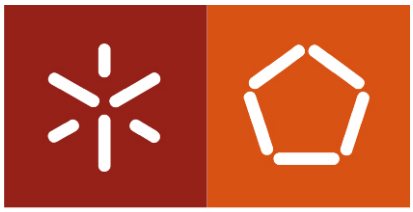

Universidade do Minho
Departamento de Informática

Jaime Emanuel V. S. M. Neto

Geração Automática de Código para Padrões de Concepção

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob a orientação do
Professor Doutor António Nestor Ribeiro

Outubro de 2011

Geração Automática de Código para Padrões de Concepção

Agradecimentos

Agradeço à Universidade do Minho, à Escola de Engenharia e ao Departamento de Informática pela importância que tiveram na minha formação, em especial os docentes com quem tive o privilégio de aprender. Queria ainda destacar toda a disponibilidade das instalações da Universidade do Minho, que permitiram que pudesse lá estudar e trabalhar sempre que necessário, ao longo da minha formação.

Queria também manifestar aqui um agradecimento ao orientador da presente dissertação, que se mostrou sempre interessado e empenhado em ajudar a encaminhar a dissertação para a direcção certa, nos momentos mais complicados, e sem o qual teria sido impossível levar a cabo todo este trabalho.

Por fim, não posso deixar de agradecer a todos os meus colegas, familiares e amigos, cujo apoio foi também determinante para conseguir realizar este trabalho.

Resumo

O recurso a ferramentas de geração automática de código permite economizar tempo quando se desenvolvem soluções de software, factor importante em questões de produtividade.

Existe um conjunto de padrões de concepção [Gamma et al., 1995] que representam soluções genéricas para problemas relativos ao desenvolvimento de aplicações de software, numa perspectiva orientada aos objectos. Para cada um deles pode ser vista a sua estrutura de classes, métodos e relacionamentos, bem como as situações mais adequadas para a sua utilização. Bastará consultar o catálogo de padrões de concepção [Gamma et al., 1995] e utilizar aquele que mais se adequar à resolução de determinado problema que surja no desenvolvimento de um novo programa.

A existência de uma aplicação de software capaz de fazer a geração automática do código associado aos padrões de concepção, agiliza o desenvolvimento de novas aplicações, porque fornece de imediato o respectivo código.

O que se propõe com o desenvolvimento desta dissertação é uma solução de software, capaz de efectuar a geração automática de código para os padrões de concepção catalogados em [Gamma et al., 1995]. Juntamente com o programa desenvolvido, é também apresentado um levantamento do estado da arte sobre os padrões de concepção, considerando também situações actuais da sua aplicabilidade. Em seguida, é descrita a especificação da aplicação elaborada, bem como o seu processo de desenvolvimento, acompanhado de um exemplo de utilização. Por fim encontram-se dois casos de estudo, servindo para provar que o programa elaborado pode ser utilizado em contextos reais.

Área de Aplicação: Desenvolvimento de Sistemas de Software.
Palavras-chave: Geração de Código; Padrões de Concepção.

Abstract

Automatic code generation tools are very important when developing software, since they generate code very quickly, the software can be released earlier, which is a key factor nowadays.

There is a set of design patterns [Gamma et al., 1995] that represent generic solutions to software development problems, regarding an object-oriented perspective. For each design pattern there is a class diagram with some methods and relationships between classes, and some examples of use. To solve a problem that arises when developing a new software program, it is enough searching for the appropriate design pattern [Gamma et al., 1995].

So, a software application that automatically generates code for design patterns eases developing new software, once the patterns' code is immediately provided.

In this master dissertation it is proposed a software solution to automatically generate code for design patterns [Gamma et al., 1995]. It is also presented the state of the art about design patterns, as well as some recent examples using them. The design of the developed program is also approached, and its implementation process too. Finally, there are two case studies proving the developed program can be used in real contexts.

Application Area: Software Systems Development.
Keywords: Code Generation; Design Patterns.

Conteúdo

Lista de Siglas e Acrónimos

DPIO	*Design Pattern Intent Ontology*
EML	*Entity Meta-specification Language*
IDE	*Integrated Development Environment*
OSGi	*Open Service Gateway Initiative*
PatGer	*Gerador Automático de Código para Padrões de Concepção*
TEO	*Sistema de Gestão de Serviço de Televisão (pay per view)*
UML	*Unified Modeling Language*

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Motivação

Os padrões de concepção permitem resolver problemas recorrentes relacionados com o desenvolvimento de software, numa perspectiva orientada aos objectos. Cada um deles identifica um determinado problema, e apresenta a solução a adoptar, além do contexto onde poderá funcionar devidamente [Johnson, 1997]. Essa solução consiste num conjunto de classes ou objectos, além dos seus relacionamentos, e tenta ser abstracta, uma vez que a preocupação principal reside na estrutura da solução genérica a apresentar, e não numa solução particular para um problema concreto [Gamma et al., 1993]. No entanto, como referido em [Bosch, 1998], a partir da solução genérica é possível especializá-la e adaptá-la para um problema concreto.

Os padrões de concepção oferecem um aumento de vocabulário para quem desenvolve sistemas de software, bem como reduzem a complexidade associada a um sistema de software [Gamma et al., 1993], pois com eles já é possível falar das soluções (abstractas) para alguns problemas, usando apenas o nome do padrão adequado. Como é dito em [Welicki et al., 2006b], os padrões de concepção são "peças" de conhecimento da engenharia de software, e, ao serem divulgados e partilhados pela comunidade, vão permitir elevar o nível de abstracção e conhecimento no desenvolvimento de aplicações de software. Pode-se ainda dizer que os padrões de concepção são capazes de funcionar como pequenos componentes, a partir dos quais outros componentes mais complexos poderão ser obtidos, durante o desenvolvimento de uma aplicação de software.

Com as soluções apresentadas pelos padrões de concepção, é ainda possível produzir software a partir de software já existente, ou seja, reutilizando código já desenvolvido. Para que esta ideia possa ser posta em prática, tal como sugerido em [Krueger, 1992], é necessário que o código escrito seja suficientemente genérico e abstracto, de modo a poder ser reutilizado em diversas ocasiões.

No que diz respeito à identificação dos vários padrões de concepção, foram propostas algumas sugestões, por diferentes autores [Coad et al., 1995, Gamma et al., 1993, Zimmer, 1994]. A abordagem considerada ao longo desta dissertação será a proposta por [Gamma et al., 1995], que foi adaptada e redefinida a partir da sua versão de 1993.

Além desta abordagem, existem outras, nomeadamente a seguida em [Buschmann et al., 1996], onde são identificados padrões orientados à arquitectura de software, sendo que também há a identificação de *antipatterns*, presente em [Brown et al., 1998], onde são mencionadas situações a evitar seguir durante o desenvolvimento de um projecto de software.

Como foi já referido anteriormente, os padrões de concepção sugerem soluções, ou seja, um conjunto de entidades relacionadas entre si, para resolver alguns problemas associados ao desenvolvimento de software. Assim, surge a ideia de passar do papel para código as soluções associadas aos padrões de concepção.

Pretende-se com isto dar mais importância aos padrões de concepção, deixando apenas de ser relevantes a nível teórico, e passando a ter uma utilização mais prática. Isto é, o objectivo passa a ser poder incluir nas aplicações de software a desenvolver, "blocos" de software mais pequenos, com os quais se consiga programar uma aplicação maior. Esses "blocos" serão os padrões de concepção, com o código a eles associado gerado automaticamente por outra aplicação informática.

Para que esse tipo de aplicações possa funcionar correctamente, o utilizador terá de ter uma noção mínima de qual o padrão de concepção a que deve recorrer numa dada situação, e terá que fornecer alguns dados importantes (por exemplo, o nome das classes) para que o programa lhe possa gerar o código associado ao padrão em causa. É também importante que a referida aplicação seja *cross-platform*, para se maximizar o número de situações onde possa ser útil para quem desenvolve aplicações de software.

Sobre aplicações capazes de fazer a geração automática de código associado aos padrões de concepção, para linguagens de programação orientadas aos objectos, importa dizer que já foram desenvolvidas algumas, como se pode verificar em [Budinsky et al., 1996, Reiss, 2000, Welicki et al., 2006a]. Além destas, é também importante destacar uma aplicação proposta em [Kampffmeyer and Zschaler, 2007], que procura sugerir os padrões de concepção mais adequados à resolução de problemas de design, indicados pelo utilizador.

1.2 Objectivos

Este trabalho tem como objectivo principal o desenvolvimento de uma aplicação de software, responsável por efectuar a geração automática de código para os padrões de concepção identificados em [Gamma et al., 1995]. Para

que seja possível satisfazer este objectivo global, há outros objectivos que devem ser cumpridos.

Em primeiro lugar, há que fazer um levantamento do estado da arte associado ao tema escolhido:

- identificar os vários padrões de concepção, de acordo com o proposto em [Gamma et al., 1995], bem como os critérios usados para a sua classificação e agrupamento, analisando também situações actuais da aplicabilidade de alguns desses padrões, mostrando que, apesar de já terem alguns anos, continuam a poder ser utilizados no desenvolvimento de aplicações informáticas recentes;

- referir as abordagens seguidas em [Buschmann et al., 1996] e também em [Brown et al., 1998], para a identificação de padrões orientados à arquitectura de software, e para os *antipatterns*, respectivamente, fazendo uma descrição sucinta de cada um desses padrões;

- identificar aplicações, já desenvolvidas, que façam também a geração automática de código para os padrões de concepção.

Em seguida, deverá ser feita uma análise mais cuidada sobre o problema relacionado com o tema da dissertação, ou seja:

- descrever alguns dos vários padrões de concepção de [Gamma et al., 1995], mostrando a sua estrutura e um exemplo concreto da sua utilização, com recurso a diagramas;

- desenvolver o Modelo do Domínio da aplicação a elaborar, e fazer a análise de requisitos, ou seja, as funcionalidades que deverá disponibilizar aos seus utilizadores.

Em relação à forma como se planeia desenvolver uma solução para o problema, ou seja, programar uma aplicação de software para gerar automaticamente o código para os padrões de concepção, importa referir os seguintes aspectos:

- especificar todo o programa a desenvolver, isto é, desenhar esboços para a sua interface, e especificar a lógica de negócio a implementar, ou seja, os vários packages e classes que compõem o diagrama de classes da aplicação;

- implementar a solução proposta, ou seja, desenvolver a referida aplicação de software, capaz de gerar automaticamente o código para os padrões de concepção, obtendo-se como resultado final um conjunto de ficheiros, respeitando a estrutura do padrão escolhido, mas contendo o código de acordo com as necessidades concretas do utilizador;

- descrever o processo de implementação da aplicação de software referida, indicando a tecnologia e linguagem de programação adoptadas, juntamente com um breve exemplo de utilização.

Para terminar, interessa ainda provar que o programa desenvolvido pode ser aplicado em contextos reais, elaborando casos de estudo:

- recorrendo a uma aplicação de software de pequena/média dimensão, aplicando o programa entretanto desenvolvido para modificar a sua estrutura de classes, e/ou acrescentar novas funcionalidades;

- fazendo o mesmo que descrito no tópico anterior, mas desta vez utilizando uma aplicação de software de maior dimensão, provando que o programa desenvolvido pode ser utilizado em mais do que uma situação.

1.3 Organização do documento

No capítulo 2 vai ser analisado o estado da arte respeitante ao tema da dissertação, tendo em consideração a identificação dos vários padrões de concepção [Gamma et al., 1995], sendo descrito o tipo de problemas para os quais apresentam solução, e dando também importância a situações actuais onde estes padrões são utilizados, tendo por objectivo mostrar que são aplicáveis, nos dias de hoje, no desenvolvimento de aplicações de software.

Em seguida, serão referidas outras abordagens diferentes na identificação de padrões de software, das quais resultaram outros padrões de software, e ainda padrões que reflectem comportamentos tidos em projectos e em equipas de desenvolvimento de software que devem ser evitados.

Depois, serão analisados alguns programas capazes de fazer a geração de código para os padrões de concepção. Além disso, será também incluída uma ferramenta que sugere padrões de concepção de acordo com o problema de design do utilizador.

