Brick {
   private int hitCounter;
   // template method
   public void onCollision(OnCollisionParams params, Ball b) {
      if (hitCounter-- == 0) {
         GameEnviromentPG.removeOjectFromLevel(this);
         doAfterBrickRemove();
      }
      else
         getShape().setFillColor(colors[hitCounter]);
   }
   public int getHitCounter() { return hitCounter; }
   // hook method
   public void doAfterBrickRemove() { }
   //...
}
class BonusBrick extends HitBrick {
   public void doAfterBrickRemove() {
      GameEnviromentPG.addObjectToLevel(new Bonus(...));
   }
}
```

Listagem 3.2: *Template method* na definição de diferentes tipos de tijolos

A abordagem clássica tem a vantagem de separar mais claramente os aspectos a tratar,

apesar de dar ênfase a uma técnica que de algum modo contradiz dois princípios do desenho orientado por objectos [4]: *i)* programar para uma interface e não para uma implementação *ii)* favorecer a composição de objectos em detrimento da herança de implementação.

Capítulo 4

Aspectos de desenho e implementação da biblioteca *JShapes*

Este capítulo apresenta sucintamente as classes que fazem parte da biblioteca *JShapes*, realçando alguns aspectos de implementação. É discutida a abordagem seguida para a representação dos objectos gráficos. É também explicado o uso da biblioteca de reflexão para verificar a compatibilidade dos objectos com a meta-descrição das categorias pré-definidas e a chamada dinâmica dos respectivos métodos.

4.1 Visão global da biblioteca

A Figura 4.1 resume o conjunto de classes que suportam a utilização dos objectos gráficos e do micro motor de jogo apresentados nas secções anteriores.

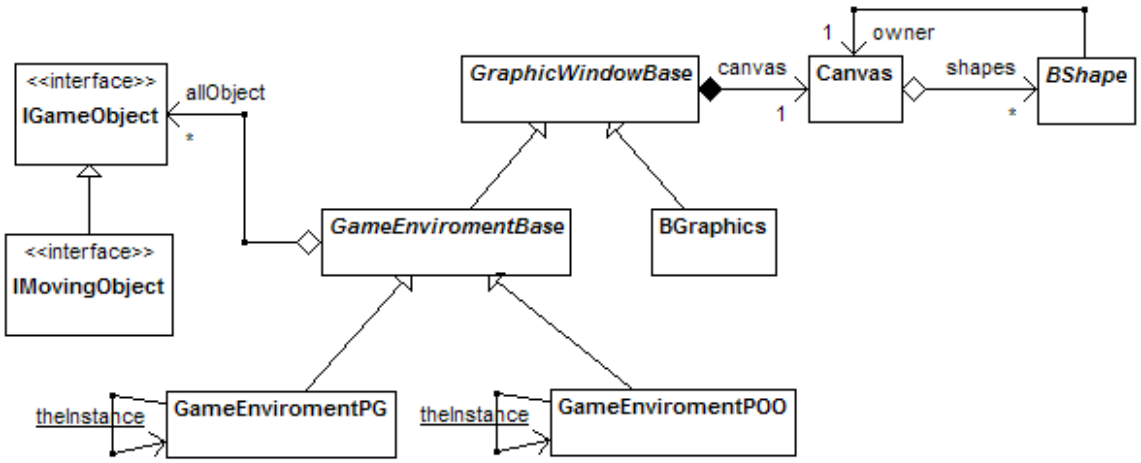

Figura 4.1: Resumo do diagrama de classes UML da biblioteca JShapes

As classes `GraphicWindowBase` e `Canvas` fazem, em conjunto, a ligação entre a biblioteca *Swing* e o modelo de objectos gráficos.

4.2 Componente `Canvas`

A classe `Canvas` estende de `JComponent` e é por isso um componente *Swing*. Ela tem a responsabilidade de gerir objectos `BShape`, nomeadamente, adicioná-los, remove-los e pintá-los na área gráfica. A classe `BGraphics` fornece apenas uma fachada para adicionar objectos gráficos.

Cada `BShape` tem uma referência para o `Canvas` a que pertence. Isto é necessário porque os métodos da classe `BGraphics` retornam o objecto gráfico criado. Assim, qualquer alteração sobre o estado do objecto, a qual tenha implicações na sua representação visual, resulta numa actualização do `Canvas` que lhe está associado.

Parte dos tipos que concretizam `BShape` são suportados nas formas geométricas disponíveis na biblioteca *Java2D* [13]. A Figura 4.2 mostra essa dependência.

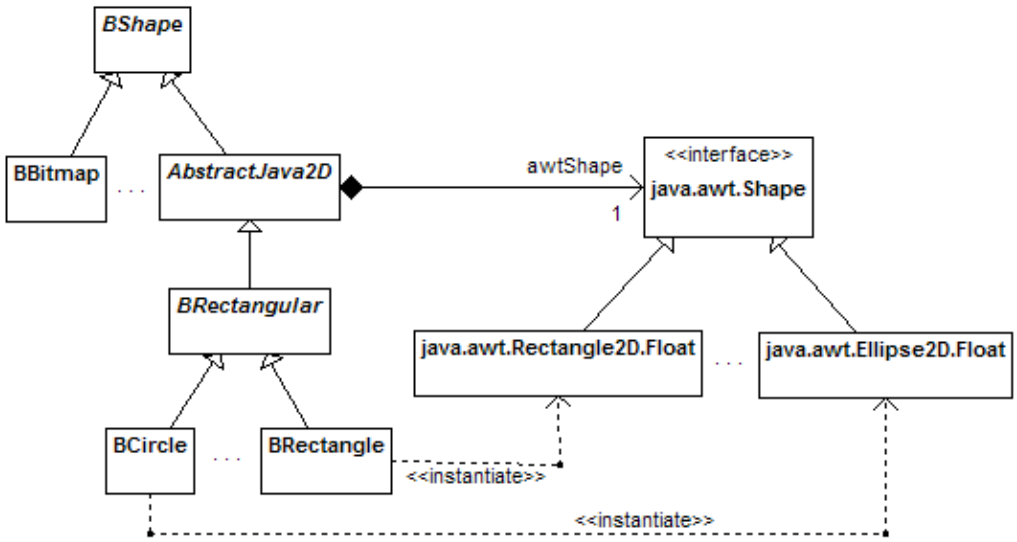

Figura 4.2: Hierarquia `BShape`

4.3 Micro motor de jogo

O micro motor de jogo é utilizável através das classes `GameEnviromentPG` e `GameEnviromentPOO`, tal como descrito nos capítulos anteriores. Estas classes são fachadas para os serviços prestados pela classe `GameEnviromentBase`, a qual gere os elementos do jogo. Na actual implementação, os serviços prestados pelo micro motor de jogo são: *i)* chamada periódica ao método `onMove`, *ii)* detecção de colisões, *iii)* encaminhamento de teclas e *iv)* identificação dos objectos por nome.

4.3.1 Gestão de objectos

Os objectos que representam os elementos do jogo são guardados numa árvore binária de pesquisa (tal como implementada pela classe `java.util.TreeMap<K,V>`). A árvore mantém os elementos ordenados pela ordem natural da *string* associada a cada objecto. A *string* corresponde ao nome do elemento do jogo. Este nome é definido por quem adicionou o objecto ou, em caso de omissão, pelo resultado da chamada ao método `hashCode` do objecto. Esta implementação tem por base as seguintes premissas:

- Os nomes atribuídos explicitamente aos objectos são diferentes para qualquer objecto;
- As classes das quais estes objectos são instância não redefinem o método `hashCode` de `Object`;
- O método `hashCode` de `Object` retorna o endereço do objecto, sendo por isso um valor diferente para qualquer elemento do jogo. Este é o comportamento que a documentação da plataforma refere como "prática comum" [14].