No capítulo 3 será apresentado o problema relacionado com a dissertação, onde serão abordados, com detalhe, alguns dos vários padrões de concepção, mostrando a sua estrutura, além de um exemplo concreto para a sua utilização, juntamente com o código associado.

Ainda neste capítulo vai ser feita uma breve descrição da aplicação a desenvolver, com a ajuda do seu Modelo do Domínio, bem como a identificação dos requisitos que deverá respeitar.

No capítulo 4 vai ser descrita a aplicação de software desenvolvida, onde será tida em consideração toda a especificação efectuada, desde os esboços elaborados para a interface da aplicação, até aos packages e classes que implementam a sua lógica de negócio. Serão apresentados e analisados vários exemplos para os esboços e diagramas especificados.

Serão também reveladas algumas decisões, tomadas antes da etapa de codificação, juntamente com informação sobre como adicionar novas funcionalidades ao programa implementado.

O capítulo 5 está relacionado com exemplos reais de utilização da aplicação de software desenvolvida, ou seja, casos de estudo que demonstram que é possível recorrer a este programa para modificar a estrutura de classes de uma aplicação de software já existente, ou mesmo para lhe acrescentar novas funcionalidades.

O capítulo 6 diz respeito às conclusões, onde irá ser apresentada uma breve síntese do que foi feito durante a dissertação, além de uma análise crítica sobre os aspectos mais importantes relativos ao trabalho desenvolvido. Também neste capítulo vai estar presente uma secção sobre algum trabalho futuro que poderá vir a ser efectuado, de modo a melhorar a aplicação elaborada.

Para terminar, resta dizer que no final haverá um conjunto de anexos com informação complementar, além da bibliografia consultada durante a realização da presente dissertação.

Capítulo 2

Estado da Arte e Trabalho Relacionado

O tema escolhido para a presente dissertação está relacionado com os Padrões de Concepção [Gamma et al., 1995], e a capacidade para desenvolver uma aplicação informática capaz de fazer a geração automática do código associado.

Desta forma, em primeiro lugar, há que apresentar os conceitos relacionados com este tema, ou seja, apresentar o respectivo estudo sobre o estado da arte do tema em questão.

Assim, durante este capítulo, vão ser introduzidos os padrões de concepção, de acordo com a classificação proposta em [Gamma et al., 1995], para os quais a aplicação a programar deverá gerar o código associado. Em seguida, será dada importância à aplicabilidade actual de alguns dos vários padrões de concepção. Posteriormente, irão ser analisados, de forma sucinta, os padrões orientados à arquitectura de software [Buschmann et al., 1996], bem como os *antipatterns* [Brown et al., 1998], mostrando outros padrões para além dos considerados para a aplicação a elaborar. No final deste capítulo serão revistas algumas ferramentas de geração de código para os padrões de concepção.

2.1 Padrões de Concepção

Ao longo dos anos foram sendo propostas várias formas de catalogar os padrões de concepção, e, como consequência disso, podem ser encontrados diferentes padrões conforme os autores e a forma como os identificaram e catalogaram.

De acordo com o descrito em [Coad et al., 1995], é utilizado um esquema de classificação que agrupa os padrões em quatro subclasses: Transacção, Acordo, Plano e Interacção, existindo em cada uma destas subclasses vários padrões de concepção. O problema desta abordagem é que grande parte dos

		Purpose		
		Creational	*Structural*	*Behavioral*
Scope	Class	*Factory Method*	*Adapter*	*Interpreter* *Template Method*
	Object	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Adapter* *Bridge* *Composite* *Decorator* *Facade* *Flyweight* *Proxy*	*Chain of Responsibility* *Command* *Iterator* *Mediator* *Memento* *Observer* *State* *Strategy* *Visitor*

Tabela 2.1: *Padrões de Concepção [Gamma et al., 1995]*

padrões são apenas instâncias da subclasse a que pertencem.

Existe também a abordagem tomada por [Zimmer, 1994], onde é dada importância às relações entre os padrões de concepção para os catalogar. Contudo, esta forma de identificar os padrões não é muito clara, pois nem sempre se conseguem distinguir dois padrões que estão relacionados.

A classificação proposta por [Gamma et al., 1993] tem em consideração dois critérios: jurisdição e caracterização. Relativamente à jurisdição, pode-se dizer que trata do domínio ao qual o padrão pode ser aplicado (Classe, Objecto ou Composto). Sobre a caracterização, importa mencionar que podem ser considerados três grupos: Criação, Estruturais e Comportamentais.

Posteriormente, em [Gamma et al., 1995], os autores redefiniram a classificação sugerida. Assim, a nova classificação utiliza dois critérios, idênticos aos anteriores, mas com nomes diferentes: propósito e âmbito. O critério propósito está relacionado com o que um padrão faz, e o âmbito considera o domínio de aplicação do padrão. Na Tabela 2.1 estão representados os padrões de concepção catalogados por [Gamma et al., 1995], e agrupados de acordo com os critérios descritos.

Uma vez mais, tal como no anterior critério de caracterização, relativamente ao seu propósito, os padrões podem ser de Criação (estão relacionados com o processo de criação de objectos), Estruturais (relativos à composição de classes ou objectos), ou Comportamentais (mostram como as classes e os objectos interagem e distribuem responsabilidades) [Gamma et al., 1995].

Também como no anterior critério de jurisdição, os padrões podem ser de Classe ou Objecto (desta vez não há a opção Composto). Os padrões de classe estão relacionados com o relacionamento estático entre as classes e as suas subclasses, recorrendo à herança. Os padrões de Objecto lidam com relações entre objectos, que podem ser modificadas em tempo de execução [Gamma et al., 1995].

2.1.1 Padrões de Criação

Os padrões de criação são responsáveis por criar objectos de forma adequada ao problema a resolver, abstraindo o processo de instanciação. Permitem que se construa um sistema independente da forma como os objectos que o constituem são criados, compostos e representados. Para os padrões de criação de classe, interessa referir que utilizam a herança para instanciar as classes desejadas. Por outro lado, os padrões de criação de objecto delegam o processo de instanciação para outro objecto [Gamma et al., 1995]. Por exemplo, quando estamos a utilizar a framework Hibernate [King et al., 2010], ao instanciarmos um objecto que se encontra mapeado na base de dados, estamos a recorrer a uma fábrica de objectos desse tipo, providenciada pelo Hibernate.

Uma vez que as aplicações a desenvolver começam a ser cada vez mais complexas, e a depender mais da composição do que da utilização de herança, a importância dos padrões de criação tem vindo a crescer [Gamma et al., 1995]. Desta forma, é possível definir comportamentos num dado objecto, e depois incorporar esse objecto num outro, passando este último a poder utilizar as suas funcionalidades, e também as do objecto que incorporou.

Em relação a este conjunto de padrões, interessa referir que encapsulam o conhecimento de que concretização de classes o sistema está a utilizar, e, além disso, escondem a forma como as instâncias são criadas e compostas. Apenas temos noção dos interfaces a utilizar, que são definidos por classes abstractas [Gamma et al., 1995].

2.1.2 Padrões Estruturais

Os padrões estruturais são responsáveis pela forma como as classes e os objectos são compostos, de modo a serem criadas estruturas maiores. Podemos encontrar dois tipos de padrões estruturais, de Classe ou de Objecto [Gamma et al., 1995].

Assim, em relação aos padrões estruturais de classe, há que mencionar que recorrem ao mecanismo de herança para comporem interfaces e implementações. Quanto aos padrões estruturais de objecto, pode-se dizer que descrevem formas de compor objectos para que seja possível a implementação de novas funcionalidades. Esta última opção permite uma maior flexibilidade, pois é possível alterar a composição em tempo de execução [Gamma et al., 1995].

2.1.3 Padrões Comportamentais

Os padrões comportamentais estão relacionados com os algoritmos e a atribuição de responsabilidades entre os objectos. Preocupam-se em definir a comunicação existente entre os objectos que compõem a estrutura do padrão, dando mais importância à forma como os objectos interagem entre si, e menos ao controlo de fluxo dos dados. Permitem assim caracterizar o controlo de fluxo, bastante complexo, em tempo de execução, e permitem a

quem desenvolve a aplicação concentrar-se apenas na forma como os objectos estão relacionados [Gamma et al., 1995].

Estes padrões de comportamento podem ser de objecto ou de classe, sendo que se forem de objecto, vão recorrer à composição para distribuírem aspectos de comportamento, e se forem de classe utilizarão o mecanismo de herança presente nas linguagens orientadas aos objectos [Gamma et al., 1995].

2.2 Aplicabilidade Actual dos Padrões de Concepção

Nas próximas subsecções vai ser dada uma ideia da aplicabilidade recente de alguns dos vários padrões de concepção. Pretende-se assim mostrar que os padrões de concepção, apesar de já identificados há vários anos atrás, são ainda aplicáveis no desenvolvimento de aplicações informáticas recentes.

2.2.1 Padrões de Criação

Nesta subsecção irá ser analisada a aplicabilidade actual de alguns padrões de criação, nomeadamente dos padrões *factory* (por exemplo, Abstract Factory), procurando mostrar que, embora à primeira vista não seja muito simples trabalhar com base neste grupo de padrões, actualmente existem aplicações que os utilizam para desenvolver as suas APIs [Ellis et al., 2007].

Uso de APIs baseadas em padrões *factory versus* construtores Em [Ellis et al., 2007] é mostrado um estudo comparativo relacionado com a facilidade de utilização de APIs baseadas em padrões *factory* ou construtores, onde foi solicitado a um grupo de doze pessoas que completassem um conjunto de tarefas de programação variadas, num tempo limitado, em Java e utilizando o Eclipse.

Para cada tarefa, cada pessoa seria aleatoriamente seleccionada para utilizar padrões *factory* ou construtores. Como resultado global foi concluído que era mais simples a utilização de construtores em relação aos padrões *factory*.

Porém, relativamente a este estudo, importa dizer que os resultados não devem ser generalizados, pois embora o universo da amostra utilizado tenha sido relativamente variado (pessoas entre os 18 e os 35 anos, com pelo menos um ano de experiência a programar em Java [Ellis et al., 2007]), foram apenas doze os intervenientes, o que pode ser considerado um número reduzido. Além de tudo isto, certamente que uma parte dos participantes (provavelmente os mais jovens, ou aqueles com menos experiência de programação) nunca terá utilizado

estes padrões no desenvolvimento dos seus programas, por isso é natural que, num primeiro impacto, fossem sentidas dificuldades na sua compreensão e modo de utilização.

Ainda sobre este estudo, em [Ellis et al., 2007] os próprios autores referem algumas aplicações bastante utilizadas (Microsoft .NET e Java 1.5 SE) que fornecem APIs baseadas em padrões de criação. Então, pode depreender-se que o recurso aos padrões de criação é uma opção válida quando se está a desenvolver software, embora exija um pouco mais dos seus futuros utilizadores, que inicialmente poderão ter mais dificuldades na interacção com a API fornecida, que terá provavelmente uma curva de aprendizagem maior.

Framework de persistência de dados Hibernate O Hibernate permite fazer o mapeamento objecto-relacional para as classes que queremos persistir, de acordo com o modelo orientado a objectos, guardando-as numa base de dados relacional [King et al., 2010].

Além disto, é capaz de gerir as chamadas SQL, libertando o programador da tarefa de conversão dos dados resultantes, possibilitando ainda que o programa seja portável para qualquer tipo de base de dados que utilize SQL.

De acordo com o descrito em [King et al., 2010], o programa que utilize o Hibernate tem de obter uma instância de SessionFactory para poder depois comunicar com a base de dados, ou seja, a API do Hibernate também utiliza padrões de criação. Há ainda outras situações onde se utiliza uma *factory* de um objecto, em vez do construtor, como por exemplo quando pretendemos carregá-lo a partir da base de dados.

2.2.2 Padrões Estruturais

Nesta subsecção será analisada a aplicabilidade de alguns padrões estruturais, nomeadamente do Adapter (versão que utiliza a composição de objectos), do Bridge, do Decorator, do Facade e do Flyweight.

Adapter Este padrão estrutural foi objecto de estudo em [Ferenc et al., 2005], juntamente com o padrão comportamental Strategy. Os autores desenvolveram uma aplicação para identificar estes dois padrões na arquitectura de classes do StarOffice (predecessor do OpenOffice), baseada em árvores de decisão e em redes neuronais.