A associação de um nome ao objecto, e a possibilidade de mais tarde obter referência para ele, levanta problemas à utilização da biblioteca em PG. O aluno não conhece as regras de herança do sistema de tipos, não sendo por isso natural obrigá-lo a fazer um *downcast* para guardar a referência numa variável de um tipo concreto do jogo (e.g. `Wall` em oposição a `Object`). Para evitar este *downcast* explícito, o método `getObjectByName` da classe `GameEnviromentPG` é genérico no tipo de retorno. A sua assinatura é apresentada na Listagem 4.1.

```
class GameEnviromentPG {
    //...
    public static <E> E getObjectByName(String);
}
```

Listagem 4.1: Assinatura do método `getObjectByName`

Na chamada ao método `getObjectByName` o compilador infere o tipo do retorno através do tipo da variável onde este é guardado e produz as instruções para realizar o *downcast* implicitamente. Existe contudo o potencial de ser lançada excepção em tempo de execução, indicando que o tipo do objecto retornado não pode ser convertido no tipo da variável. Não é possível evitar que seja lançada esta excepção porque a informação de tipo perde-se aquando da compilação do método `getObjectByName` para *byte codes*, não sendo possível saber, em tempo de execução, se o tipo do objecto obtido através do nome é compatível com o tipo inferido pelo compilador.

4.3.2 Utilização da biblioteca de reflexão

A classe `GameEnviromentPG` usa a biblioteca de reflexão da linguagem Java para realizar invocações dinâmicas sobre os objectos que representam os elementos do jogo. Os critérios de desenho subjacentes à utilização dos mecanismos de reflexão foram os seguintes:

- A base de código do motor de jogo não deve estar comprometida com aspectos relacionados com o uso dos mecanismos de reflexão;
- O processo de verificação de compatibilidade com uma interface não deve estar limitada às interfaces existentes na actual implementação.

Estes dois critérios foram cumpridos através da utilização do padrão de desenho `Adapter` [4] e de anotações [2].

Padrão *Adapter*

A classe `GameEnviromentBase` é composta por um contentor de objectos que implementam a interface `IGameObject`. Quer se trate de objectos que não implementam qualquer interface Java (definidos em PG e adicionados através da classe `GameEnviromentPG`) ou aqueles que implementam a interface `IGameObject` (definidos em POO e adicionados através da classe `GameEnviromentPOO`), a classe `GameEnviroment` guarda-os no mesmo contentor e realiza sobre eles operações comuns. Para cumprir as regras do sistema de tipos da linguagem foi usado o padrão *Adapter* o qual resolve a incompatibilidade que existe com as classes definidas em PG.

O padrão de desenho *Adapter* converte a interface de uma classe noutra interface, de maneira a que instâncias da classe original possam ser utilizadas no código cliente sem o modificar. A Figura 4.3 mostra a utilização deste padrão no contexto da biblioteca *JShapes*.

As classes `GameObjectAdapter` e `MovingObjectAdapter` desempenham o papel de adaptadoras. Elas usam a biblioteca de reflexão para fazer as chamadas aos objectos que realmente têm a lógica do jogo, os quais desempenham o papel de adaptados. Estas operações são realizadas com base nos serviços prestados pela classe `GenericObject`.

Anotações

Cada vez que um objecto é adicionado ao jogo através da classe `GameEnviromentPG`, é verificada a sua compatibilidade com uma das categorias de objectos que a biblioteca *JShapes* admite. Esta operação evita que as estruturas de dados internas da biblioteca sejam preenchidas com referências para objectos que não sabem reagir aos eventos do jogo.