Após os resultados revelados pela aplicação, foi necessário inspeccionar manualmente cada um dos "candidatos", de maneira a eliminar falsos positivos. Sobre o padrão Adapter, foram encontrados alguns exemplos, embora algo simples, ou seja, com poucos métodos realmente adaptados, ou mesmo outros onde o Adapter foi utilizado para cumprir outras funções que não a sua função principal [Ferenc et al., 2005]. Mas,

é possível demonstrar a aplicabilidade do Adapter numa aplicação de software bastante conhecida.

Bridge Foi bastante importante na actualização dos serviços da framework *Open Service Gateway Initiative* (OSGi), de acordo com o descrito em [Pohl and Gerlach, 2003].

Sobre o OSGi, importa dizer que se trata de uma plataforma de serviços para o desenvolvimento de aplicações Java, numa arquitectura orientada a serviços. No OSGi são registados serviços por um componente, e referenciados por outros. Um serviço trata-se de um objecto de uma classe que implementa um/vários interfaces de serviços. Para se obter um serviço, é necessário pedir em primeiro uma referência para esse serviço ao OSGi [Pohl and Gerlach, 2003].

O problema reside quando se pretende actualizar um serviço, sendo para isso necessário removê-lo e registá-lo de novo, podendo originar problemas nas referências já fornecidas. Então, com a utilização do padrão Bridge consegue-se superar esta dificuldade, usando mecanismos de indirecção para a actualização de um serviço [Pohl and Gerlach, 2003].

Decorator Em [Duffy et al., 2003] é referida a sua utilização numa aplicação destinada a capturar o perfil de aplicações orientadas aos objectos, sem que se altere o comportamento do sistema em questão.

Em relação ao perfil de uma aplicação podemos encontrar, entre outras coisas, o nível de utilização dos recursos que necessita para correr. Este perfil pode ser bastante útil para se optimizar o programa e para fazer testes e operações de *debug* [Duffy et al., 2003].

Como exemplo de utilização do Decorator, os autores abordam uma aplicação, bastante simples, que trata de simular as cores das luzes dos semáforos de trânsito, sendo acrescentada uma classe Profiler, de acordo com a estrutura do Decorator, para registar as mudanças que houve nas cores das luzes durante a execução da aplicação, não havendo portanto nenhuma alteração no comportamento da mesma, antes da introdução do Decorator [Duffy et al., 2003].

É ainda mencionada a utilização do Decorator para monitorizar a utilização da memória em aplicações C++, através da contagem de operações de alocação e libertação de memória, com os operadores *new* e *delete* [Duffy et al., 2003].

Facade Há que destacar a sua utilização na programação de jogos de computador, de acordo com o estudo descrito em [Gestwicki, 2007], onde, inicialmente, é dito que um jogo pode ser desenvolvido como uma aplicação Java, por exemplo, podendo ser apresentado de vários modos,

como por exemplo numa janela, em modo *full-screen* exclusivo, ou ainda em modo "quase" *full-screen*. É ainda mencionado que pode ser necessário alternar entre estes modos enquanto se está a desenvolver o jogo, nomeadamente em operações de *debug*, onde será mais útil não utilizar o modo *full-screen*.

Esta escolha diz apenas respeito à apresentação do jogo, não tendo que estar relacionada com a lógica subjacente ao jogo, embora por vezes haja casos onde estes dois conceitos se encontram misturados nas mesmas classes [Gestwicki, 2007].

Então, com o padrão Facade, é possível esconder a lógica do jogo numa única classe, cujos métodos serão os únicos a poder ser acedidos pelas outras classes. Consegue-se assim que o modo de apresentação do jogo possa ser alterado sem afectar a lógica do jogo, e que a lógica do jogo possa evoluir sem afectar a apresentação [Gestwicki, 2007].

Flyweight Pode-se considerar o exemplo descrito em [van Gurp and Bosch, 2002], onde é mencionada a sua utilização, além do State, no desenvolvimento da arquitectura de um simulador de uma máquina ATM. Os autores referem também a necessidade do desenho de uma arquitectura que não venha a ficar obsoleta, permitindo assim a manutenção mais simples do software, e também a adição de novas funcionalidades.

Tal como proposto em [van Gurp and Bosch, 2002], numa das suas versões para o desenvolvimento do simulador de uma máquina ATM, foi utilizado o Flyweight para reduzir o tempo relacionado com a criação de objectos. Assim, em vez de, para cada simulação, serem instanciados novos objectos relacionados com os vários estados da máquina ATM, são apenas instanciados uma vez, e partilhados nas vezes seguintes, dado que não guardam dados e são redundantes [van Gurp and Bosch, 2002].

2.2.3 Padrões Comportamentais

Nesta subsecção vai ser referida a aplicabilidade de alguns padrões comportamentais, entre os quais o Chain of Responsibility, o Iterator, o Observer, o State, o Strategy e o Visitor.

Chain of Responsibility Importa salientar os benefícios da sua utilização em várias aplicações, como indicado em [Vinoski, 2002], onde é mencionada a utilização do Chain of Responsibility tanto na elaboração de sistemas operativos, como de middleware (direccionado a sistemas distribuídos), permitindo uma comunicação eficiente e flexível entre uma aplicação cliente e objectos destino.

É um padrão de comportamento importante na elaboração deste tipo de aplicações, nomeadamente arquitecturas orientadas a serviços, que são parte fundamental de sistemas de middleware [Vinoski, 2002].

Iterator Em [Noble, 2000] é dito que é um dos padrões de concepção mais importantes, pois oferece acesso sequencial aos elementos de um conjunto de objectos.

Ao longo do presente artigo são abordados vários tipos de iteradores e as diferentes formas de encapsulamento que oferecem, agrupando-os em três grupos, sendo depois identificadas vantagens e desvantagens associadas a cada um. Mas o essencial a reter é a necessidade de utilizarmos recorrentemente este padrão no desenvolvimento de aplicações de software, existindo até uma classe Java para este efeito.

Observer Em relação a este padrão de comportamento, importa salientar a sua possível utilização num jogo de computador, de acordo com o descrito em [Gestwicki, 2007]. Assim, é dito que este padrão pode ser utilizado para efectuar o processamento, de forma assíncrona, dos controlos introduzidos a partir do teclado.

Então, quando alguém está a jogar, vai utilizando o teclado para interagir com o jogo, mas não é de todo viável estar a actualizar a *sprite* (elementos do jogo que não fazem parte do *background*, como o jogador ou um projéctil) da posição do jogador no método que processa um clique numa tecla; isto tira fluidez e suavidade ao jogo. Desta forma, para que a animação se mantenha suave, a velocidade de uma *sprite* deve ser definida com base no número de actualizações por segundo, e não no número de eventos (cliques no teclado) por segundo. Portanto, utilizando o Observer, a classe que processa os eventos de cliques no teclado escreve a informação associada ao estado das teclas num objecto partilhado; depois, a *thread* principal do jogo irá ler esse estado presente no objecto partilhado, na fase de actualização do ciclo principal do jogo [Gestwicki, 2007].

State Também em [Gestwicki, 2007] é analisada a utilização do padrão State para tratar do estado associado às *sprites* do jogo. Como é referido, o estado de uma *sprite* consiste frequentemente na sua localização, velocidade, tamanho, entre outros atributos. Os comportamentos associados às *sprites* são, por exemplo, a movimentação do jogador no terreno de jogo, ou as colisões entre o jogador e os inimigos.

Por vezes é necessário que as *sprites* se comportem de forma diferente, mediante o estado do jogo. Por exemplo, à medida que o jogador vai perdendo vida, poderá apresentar uma aparência diferente. O que se faz normalmente é representar os vários estados possíveis de uma *sprite* com uma variável (um inteiro, por exemplo), e depois definir métodos

com estruturas condicionais para atribuir o comportamento adequado para cada estado. Assim, a solução proposta reside na utilização do padrão State, onde cada estado possível é representado por uma subclasse de uma classe principal State, contendo a lógica de comportamento associada a cada estado definida em cada subclasse. Cada mensagem é delegada para a subclasse que representa o estado actual da *sprite* [Gestwicki, 2007].

Strategy Ainda em [Gestwicki, 2007], pode ser encontrada uma situação de utilização deste padrão comportamental, para lidar com a animação associada às *sprites* do jogo.

Tal como referido nesse artigo, pode-se optar por carregar uma série de imagens, e depois ir alternando entre elas, ou então desenhar cada *frame* com primitivas gráficas. No entanto, se a lógica da animação está misturada com a lógica da *sprite*, fica complicado manter/modificar esses componentes do programa. Como alternativa, é então sugerido o Strategy, onde se pressupõe a utilização de um interface Animation, juntamente com a adição de animações às *sprites*, em tempo de execução, sendo agora o desenho da *sprite* feito pelas subclasses de Animation.

Visitor Também em [Gestwicki, 2007] se encontra um exemplo de utilização do Visitor, relacionado com as colisões presentes num jogo de computador. Uma colisão pode ser, por exemplo, a perda de vida do personagem do jogo quando é atingido por um projéctil. Pode ser utilizada a opção de tratar os elementos do jogo como números inteiros, e utilizar estruturas condicionais para operar com eles. O problema seria que a lógica associada ao tratamento das colisões estaria espalhada por todo o programa, mas, recorrendo ao Visitor consegue-se evitar este problema.

2.3 *Pattern-Oriented Software Architecture*

Ao longo desta secção vão ser referidos, de forma breve, os padrões para uma arquitectura de software orientada aos objectos, que foram apresentados em [Buschmann et al., 1996], servindo para mostrar outros pontos de vista para além da abordagem escolhida para o desenvolvimento da presente dissertação, que é a adoptada em [Gamma et al., 1995].

Estes padrões foram agrupados em categorias: Structural Decomposition, Distributed Systems, Interactive Systems, Adaptive Systems, Organization of Work, Service Access, Resource Management e Communication [Buschmann et al., 1996]. Nas Tabelas 2.2 e 2.3 podem ser encontrados os vários padrões pertencentes a cada uma das categorias.

Blackboard Deve ser utilizado quando se pretende reunir conhecimento

Structural Decomposition	*Distributed Systems*	*Interactive Systems*
Blackboard	Broker	Model-View-Controller
Layers	Client-Dispatcher-Server	Presentation-Abstraction-Control
Pipes and Filters	Forwarder-Receiver	
Whole Part		

Tabela 2.2: *Pattern-Oriented Software Architecture (a) [Buschmann et al., 1996]*

Adaptive Systems	*Organization of Work*	*Service Access*	*Resource Management*	*Communication*
Microkernel	Master-Slave	Proxy	Counted Pointer	Publisher-Subscriber
Reflection			Command Processor	
			View Handler	

Tabela 2.3: *Pattern-Oriented Software Architecture (b) [Buschmann et al., 1996]*

proveniente de fontes variadas, recorrendo a um repositório partilhado [Avgeriou and Zdun, 2005].

O Blackboard será uma classe onde os vários dados (conhecimento) serão geridos, de forma centralizada. Cada utilizador poderá acrescentar novos dados ao Blackboard, que é gerido por um administrador.

Layers Este padrão deve ser utilizado quando temos um sistema com vários componentes que dependem uns dos outros [Avgeriou and Zdun, 2005].

Desta forma, os vários componentes devem ser separados em camadas verticais, onde cada uma oferece um interface bem definido. Cada camada interage com os serviços fornecidos pela camada imediatamente inferior, e fornece serviços à camada imediatamente superior. Consegue-se assim que a aplicação seja mais simples de manter, sendo ainda o código mais facilmente reutilizável.

Pipes and Filters Pode-se dizer que disponibiliza uma estrutura para sistemas que têm de processar um fluxo de dados.

Os *pipes* correspondem ao fluxo de dados que entra na aplicação, sendo que os *filters* equivalem à forma como vai ser efectuado o processamento dos dados, sendo estes lidos/modificados ao longo do processo, para que no final se obtenha o resultado desejado [Buschmann et al., 1996].

De realçar que poderão existir vários *filters* ao longo do processo, ou seja, várias classes que vão processando os dados, sendo bastante flexível a introdução/remoção destas classes para o processamento do fluxo de dados.