Para que no futuro a biblioteca possa mudar o critério de compatibilidade, sem que isso implique alterações substanciais na implementação, os métodos das interfaces *Java* que caracterizam os elementos do jogo estão marcados com dois tipos de anotações:

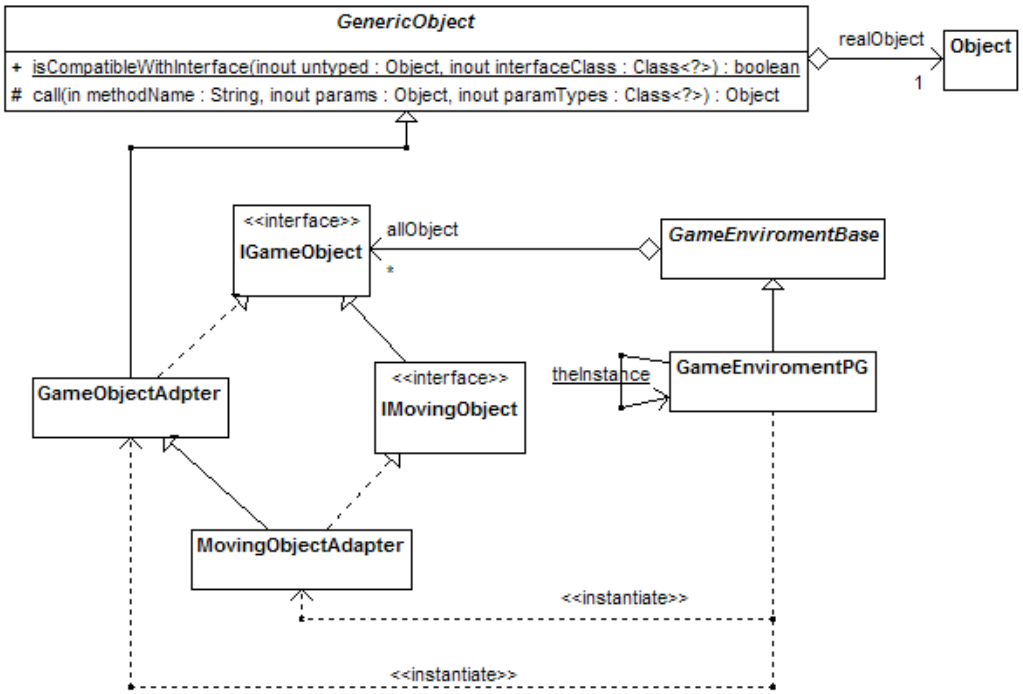

Figura 4.3: Classes adaptadores

- `@MustImplement`. Os métodos onde esta anotação está presente têm de existir na classe que define o elemento de jogo;
- `@Optional`. Os métodos onde esta anotação está presente podem ou não fazer parte da classe que define o elemento de jogo. Caso não exista qualquer método anotado com `@MustImplement`, a compatibilidade com determinada interface é equivalente a um "ou lógico", ou seja, tem de ser definido pelo menos um dos métodos opcionais.

Os métodos `onCollision` têm a particularidade de apenas ser preciso uma compatibilidade parcial com a sua assinatura, ou seja, tem de existir obrigatoriamente um segundo parâmetro nos métodos das classes que definem o elemento do jogo, mas o seu tipo não conta para efeitos de compatibilidade. Este parâmetro (e outros que no futuro venham a existir com o mesmo comportamento) é anotado com a anotação `@Any`.

Todas estas anotações são persistidas na *metadata* associada às interfaces `IGameObject` e `IMovingObject` (i.e. tem o nível de retenção igual a `RUNTIME`).

Capítulo 5

Conclusões

O paradigma da programação orientado por objectos é hoje um saber fundamental para os profissionais na área das tecnologias da informação. Por esse motivo, este paradigma deve ser apresentado nas primeiras disciplinas onde são explicados os fundamentos da programação [1, 15].

A abordagem seguida nas três licenciaturas do Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores (DEETC) do Instituto Superior de Engenharia de Lisboa (ISEL) é baseada no modelo *imperative-first* [1]. Apesar de não ser adoptada a estratégia *objects-first* [1], o programa da unidade curricular Programação incluí a introdução à programação orientada por objectos.