Whole Part Este padrão está relacionado com a utilização de vários componentes, aos quais o cliente não pode aceder directamente, numa só

classe, que será o único ponto de comunicação entre o sistema e o cliente [Buschmann et al., 1996]. Apenas os métodos definidos nessa classe poderão ser chamados pelo cliente, que não sabe nada acerca da sua implementação, com recurso aos vários componentes dessa classe.

Esta solução permite que possam ser alteradas propriedades dos componentes sem que o cliente se aperceba, pois ele utilizará sempre o mesmo interface para comunicar com o sistema. Além disso, estes componentes podem ainda ser reutilizados noutras aplicações informáticas.

Broker É normalmente utilizado em sistemas distribuídos, onde é necessário que haja coordenação entre o cliente e o servidor.

Assim, as questões relacionadas com a comunicação entre o cliente e o servidor são tratadas pelo *broker*, que funciona então como um mediador [Zdun et al., 2004].

Por exemplo, um sistema distribuído *peer-to-peer*, onde cada interveniente faz o papel de cliente e/ou servidor, precisa de um *broker* que permita coordenar a comunicação entre eles.

Desta forma, o *broker* vai sabendo quem está na rede, de modo a poder encaminhar o pedido de um *peer* para um dos *peers* capazes de o atender (situação onde o *broker* sabe quais os *peers* que têm o ficheiro pretendido pelo cliente, por exemplo).

Client-Dispatcher-Server Serve para introduzir um nível intermédio entre o cliente e o servidor, para que o cliente saiba sempre quem deve contactar (o intermediário, que depois lhe encaminhará o pedido), mas desconheça como o intermediário vai comunicar com o servidor [Buschmann et al., 1996].

Forwarder-Receiver Pode ser utilizado para implementar um modelo de comunicação *peer-to-peer*, onde o pedido de um *peer* direccionado para um outro *peer* destino, não vai directamente para este, mas vai sendo encaminhado por vários *peers* da rede [Buschmann et al., 1996].

Isto permite que os *peers* não tenham de ter conhecimento de todos os outros *peers* que constituem a rede, num dado momento, mas sim que saibam a localização de apenas alguns *peers*. Vão passando a mensagem para potenciais *peers* destino, e assim sucessivamente, até que a mensagem chegará ao *peer* destino pretendido.

Claro que para isto funcionar é preciso que haja alguma ordem associada aos *peers*, por exemplo identificadores únicos, onde cada *peer* contém a localização exacta de vários *peers* muito perto do seu identificador, e saiba a localização de poucos *peers* longe do seu identificador, para que possam encaminhar a mensagem através de grandes saltos, numa fase inicial, e depois com pequenos saltos, na fase final.

Model-View-Controller Este padrão é também conhecido por MVC, e propõe dividir uma aplicação interactiva em três partes diferentes.

Existe um Modelo que representa os dados essenciais da aplicação, e a forma como são tratados, independentemente da interface do programa; uma ou mais Vistas, que tratam de mostrar alguns dos conteúdos da aplicação ao utilizador; e um Controlador associado a cada uma das Vistas existentes, que vai receber dados do utilizador, e vai transformá-los em pedidos a invocar no Modelo [Avgeriou and Zdun, 2005].

Presentation-Abstraction-Control Pode-se dizer que define uma estrutura para sistemas interactivos, numa forma hierárquica, onde é possível tratar de forma independente cada um dos membros da hierarquia.

Por exemplo, na contagem dos resultados eleitorais, vão sendo contabilizados os votos das freguesias, que depois de somados originam os totais do concelho, que por sua vez depois de somados darão origem aos totais distritais, e assim sucessivamente. O resultado global não é modificado, e assim vão sendo apresentados vários resultados parciais, sendo simples acrescentar mais intervenientes no meio desta hierarquia [Buschmann et al., 1996].

Microkernel Pretende separar as funcionalidades mínimas essenciais de outras funcionalidades que podem ser acrescentadas [Buschmann et al., 1996].

Por exemplo, um computador pode ser visto como um microkernel, que providencia as funcionalidades mínimas essenciais para se operar com ele; uma aplicação de software pode ser vista como um conjunto de novas funcionalidades que são acrescentadas ao computador, sem que isso implique mudanças no próprio computador, nem na interface que disponibiliza para o cliente (teclado, entre outros).

Reflection Oferece um mecanismo para trocar a estrutura e o comportamento de um sistema, de forma dinâmica.

Temos um sistema definido com o recurso a um meta-nível, que contém informação sobre as propriedades do sistema, ao passo que o nível normal define a estrutura e o comportamento da aplicação. Por isso, fazendo modificações no meta-nível vai-se afectando a estrutura e o comportamento do programa [Suzuki and Yamamoto, 1998].

Master-Slave Diz respeito à utilização de um componente *Master*, que vai distribuir trabalho idêntico por vários componentes *Slave*, que depois de o terminarem, devolverão o resultado de cada um ao *Master*. Em seguida, o *Master*, com base nos resultados individuais recebidos, calculará o resultado final [Buschmann et al., 1996].

Este padrão tem o seu impacto na computação paralela, onde um processo principal delega trabalho para vários outros processos, e depois recebe esses resultados, agrega-os e devolve-os ao cliente, por exemplo. Em vez de se fazerem essas operações de forma sequencial, se for possível paralelizá-las, consegue-se um melhor desempenho.

Proxy Trata-se de um contentor para um objecto, permitindo ao cliente comunicar com o Proxy, a pensar que está a comunicar com o objecto real [Buschmann et al., 1996].

Com a indirecção utilizada consegue-se proteger o objecto real do cliente, indo os pedidos parar ao Proxy, e só depois ao objecto real.

Counted Pointer Está relacionado com a gestão manual da memória de uma aplicação informática, quando a linguagem de programação utilizada não o faz automaticamente [Buschmann et al., 1996].

Portanto, se houver a necessidade de optimizar a memória gasta pela aplicação, podemos ir alocando e libertando espaço, conforme as necessidades da aplicação, ou seja, à medida que os objectos vão sendo criados e à medida que deixam de ser necessários.

Command Processor Separa o pedido de um serviço da sua execução [Buschmann et al., 1996]. O cliente limita-se a interagir com o interface que lhe é disponibilizado para, por exemplo, efectuar a compra de um produto; em seguida, será desencadeado o processo de verificação/actualização do stock existente, recolhidos os dados do cliente, e efectuado o respectivo envio da mercadoria, por parte da empresa de vendas. As medidas a tomar após o pedido do cliente serão sempre as mesmas, independentemente da forma como chegar o pedido do cliente à empresa.

View Handler Pode-se dizer que ajuda a gerir todas as vistas existentes para um dado sistema. Permite que possam ser adicionadas novas vistas, sem que se altere as já existentes. Caberá ao View Handler oferecer um tratamento uniforme para todas as vistas [Buschmann et al., 1996].

Publisher-Subscriber É responsável por manter sincronizado o estado entre vários componentes de uma aplicação [Carvalho, 2002]. Então, teremos entidades que oferecem serviços, que são os *publishers*, e entidades interessadas nesses serviços, que são os *subscribers*.

Por exemplo, poderemos ter classes de uma aplicação interessadas em saber se uma dada variável de uma outra classe mudou, mas não podem estar sempre a perguntar isso, senão teremos uma situação de espera activa. Então, registam-se como entidades interessadas nesse evento, sendo que quando o evento acontecer, essas classes serão informadas do novo valor para a variável.

Desenvolvimento de Software	*Arquitectura do Software*	*Gestão de Projectos*
The Blob	Autogenerated Stovepipe	Blowhard Jamboree
Continuous Obsolescence	Stovepipe Enterprise	Analysis Paralysis
Lava Flow	Jumble	Viewgraph Engineering
Ambiguous Viewpoint	Stovepipe System	Death by Planning
Functional Decomposition	Cover Your Assets	Fear of Success
Poltergeists	Vendor Lock-In	Corncob
Boat Anchor	Wolf Ticket	Intellectual Violence
Golden Hammer	Architecture by Implication	Irrational Management
Dead End	Warm Bodies	Smoke and Mirrors
Spaghetti Code	Design by Committee	Project Mismanagement
Input Kludge	Swiss Army Knife	Throw It over the Wall
Walking through a Minefield	Reinvent the Wheel	Fire Drill
Cut-and-Paste Programming	The Grand Old Duke of York	The Feud
Mushroom Management		E-Mail Is Dangerous

Tabela 2.4: *Antipatterns [Brown et al., 1998]*

2.4 *Antipatterns*

Nas próximas subsecções serão analisados, de forma sucinta, os *antipatterns*, propostos em [Brown et al., 1998], que referem práticas comuns na gestão e no desenvolvimento de um projecto de software, que numa fase inicial podem parecer benéficas, mas que acabam por não o ser, e até prejudicam o próprio projecto. A sua inclusão neste capítulo sobre o estado da arte e trabalho relacionado tem por objectivo mostrar mais um ponto de vista, diferente do adoptado nesta dissertação, que é o seguido em [Gamma et al., 1995], servindo como informação complementar para o estudo elaborado.

Os *antipatterns* encontram-se agrupados em três categorias, que são: Desenvolvimento de Software, Arquitectura do Software e Gestão de Projectos [Brown et al., 1998]. Na Tabela 2.4 podem ser encontrados os vários *antipatterns* pertencentes a cada grupo.

2.4.1 Desenvolvimento de Software

Nesta subsecção vão ser apresentados, de forma muito breve, os *antipatterns* pertencentes à categoria Desenvolvimento de Software. Como o próprio nome indica, são situações a evitar seguir durante a programação de uma qualquer aplicação de software, de modo a não comprometer a sua manutenção e evolução futura, por exemplo.

The Blob Pode ser encontrado em aplicações de software onde uma única classe é responsável pelo processamento, havendo outras classes relacionadas com esta, mas que apenas servem para encapsular dados [Khomh et al., 2009].

O problema reside no facto de que a maioria da responsabilidade está apenas numa única classe.

Continuous Obsolescence Pode-se dizer que representa um problema associado à evolução da tecnologia, nos dias de hoje [Brown et al., 1998].

Assim, torna-se difícil haver informação actualizada para uma qualquer tecnologia, visto que no dia seguinte já poderá sair uma nova versão dessa mesma tecnologia. Isto dificulta o desenvolvimento de aplicações de software, onde é fundamental garantir que o programa em desenvolvimento vai funcionar com a versão actual de uma dada tecnologia, ou mesmo ser compatível com novas versões, por exemplo.

Lava Flow Este *antipattern* encontra-se frequentemente em aplicações de software que tiveram várias pessoas a programá-las, as quais já não fazem parte da empresa, ou mudaram de departamento, e por isso abandonaram a aplicação, que ainda não está terminada [Wirfs-Brock, 2007], tornando-se necessário colocar novas pessoas a continuar o desenvolvimento dessa aplicação.

O problema está na falta de documentação associada à aplicação, pois quem saiu não a fez, ou fez muito pouco. Isto faz com que os novos programadores não compreendam certos fragmentos de código, mas que tenham receio de os eliminar, por não saberem como corrigir o problema, caso a aplicação deixasse de funcionar por causa disso. Portanto, a tendência é para estes fragmentos continuarem na aplicação indefinidamente, podendo ter impactos negativos no desempenho da aplicação [Wirfs-Brock, 2007].

Ambiguous Viewpoint Está relacionado com os modelos que são elaborados durante o desenvolvimento de uma aplicação de software, onde deve ficar bem claro o ponto de vista associado ao modelo [Brown et al., 1998].

Quando isso não acontece, por vezes torna-se complicado separar os interfaces dos detalhes de implementação, perdendo-se uma característica importante associada à programação orientada aos objectos.

Functional Decomposition É um *antipattern* que aparece quando alguém que não está habituado ao paradigma de programação orientada aos objectos tem de implementar uma aplicação de software nesse paradigma [Moha et al., 2006].

Então, o que normalmente acontece é que a aplicação vai ser baseada numa rotina principal, que vai chamar várias subrotinas auxiliares, sendo que geralmente a cada subrotina vai corresponder uma classe, não havendo o aproveitamento da possibilidade de utilizar uma hierarquia de classes.