Sendo *Java* a linguagem usada nas unidades curriculares de introdução à programação, não é possível usar directamente os recursos da linguagem e respectiva biblioteca para oferecer ao aluno uma experiência interactiva (mesmo que simples) como é sugerido pelo trabalho do grupo de trabalho do IEEE [1]. A biblioteca *Swing*, a qual faz parte do conjunto de classes que acompanham a linguagem Java, pressupõe que o programador domina a programação orientada por objectos e as técnicas subjacente (e.g. padrões de desenho como é o caso do *Observer*), para além de aspectos transversais como é o problema da concorrência.

Este documento apresenta a biblioteca *JShapes*, a qual tem como objecto servir de material pedagógico no ensino da programação orientada por objectos. A biblioteca proporciona uma experiência de utilização onde desde os primeiros programas o aluno lida com a abstracção de objecto de forma simples, no contexto de aplicações com interface gráfica em aplicações de *desktop*. A abordagem sugerida consiste em três fases. Em cada uma delas o aluno é exposto a uma parte do problema. Primeiro é utilizador de objectos gráficos os quais pode desenhar no ecrã e manipulá-los directamente. Depois define as classes que representam os objectos a usar num micro motor de jogo, analisando primeiro o problema do encapsulamento e depois a herança de implementação para efeitos de reutilização de código. Por último, utiliza a biblioteca tendo de obedecer às regras do sistema de tipos da

linguagem, dando-lhe oportunidade de explorar soluções que usam os vários aspectos da programação orientada por objectos.

A abordagem proposta precisa contudo de validação prática. Essa validação consiste em expor os alunos à utilização da biblioteca e aferir a melhoria dos resultados no final do período de avaliação. Do ponto de vista quantitativo, essa melhoria consiste num aumento na taxa de aprovação às disciplinas de Programação e Programação Orientada por Objectos, usando o modelo de avaliação actualmente em prática. A melhoria qualitativa poderá ser avaliada de duas formas: *i)* na qualidade da discussão que o aluno realiza depois de obter aprovação no exame; *ii)* na complexidade das soluções com as quais consegue lidar.

Bibliografia

[1] "Computing curricula 2001 computer science," IEEE Computer Society, Dezembro 2001.

[2] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*. Addison-Wesley, 2005.

[3] W. Savitch, *Java: An Introduction to Problem Solving and Programming*. Publisher: Prentice Hall, 2005.

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

[5] J. Bergin, M. Clancy, D. Slater, M. Goldweber, and D. B. Levine, "Day one of the objects-first first course: what to do," in *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, 2007, pp. 264–265.

[6] J. Bergin, E. Wallingford, M. Caspersen, M. Goldweber, and M. Kolling, "Teaching polymorphism early," pp. 342–343, 2005.

[7] S. Cooper, W. Dann, and R. Pausch, "Teaching objects-first in introductory computer science," pp. 191–195, 2003. [Online]. Available: http://dx.doi.org/10.1145/611892.611966

[8] D. J. Barnes and M. Kölling, *Objects First With Java: A Practical Introduction Using BlueJ, 4ª Edição*. Prentice Hall, 2009.

[9] J. Bergin, "Fourteen pedagogical patterns," 2000. [Online]. Available: http://csis.pace.edu/ bergin/PedPat1.3.html

[10] J. Richter, *CLR via CSharp*. Microsoft Press, 2006.

[11] Arkanoid wiki. [Online]. Available: http://en.wikipedia.org/wiki/Arkanoid, Visitado em 9 de Setembro de 2009

[12] R. Muschevici, A. Potanin, E. Tempero, and J. Noble, "Multiple dispatch in practice," *SIGPLAN Not.*, vol. 43, no. 10, pp. 563–582, 2008.

[13] K. Walrath and M. Campione, *The JFC Swing Tutorial: A Guide to Constructing GUIs*. Addison-Wesley, 1999.

[14] S. Microsystems. [Online]. Available: http://java.sun.com/javase/6/docs/, visitado em 9 de Setembro de 2009

[15] "Software engineering 2004 - curriculum guidelines for undergraduate degree programs in software engineering," IEEE Computer Society, Agosto 2004.