Poltergeists Está associado a classes com responsabilidades muito limitadas numa aplicação de software. São classes que, por exemplo, são

instanciadas apenas para invocar um método noutra classe, não servindo para mais nada. Assim, fazem com que se gastem recursos desnecessariamente, e dificultam a compreensão/manutenção da aplicação [Brown et al., 1998].

Boat Anchor Surge, por exemplo, quando está a ser desenvolvida uma aplicação de software e é necessário, numa dada ocasião, adquirir um novo componente de hardware, para a aplicação poder ser executada; algum tempo depois, essa aplicação deixa de ser utilizada, pois não era viável e não iria trazer os lucros esperados [Brown et al., 1998], acabando por se concluir que foi uma decisão precipitada que levou à perda de dinheiro por parte da empresa.

Golden Hammer Reflecte o comportamento de alguém que se limita a utilizar sempre a mesma solução genérica, já desenvolvida, para resolver qualquer problema novo que apareça.

No entanto, há muitas situações onde a solução genérica existente não é a mais indicada, sendo necessário pensar melhor o problema, e elaborar uma solução particular para esse novo problema [Rogers and Pheatt, 2009].

Dead End Ocorre quando se modificou um determinado componente, necessário para o funcionamento de uma aplicação de software, mas que por algum motivo foi descontinuado. Isto implica refazer a aplicação, e utilizar formas alternativas para que possa funcionar correctamente [Brown et al., 1998].

Spaghetti Code Este *antipattern* é tido como um dos mais conhecidos, e também dos que mais facilmente ocorrem. Desta forma, pode ser encontrado em aplicações muito pouco estruturadas, onde as várias funcionalidades vão sendo acrescentadas num sítio qualquer, sem preocupação nenhuma [Brown et al., 1998].

Torna-se muito difícil manter uma aplicação destas, sendo por vezes mais simples reescrever todo o código novamente. É mais frequente aparecer em aplicações não orientadas aos objectos, mas pode também existir em aplicações orientadas aos objectos, quando por exemplo existem poucas classes na arquitectura, e poucos métodos, mas com muitas linhas de código cada um [Moha et al., 2006].

Input Kludge Pode-se dizer que acontece quando uma aplicação de software não está preparada para responder a inputs inesperados, podendo estes fazer *crashar* a aplicação. Assim, para evitar isso, deve ser sempre feita a validação do input introduzido pelo utilizador [Brown et al., 1998].

Walking through a Minefield Diz respeito às aplicações de software em geral, que normalmente são colocadas no mercado ainda contendo *bugs* não identificados [Brown et al., 1998]. Mesmo após terem sido realizados muitos testes, identificado e corrigido muitos *bugs*, é muito difícil/impossível lançar uma aplicação de software sem ter um único *bug*.

Cut-and-Paste Programming Há que dizer que é uma forma comum, mas não muito aconselhada, de reutilizar software já elaborado.

O problema é que assim os erros existentes serão replicados, tornando difícil manter a aplicação a longo prazo, pois quando se alterar um pedaço de código numa classe, provavelmente essa alteração terá de ser repetida em muitas outras classes [Hammouda and Harsu, 2004]. A curto prazo, pode até ser uma boa opção, permitindo programar com base na modificação de código existente, indo mais rapidamente ao encontro dos requisitos propostos.

Mushroom Management Acontece quando os programadores de uma aplicação de software são "obrigados" a programar tendo apenas por base os diagramas entretanto elaborados por outros colegas.

Estes diagramas podem não estar totalmente claros, o que vai fazer com que os programadores tomem decisões de implementação, por vezes erradas, não satisfazendo os requisitos da aplicação, levando a que a solução final não seja o que o cliente pediu [Brown et al., 1998].

2.4.2 Arquitectura do Software

Ao longo desta subsecção serão referidos, sucintamente, os *antipatterns* pertencentes à categoria Arquitectura do Software. Estão relacionados com a etapa de especificação de uma aplicação de software, onde, por exemplo, se evitou tomar decisões importantes, ou se desenhou um diagrama extremamente complexo, o que irá dificultar imenso a etapa de programação, e, consequentemente, a manutenção e evolução do programa elaborado.

Autogenerated Stovepipe Este *antipattern* surge quando se está a tentar migrar uma aplicação de software já desenvolvida para uma infraestrutura distribuída.

O problema aparece ao converter os interfaces existentes para interfaces distribuídos, podendo haver um conjunto de operações cuja performance melhoraria se se tirasse partido da arquitectura distribuída, levando a que muita coisa tenha de ser revista [Brown et al., 1998].

Stovepipe Enterprise Está relacionado com sistemas de software com uma arquitectura desenvolvida de forma *ad hoc* [Brown et al., 1998].

Acontece por vezes quando várias pessoas estão a desenvolver uma aplicação, não havendo coordenação/comunicação entre elas, desenvolvendo cada uma a sua parte, à sua maneira, havendo depois problemas de integração entre as partes.

Jumble Este *antipattern* ocorre quando, numa aplicação de software, os elementos de design verticais e horizontais aparecem misturados. Os elementos de design verticais estão dependentes das aplicações e das implementações específicas utilizadas; os elementos de design horizontais são os mais comuns nas aplicações e nas implementações específicas [Brown et al., 1998].

Stovepipe System Está presente numa aplicação onde é muito difícil acrescentar funcionalidades e adaptar o programa às mudanças nos requisitos. É uma aplicação que foi desenvolvida com pouca coordenação e planeamento ao longo de todo o processo [Brown et al., 1998].

Cover Your Assets Pode ser identificado quando, na fase de análise de requisitos e especificação de uma aplicação de software, se evita tomar decisões importantes, por causa do receio de se cometer erros [Kärpijoki, 2001].

Então, os autores procuram soluções alternativas, tentando "contornar os problemas" na fase de elaboração da documentação, que vai depois ser dada a alguém que vai ter de a ler para poder programar a aplicação. Vai ter essa tarefa dificultada, pois a documentação produzida não lhe vai ser muito útil, não havendo abstracções claras para implementar.

Vendor Lock-In É um *antipattern* que se verifica quando um projecto de software está completamente dependente de uma tecnologia proprietária [Brown et al., 1998].

Isto leva a que quando novas versões dessa tecnologia sejam lançadas, o projecto de software necessite de manutenção, para continuar a funcionar com a nova versão, pois muitas das vezes não há compatibilidade entre as versões lançadas. É também possível que o lançamento de uma nova versão seja atrasado, obrigando a que quem desenvolve o software tenha de esperar para o poder desenvolver.

Wolf Ticket Representa um produto que afirma estar em abertura e conformidade com os standards. No entanto, os produtos são vendidos com interfaces proprietários que podem variar significativamente dos standards. O problema é que os standards são mais importantes para os fornecedores de tecnologia, que vêem assim a sua marca ser reconhecida, em vez de oferecerem benefícios aos consumidores [Brown et al., 1998].

Architecture by Implication Acontece quando há falta de especificação na arquitectura de um sistema de software em desenvolvimento.

Normalmente, isto sucede porque os responsáveis pelo projecto têm já alguma experiência na construção de sistemas de software, assumindo que a elaboração de documentação é desnecessária, levando a que se corram riscos desnecessários, e por vezes comprometendo o próprio projecto [Brown et al., 1998].

Warm Bodies Diz respeito ao número ideal de programadores a utilizar num projecto de software de pequena escala. O ideal são equipas de quatro elementos, pois com mais elementos a comunicação e a tomada de decisões já se torna mais difícil, podendo comprometer os prazos do projecto [Brown et al., 1998].

É errado pensar que, em situações de atraso face aos prazos, se pode arranjar mais programadores para juntar à equipa; só vai atrapalhar os que lá estão, pois não é fácil entrar num projecto a meio do seu desenvolvimento, sem estar por dentro dos requisitos, entre outras coisas, por exemplo.

Design by Committee Pode ser identificado quando o design de uma aplicação de software é extremamente complexo, fruto da inclusão das opiniões de todos os intervenientes na sua elaboração, numa perspectiva democrática, onde todos quiseram ver as suas ideias reflectidas na solução final.

Isto leva a que os programadores não consigam compreender a especificação no tempo desejado, originando problemas na implementação da aplicação [Kärpijoki, 2001].

Swiss Army Knife Trata-se de uma classe extremamente complexa, que disponibiliza um número excessivo de métodos para poderem ser invocados [Moha et al., 2008].

Quem desenvolveu essa classe procurou que servisse para ser utilizada num número exagerado de casos, tornando-se muito pouco claro para quem vai interagir com ela, saber para que serve realmente. É muito difícil ter documentação para uma classe deste tipo, e fazer *debug* ou manutenção são tarefas quase impraticáveis.

Reinvent the Wheel Está relacionado com o desenvolvimento de aplicações de software que não reutilizam soluções já existentes, para problemas semelhantes [Brown et al., 1998]. Ao invés, toda a aplicação é desenvolvida a partir do zero.

The Grand Old Duke of York Este *antipattern* está ligado à capacidade de quem desenvolve software ser capaz de elaborar boas abstracções,

sendo que é referido que poucos são os que realmente o conseguem. Podem ser bons a programar, mas nem sempre conseguem ser também bons a definir abstracções [Brown et al., 1998].

Desta forma, devem ser estes os responsáveis pelo design da arquitectura do sistema a desenvolver, evitando-se assim muitos detalhes desnecessários nesse nível de abstracção, que só irão complicar a posterior implementação da solução.

2.4.3 Gestão de Projectos

Durante esta subsecção vão ser analisados, de forma breve, os *antipatterns* que pertencem à categoria Gestão de Projectos. Englobam um conjunto de comportamentos tidos pelos elementos de uma equipa que está a desenvolver um projecto de software, desde os responsáveis pela especificação da aplicação, os responsáveis pelo desenvolvimento propriamente dito, até ao gestor do projecto.

São comportamentos que podem levar ao incumprimento dos prazos previstos, por exemplo, pelo excesso de tempo gasto na especificação, devido a uma má gestão do projecto, por conflitos entre elementos da mesma equipa, ou mesmo pela incapacidade do gestor do projecto.

Blowhard Jamboree Este *antipattern* está relacionado com pessoas supostamente experientes em tecnologia que pretendem influenciar na escolha de uma dada tecnologia em detrimento de outra, no desenvolvimento de um projecto de software [Laplante et al., 2007].

O problema é que muitas vezes se limitam a apresentar a sua opinião pessoal, ou o que ouviram nos meios de comunicação, sem ser baseada em fundamentos sérios, ou seja, sem ter por base um estudo/experiência com a tecnologia em causa.

Analysis Paralysis É um *antipattern* bastante frequente em aplicações orientadas aos objectos. Durante a fase de análise de uma aplicação de software, é suposto que o problema seja decomposto nas várias partes que o constituem, mas não há nenhuma fórmula para saber qual o nível de decomposição a utilizar.

Então, temos um problema quando esta fase de análise se prolonga demasiado, tardando a aplicação a entrar na fase de implementação. Começa-se a perder demasiado tempo na construção de diagramas, com detalhes em demasia, tentando que fiquem absolutamente perfeitos, mas na realidade vão é ficar muito mais difíceis de compreender por quem vai ter de os codificar [Kärpijoki, 2001].

Viewgraph Engineering Pode-se dizer que está relacionado com a situação de, por vezes, quem está a desenvolver uma aplicação de software,

tenha de parar essa actividade para escrever a documentação associada, ou então para preparar apresentações relacionadas com o projecto, para depois as mostrar aos clientes, ou mesmo a outros membros da empresa [Laplante et al., 2007]. Isto pode levar a que a aplicação não fique pronta a tempo.

Death by Planning Refere-se a situações onde é dada extrema importância ao planeamento de um projecto de software, não deixando que se comece a fazer nada até que o plano esteja pronto, pois se acredita que se se seguir à risca o plano, tudo correrá bem, e de acordo com o previsto. O que provavelmente vai acontecer é que a fase de desenvolvimento do software vai começar muito tarde [Laplante, 2004].

Fear of Success Este *antipattern* pode ser visto como os comportamentos que, por vezes, alguns intervenientes no desenvolvimento de um projecto de software podem apresentar na fase final, quando o produto está prestes a ser lançado [Brown et al., 1998].

Assim, algumas pessoas começam a ficar excessivamente preocupadas sobre assuntos que podem correr mal; começam a ficar inseguras sobre as suas competências a nível profissional, afectando o desenrolar da fase final do desenvolvimento do projecto.

Corncob Caracteriza os membros de uma equipa de desenvolvimento de software que têm um temperamento muito difícil de lidar [Kärpijoki, 2001]. As suas atitudes podem ser devido à sua personalidade, ou por vezes apenas para se tentarem impor na equipa/empresa onde trabalham.

Intellectual Violence Está relacionado com situações onde pessoas muito experientes numa dada matéria utilizam esse conhecimento para intimidar outros colegas de trabalho [Laplante et al., 2007].

Como os colegas podem ter receio de admitir que não sabem muito do tema, tentam fugir dele ao longo do desenvolvimento do projecto, atrasando-o. Por isso é muito importante encorajar a partilha de conhecimento entre todos os membros de uma equipa de trabalho.

Irrational Management Pode-se dizer que está relacionado com a incapacidade do gestor de projecto em tomar as melhores decisões. Pode acontecer por incapacidade da pessoa, não tendo o perfil apropriado para a responsabilidade exigida pelo cargo, ou então por ser obcecado em detalhes por aspectos que lhe interessem pessoalmente, nem sempre sendo o melhor para o projecto [Brown et al., 1998].

Por vezes pode ocorrer devido ao gestor ter entrado para um projecto a meio do seu desenvolvimento, não estando totalmente por dentro do

assunto nem das capacidades dos membros da equipa de desenvolvimento.

Smoke and Mirrors Ocorre quando uma empresa se compromete com um cliente em satisfazer todos os requisitos para uma determinada aplicação de software no prazo previsto, mas de facto não o vai conseguir fazer [Brown et al., 1998].

Isto acontece porque os prazos foram mal estimados, ou porque as funcionalidades a implementar são complicadas. Assim, o produto final vai ser entregue fora do prazo previsto, e com um custo mais elevado, ou então é entregue a tempo, mas não faz tudo o que era pretendido pelo cliente.

Project Mismanagement Está relacionado com a monitorização de um projecto de software, enquanto está a ser desenvolvido, desde a fase inicial de análise até à fase de testes [Brown et al., 1998].

Os problemas surgem quando foi feito um mau desenho da arquitectura, obrigando a ter de ser revisto; quando o código não é bem revisto, deixando escapar alguns *bugs*, ou então quando os testes não cobrem a totalidade do código, podendo aí haver *bugs* que não foram encontrados.

Throw It over the Wall É um *antipattern* relativo à produção da documentação ao longo do desenvolvimento de um projecto de software. Assim, numa grande parte dos casos, a documentação elaborada ao longo das várias fases do projecto, vai ser difícil de compreender, no momento de passar de uma fase para outra [Brown et al., 1998].

Isto acontece porque nem sempre é simples explicar todos os detalhes técnicos num relatório deste tipo, sendo aconselhável que haja comunicação entre os membros da equipa de desenvolvimento para ajudar a clarificar estes assuntos.

Fire Drill Caracteriza situações onde a equipa de gestão e planeamento de um projecto de software vai atrasando o início do desenvolvimento de uma aplicação de software, devido a questões técnicas e políticas [Brown et al., 1998].

Após alguns meses, começa a ser claro para a equipa de gestão que vai ser uma missão muito difícil/impossível entregar o projecto no pouco tempo que resta, e começa a pressionar a equipa de desenvolvimento.

Como consequências, o projecto vai pecar nos testes efectuados, e a documentação também não vai estar muito detalhada, pois a equipa de desenvolvimento passou a ter como prioridade terminar o produto, independentemente dos outros aspectos já referidos.

The Feud Pode ser encontrado em casos onde há conflitos entre os elementos da equipa de gestão de projecto, afectando seriamente a qualidade do ambiente de trabalho [Laplante, 2004].

Esta situação leva a que a produtividade da equipa de desenvolvimento baixe, havendo falta de comunicação e consequentemente menor passagem de conhecimento tecnológico entre os elementos da equipa.

E-Mail Is Dangerous Apesar de o e-mail representar um meio de comunicação importante entre os membros de uma equipa de desenvolvimento, não é o mais apropriado para certas discussões [Brown et al., 1998].

Por vezes há trocas de e-mails exageradas entre membros de uma equipa para exprimir opiniões menos positivas sobre colegas de trabalho. Como se sabe, estes e-mails tenderão a cair nas mãos erradas, provocando mau ambiente entre os membros de uma equipa. Por isso, é muito importante ter muito cuidado com o que se escreve num e-mail.

2.5 Ferramentas Existentes

Nas próximas subsecções serão analisadas algumas ferramentas de geração de código, já desenvolvidas, para os padrões de concepção apresentados em [Gamma et al., 1995], que são: COGENT [Budinsky et al., 1996], PEKOE [Reiss, 2000] e um Catálogo de Padrões na *Web* [Welicki et al., 2006a].

Além destas ferramentas, vai também ser analisada brevemente uma aplicação que procura sugerir os padrões de concepção mais adequados ao problema de design que o utilizador pretende resolver, que é o *Design Pattern Intent Ontology* (DPIO) [Kampffmeyer and Zschaler, 2007].

2.5.1 COGENT

O COGENT é uma ferramenta já desenvolvida e que permite a obtenção de código a partir dos padrões de concepção presentes em [Gamma et al., 1995]. A ferramenta foi proposta por Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides e Patsy S. Yu em [Budinsky et al., 1996].

O COGENT apresenta um interface gráfico, o que simplifica bastante a interacção do utilizador com a aplicação. A ferramenta vai gerar o código C++ associado ao padrão desejado, após o utilizador indicar alguns dados imprescindíveis para a respectiva geração de código. Além disto, os autores optaram também por colocar no COGENT ligações em formato html para partes significativas de [Gamma et al., 1995], de modo a elucidar melhor o utilizador, na hora de utilizar a ferramenta de geração de código.

Tal como referido em [Budinsky et al., 1996], apesar de se saber, mais ou menos, à partida o código associado a cada padrão de concepção, não é muito evidente depreender como o adaptar às situações concretas, quando

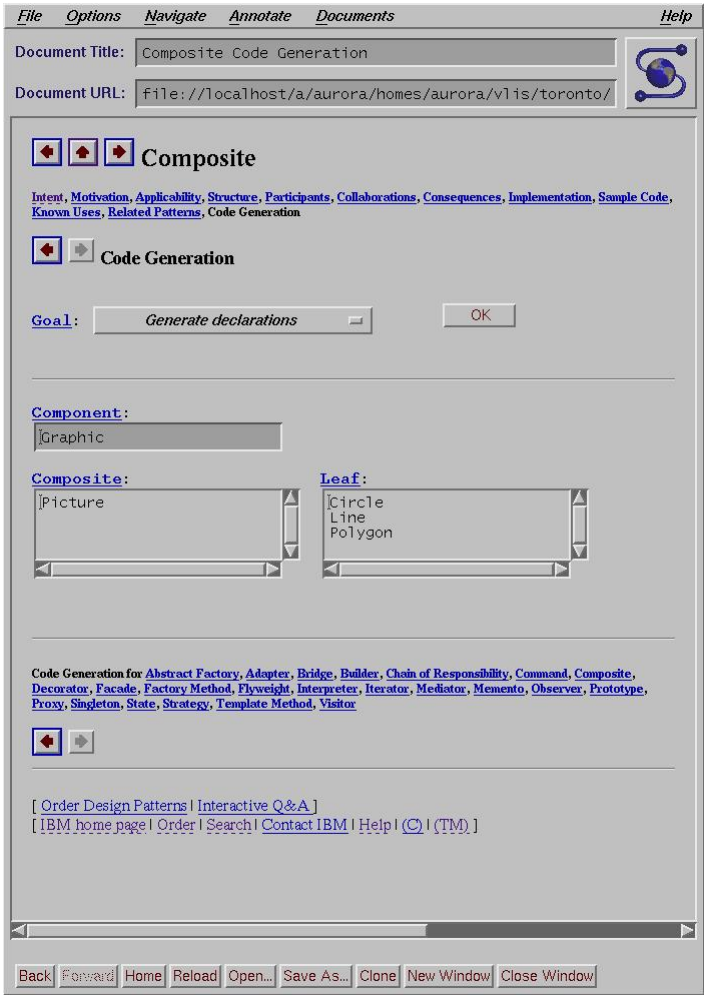

Figura 2.1: *COGENT - Exemplo de geração de código para o Composite [Budinsky et al., 1996]*

se está a desenvolver uma aplicação informática; além disso, por vezes é necessário reajustar a implementação efectuada, para se poder incluir uma nova funcionalidade, ou mesmo outros objectos na estrutura, o que pode implicar refazer o processo de escrita de código para o padrão em causa.

Assim, o intuito dos autores passou por colmatar estas dificuldades, visto que com a existência de uma ferramenta que ajude a escrever código, certamente que a tarefa do programador fica simplificada. Obviamente que o programador terá de saber programar com base nos padrões de concepção, e por isso mesmo a opção de inclusão, por parte dos autores, das já referidas ligações em formato html para partes de [Gamma et al., 1995].

O COGENT começa por mostrar uma pequena informação relacionada com cada padrão de concepção, para facilitar o processo de selecção por parte

do utilizador. Na secção relativa à geração de código, como se pode constatar pela Figura 2.1, são pedidos diferentes dados, mediante o padrão a escolher. De acordo com o descrito em [Budinsky et al., 1996], para o padrão Composite, é pedido que o utilizador introduza o nome da classe que terá o papel de Component, da classe que terá o papel de Composite, e, por fim, das classes que serão implementadas sob a forma de Leaf (pode ser mais do que uma, de acordo com o pretendido pelo utilizador). Esta informação é necessária para o COGENT poder gerar as classes C++ com as respectivas dependências, propostas pelo padrão Composite. O COGENT disponibiliza ainda uma opção de gerar também comentários juntamente com o código C++, relativos ao papel de cada classe e de cada método no padrão Composite, neste caso. O utilizador poderá depois então gravar o código gerado.

Para finalizar a análise do COGENT, importa salientar as conclusões dos autores, onde é dito que, com o desenvolvimento desta ferramenta, se conseguem tornar mais úteis os padrões de concepção, ajudando assim a programação de aplicações informáticas. No entanto, tal como referido, o COGENT é apenas o princípio numa área onde há muito por explorar, até que se torne o mais evidente possível, de modo a que os padrões de concepção possam vir a ser cada vez mais utilizados [Budinsky et al., 1996].

2.5.2 PEKOE

Ao longo desta subsecção será abordada outra ferramenta de geração de código, neste caso Java, baseada nos padrões de concepção [Gamma et al., 1995]. A ferramenta é o PEKOE e foi proposta por Steven P. Reiss em [Reiss, 2000].

Para começar, há que referir que os padrões de concepção desempenham um papel muito importante no desenvolvimento de sistemas de software, pois oferecem soluções já testadas e fiáveis para problemas que por vezes podem ser difíceis de implementar. Além disso, tornam mais simples de descrever e de compreender um sistema complexo, como são na maior parte das vezes, as aplicações informáticas programadas. Os padrões de concepção permitem também que tenhamos um nível de abstracção mais elevado durante o pensamento e implementação de uma solução de software [Reiss, 2000].

Actualmente, os padrões de concepção são utilizados no paradigma de programação orientado aos objectos, tendo por objectivo servir de "receita" para a utilização de objectos e métodos, de modo a resolver um problema, que se enquadre com algum dos padrões de concepção. No entanto, para se conseguir tirar o máximo de proveito dos padrões de concepção, é necessário que sejam incorporados em todo o processo de desenvolvimento de uma aplicação informática, e não somente na etapa de concepção. Além disso, corre-se o risco do novo código escrito durante uma revisão/manutenção da aplicação já não respeitar o padrão utilizado [Reiss, 2000].

Por isso, foi desenvolvido o PEKOE, que é uma ferramenta que pretende

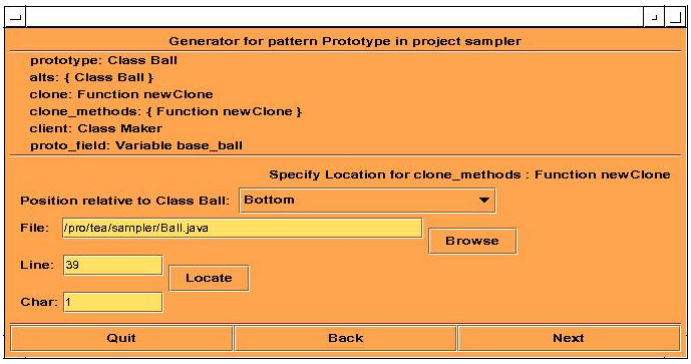

Figura 2.2: *PEKOE - Exemplo de geração de código para o Prototype [Reiss, 2000]*

fazer dos padrões de concepção uma parte integral no desenvolvimento de novas aplicações informáticas [Reiss, 2000]. Apresenta uma interface gráfica, como se verifica pela observação da Figura 2.2, simplificando a interacção do utilizador com o programa.

Esta ferramenta oferece aos utilizadores a funcionalidade de encontrar instâncias dos vários padrões, podendo assim ver já alguns exemplos em concreto. Desta forma, tenta-se simplificar a escolha do padrão mais adequado para o problema que o utilizador pretende solucionar. Além disto, o PEKOE possibilita que os utilizadores possam editar o código fonte associado a cada padrão, guardando novas instâncias para futura utilização.

Sobre o PEKOE, interessa ainda referir que a sua implementação serve para demonstrar que é possível um programador desenvolver software com base nos padrões de concepção, com o auxílio de uma ferramenta de apoio que faça a geração do código associado a esses padrões [Reiss, 2000]. Desta forma, será de esperar que, futuramente, com a existência de mais aplicações capazes de ajudar os programadores, se consiga pensar num nível de abstracção mais alto, permitindo o desenvolvimento mais ágil de sistemas de software de grande escala.

2.5.3 Catálogo de Padrões na *Web*

É uma aplicação *web* proposta por León Welicki, Juan M. C. Lovelle e Luis J. Aguilar em [Welicki et al., 2006a], que cataloga vários padrões de concepção. Permite aos seus utilizadores a pesquisa pelo padrão adequado, e, além disso, obter o código para a linguagem pretendida (Java ou C#, entre outras).

É afirmado que os padrões de concepção representam conhecimento proveniente da experiência, e, por isso, é muito importante para os engenheiros de software o acesso a este conhecimento. Desta forma, vai ser possível elevar os níveis de conhecimento da indústria de desenvolvimento de software [Welicki et al., 2006a].

Para começar a desenvolver a aplicação *web*, em primeiro lugar, os auto-

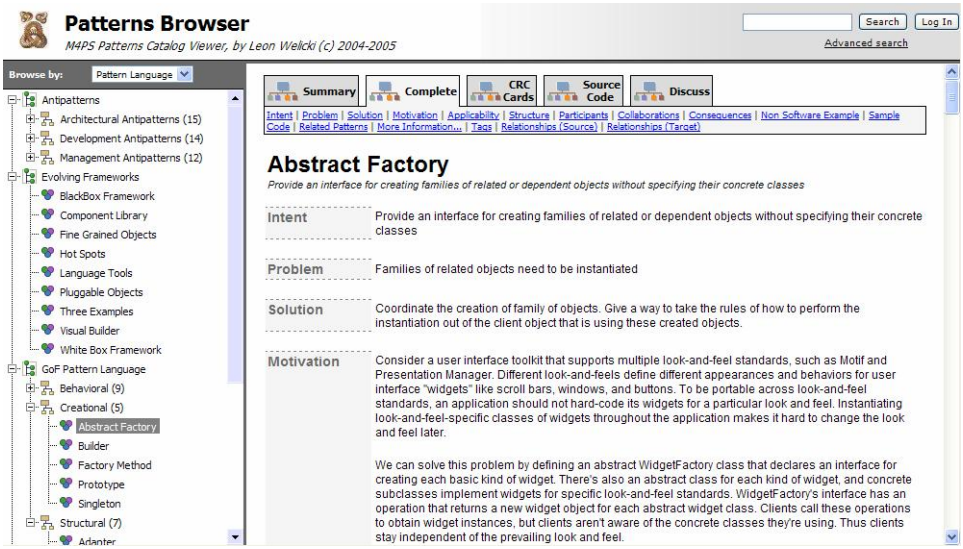

Figura 2.3: *Exemplo de informação adicional para o Abstract Factory [Welicki et al., 2006a]*

res resolveram criar uma linguagem de meta-especificação para descrever os padrões a catalogar, que é a linguagem *Entity Meta-specification Language* (EML), independente de qualquer linguagem de programação e plataforma [Welicki et al., 2006a]. Pretendiam assim resolver a questão de haver muita informação dispersa sobre os vários padrões de concepção, mesmo casos de padrões definidos apenas para uma dada linguagem de programação.

Com o recurso à linguagem EML, foi feita a descrição de todos os padrões a catalogar, de uma forma uniforme, juntando esse conhecimento disperso. Só depois disto é que se entrou no desenvolvimento propriamente dito da aplicação *web*, capaz de interpretar a linguagem EML, e mostrar ao utilizador a informação e o código para os padrões catalogados. Na Figura 2.3 pode ser vista a interface desta aplicação *web*, com destaque para a informação relativa ao padrão de criação Abstract Factory.

Uma vez que a aplicação foi desenvolvida para a *web*, os utilizadores apenas necessitam de um *browser* para ter acesso a esta informação, tornando assim o programa independente de qualquer plataforma.

Para concluir, importa ainda referir alguns dos benefícios, de acordo com os autores desta aplicação *web*, como por exemplo a utilização da linguagem EML para uniformizar todo o conhecimento dos padrões de concepção, independentemente das linguagens de programação e da plataforma; e ser capaz de oferecer, numa única aplicação acessível a qualquer utilizador, todo esse conhecimento, resultante de experiências anteriores [Welicki et al., 2006a].

2.5.4 *Design Pattern Intent Ontology*

Nesta subsecção vai ser analisado, de forma breve, o programa apresentado em [Kampffmeyer and Zschaler, 2007], responsável por sugerir o padrão de concepção (ou os padrões de concepção), de acordo com o catálogo proposto em [Gamma et al., 1995], mais adequados para a resolução do problema particular de design do utilizador desta aplicação. Esse programa é o DPIO.

Assim, inicialmente é afirmado que, para programadores ainda com pouca experiência, é complicado identificar qual o padrão ou quais os padrões de concepção que poderão implementar para solucionar o seu problema. Então, torna-se útil que haja um sistema capaz de sugerir um padrão para o problema em concreto do utilizador [Kampffmeyer and Zschaler, 2007].

É referido que para que seja possível implementar um mecanismo deste género, é necessário especificar de forma formal os padrões de concepção, para que depois se possa interrogar a aplicação, tendo por base o problema de design do utilizador. Foi também tomado em consideração que um determinado padrão pode solucionar vários problemas, e que um dado problema pode ser solucionado por mais do que um padrão [Kampffmeyer and Zschaler, 2007].

Quanto ao funcionamento da aplicação, o utilizador vai podendo adicionar informação relativa ao seu problema, na forma de predicados e conceitos já presentes na aplicação, como se pode verificar nas Figuras 2.4 e 2.5, respectivamente.

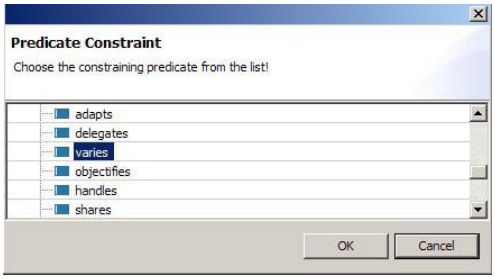

Figura 2.4: *Exemplo de escolha de predicados [Kampffmeyer and Zschaler, 2007]*

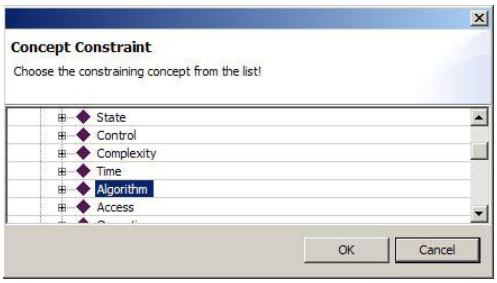

Figura 2.5: *Exemplo de escolha de conceitos [Kampffmeyer and Zschaler, 2007]*

O utilizador terá de ser capaz de contextualizar o seu problema de de-

sign nos predicados e conceitos suportados pelo programa. Tal como exemplificado em [Kampffmeyer and Zschaler, 2007], foi seleccionado o predicado "variar" e o conceito de "algoritmo", significando que o problema do utilizador está relacionado com a implementação de um algoritmo que pode variar. Como resultados foram sugeridos o Strategy e o Template Method, sendo que ambos podem implementar o conceito "algoritmo", de modo a ser definido de várias formas, ou seja, tenha a possibilidade de "variar".

Capítulo 3

Apresentação do Problema

Ao longo deste capítulo vai ser abordado, com algum detalhe, o problema relacionado com o tema da dissertação. Assim sendo, interessa nesta fase relembrar que, em linhas gerais, se pretende desenvolver uma aplicação informática para fazer a geração automática de código para os padrões de concepção presentes em [Gamma et al., 1995]. É este o problema da presente dissertação.

Para começar, torna-se importante fazer uma análise mais detalhada sobre alguns desses padrões de concepção, de modo a explicar melhor cada um deles. Serão então analisados dois padrões de criação, três padrões estruturais, e ainda três padrões comportamentais. Para cada padrão vai ser feita uma breve descrição, ou seja, os problemas para os quais propõe uma determinada solução; além disso, serão mostrados diagramas para ilustrar a sua estrutura, e um exemplo de utilização associado.

Quanto aos restantes padrões de concepção, pertencentes aos referidos subgrupos e não abordados neste capítulo, a sua análise pode ser encontrada nos Anexos A, B e C.

Terminada a análise de cada padrão de concepção, irá ser efectuada uma breve descrição da aplicação a desenvolver, já com a apresentação do seu Modelo do Domínio, e também dos requisitos associados ao problema em causa.

3.1 Padrões de Criação

Durante as próximas subsecções serão analisados dois padrões de criação, o Builder e o Factory Method, que pertencem ao primeiro dos três subgrupos de padrões de concepção [Gamma et al., 1995].

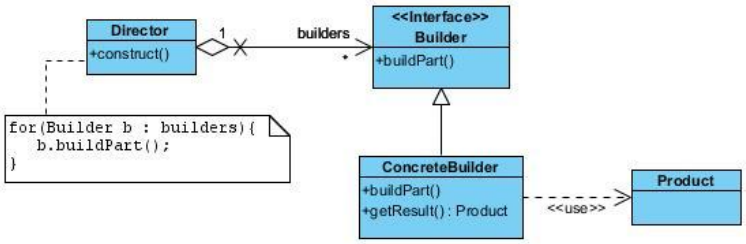

Figura 3.1: *Estrutura do Builder (adaptada de [Gamma et al., 1995])*

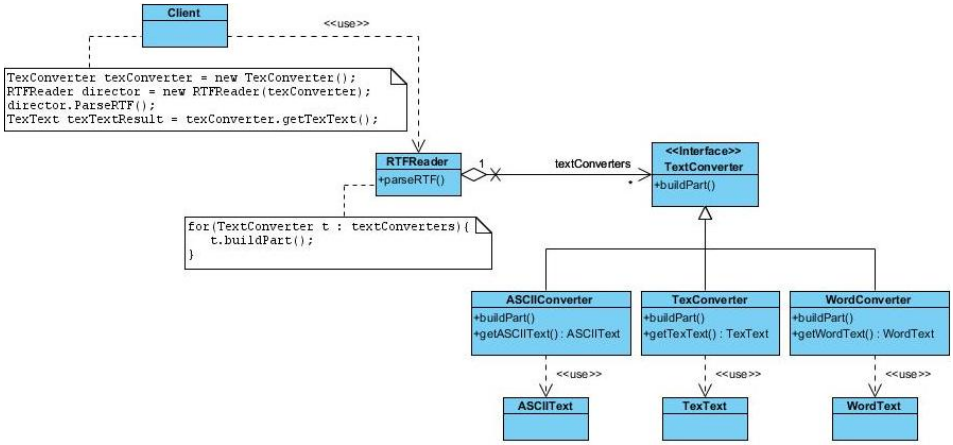

Figura 3.2: *Exemplo de Utilização do Builder (adaptada de [Gamma et al., 1995])*

3.1.1 Builder

Este padrão separa a construção de um objecto complexo da sua representação, fazendo com que o mesmo processo de construção origine representações diferentes [Gamma et al., 1995].

Como se verifica através da observação da Figura 3.1, a classe Builder é uma classe totalmente abstracta (um interface) que contém um método para criar as partes de um objecto. Esse método é implementado pelas subclasses de Builder (neste caso temos apenas uma subclasse de Builder, que é a classe ConcreteBuilder). Essa classe contém também o método getResult() que vai devolver um objecto do tipo Product. Se tivéssemos mais subclasses de Buider, cada uma delas teria um método getResult(), que devolveria um produto diferente mediante a subclasse utilizada. Em relação à classe Director, vai construir um objecto a partir do interface Builder.

Na Figura 3.2 encontra-se um exemplo de utilização deste padrão, onde pretendemos ler um ficheiro no formato *.rtf*, e converter o seu conteúdo para um de vários outros formatos (ASCII, LaTeX e Word) [Gamma et al., 1995].

Em primeiro lugar, o cliente começa por criar uma instância do ConcreteBuilder que deseja, por exemplo TexConverter. Depois, cria uma instância

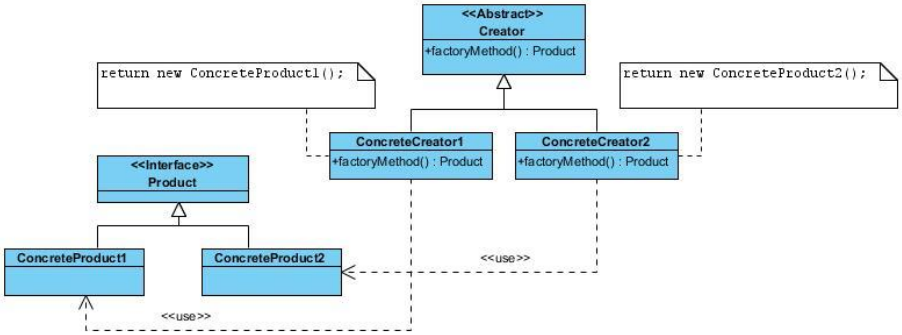

Figura 3.3: *Estrutura do Factory Method (adaptada de [Gamma et al., 1995])*

de RTFReader, dando-lhe como parâmetro a instância de TexConverter entretanto criada. Em seguida apenas vai ter de invocar o método parseRTF() (equivalente ao construct()) para que o produto desejado vá sendo construído pelo Builder (interface TextConverter neste exemplo). Terminada a construção, o cliente necessita de chamar o método getTexText() da instância de TexConverter já criada, e vai receber como valor de retorno o produto TexText pretendido.

O padrão Builder é semelhante ao Abstract Factory (consultar Anexo A), pois é também utilizada uma fábrica concreta para criar um conjunto de componentes coordenados [Tokuda and Batory, 1995]. Além disso, podem também ser utilizadas diferentes implementações do interface Builder (neste caso, diferentes implementações do interface TextConverter), para obtermos diferentes comportamentos (neste caso, diferentes representações do mesmo texto *.rtf* original).

Este padrão permite que seja relativamente simples acrescentar outras representações destino, bastando para isso que seja criada uma nova classe que implemente o interface Builder, e disponibilize um método para devolver o novo produto. Além disso, consegue-se isolar o código para a representação e para a criação.

3.1.2 Factory Method

Relativamente ao padrão Factory Method, pode-se dizer que tem por objectivo definir um interface para a criação de um objecto, mas deixando as subclasses decidir qual a classe a instanciar [Gamma et al., 1995]. É também semelhante ao Abstract Factory, tal como mencionado em [Ellis et al., 2007], na medida em que permite ao cliente obter instâncias de uma classe que para ele é desconhecida, mas que implementam o interface nela definido. Na Figura 3.3 encontra-se a estrutura deste padrão de criação.

Através da sua observação, verifica-se que a classe Creator contém o método factoryMethod() que devolve um objecto do tipo Product. Vemos tam-

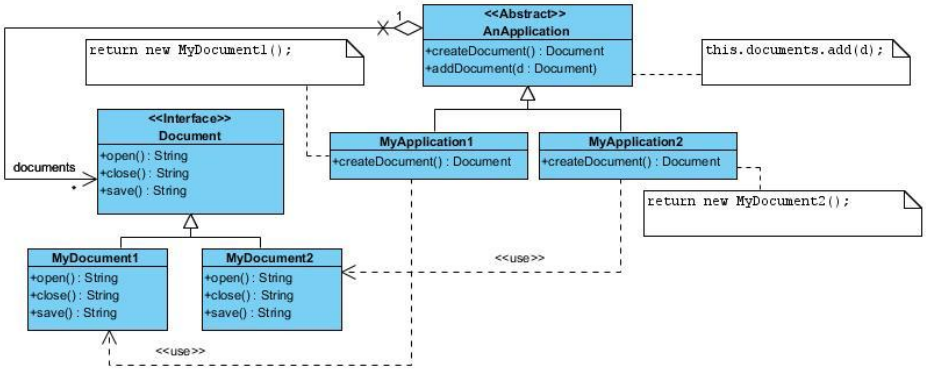

Figura 3.4: *Exemplo de Utilização do Factory Method (adaptada de [Gamma et al., 1995])*

bém que o interface Product tem, de acordo com a Figura 3.3, duas possíveis implementações (ConcreteProduct1 e ConcreteProduct2).

O método factoryMethod() da classe Creator será implementado pelas subclasses de Creator, que são, neste caso, ConcreteCreator1 e ConcreteCreator2, e devolverá respectivamente, um objecto do tipo ConcreteProduct1 ou ConcreteProduct2.

Quanto a algumas variantes associadas a este padrão, importa salientar que a classe Creator pode ser um interface, e, além disto, o método factoryMethod() pode receber como parâmetro um identificador para que se saiba qual o ConcreteProduct a instanciar.

Como exemplo de utilização deste padrão temos uma situação em que se pretende gerir vários documentos, de tipos diferentes. A aplicação sabe que vai ter de instanciar um documento, mas não sabe de que tipo vai ser [Gamma et al., 1995]. Este exemplo pode ser encontrado na Figura 3.4.

Aqui, temos o interface Document a representar um documento, cujo tipo depende das subclasses de Document. Neste caso temos MyDocument1 e MyDocument2. A classe AnApplication contém um conjunto de objectos do tipo Document (polimorfismo), métodos próprios dessa classe, e um método a implementar pelas suas subclasses (createDocument()), que devolve um Document. De acordo com a subclasse (MyApplication1 ou MyApplication2), o documento devolvido será do tipo MyDocument1 ou MyDocument2, respectivamente.

Com a utilização do Factory Method, é também simples oferecer novos produtos, pois basta acrescentar novas implementações para o interface Document, de acordo com o exemplo descrito.

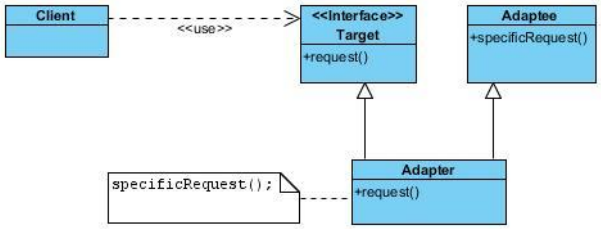

Figura 3.5: *Estrutura do Adapter de Classe (adaptada de [Gamma et al., 1995])*

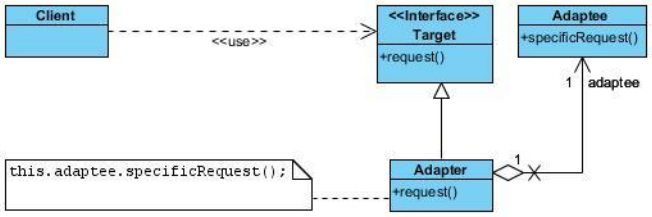

Figura 3.6: *Estrutura do Adapter de Objecto (adaptada de [Gamma et al., 1995])*

3.2 Padrões Estruturais

Ao longo das próximas subsecções vão ser analisados os padrões estruturais Adapter, Composite e Facade, que fazem parte do segundo grupo de padrões de concepção [Gamma et al., 1995].

3.2.1 Adapter

O padrão estrutural Adapter tem por missão converter a interface de uma classe noutra interface que a aplicação cliente espera. Pode ser implementado através de herança múltipla, ou através da composição de objectos [Gamma et al., 1995]. Na Figura 3.5 encontra-se a estrutura do padrão Adapter de Classe, e na Figura 3.6 para o Adapter de Objecto.

Em ambos os casos, vemos que existe um interface com o qual o cliente comunica, que é Target. O método disponibilizado chama-se request(), e será este o único método que o cliente poderá chamar. O problema está na classe Adaptee, que tem um método chamado specificRequest(), diferente do presente em Target. Por isso, é criada a classe Adapter, que implementa o método request() do interface Target, e nessa implementação vai invocar o método specificRequest() de Adaptee, permitindo assim ao cliente a utilização do método definido em Adaptee, através do novo interface.

A diferença nestas duas abordagens encontra-se apenas na classe Adapter, que pode herdar o método request() de Target, e também o método specificRequest() de Adaptee, no caso da herança múltipla (ver Figura 3.5), sendo que no caso da composição de objectos, apenas vai implementar o interface Target(), e vai ter uma variável que é uma instância de Adaptee, sobre a qual

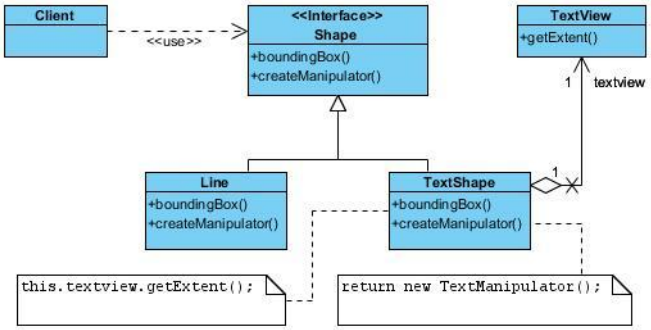

Figura 3.7: *Exemplo de Utilização do Adapter (adaptada de [Gamma et al., 1995])*

vai invocar o método specificRequest(), na definição do método request() (ver Figura 3.6).

Optando pela herança múltipla, como a adaptação é feita a uma classe concreta (Adaptee), sendo a classe adaptadora (Adapter) considerada como subclasse de Adaptee, o processo não é aplicável às subclasses de Adaptee (caso existam). Além disso, a classe Adapter, por ser subclase de Adaptee, pode redefinir algum do seu comportamento [Gamma et al., 1995].

Escolhendo a composição de objectos, um só adaptador (Adapter) pode funcionar com muitos adaptados (Adaptee1, Adaptee2, ...), bastando para isso ter uma instância de cada um deles como suas variáveis de instância. Pode ainda acrescentar funcionalidades ao adaptado. No entanto, fica mais difícil a redefinição de funcionalidades da classe adaptada. Obriga a criar subclasses da classe Adaptee, e a classe Adapter vai ter de passar a utilizar essas subclasses como suas variáveis de instância [Gamma et al., 1995].

Como exemplo de utilização deste padrão, podemos considerar o caso descrito na Figura 3.7 [Gamma et al., 1995].

Vemos então que o interface Shape disponibiliza dois métodos para o cliente invocar, e que esse interface já tem uma subclasse Line que o implementa. O problema estava na classe TextView, mais antiga, e que disponibiliza um método que não faz parte do novo interface, e por isso não pode ser considerada como subclasse de Shape.

Para resolver isto, foi criada a classe TextShape, como subclasse de Shape, que vai servir de adaptador para TextView. Assim, TextShape contém uma instância de TextView, e vai implementar os métodos existentes em Shape, recorrendo aos métodos de TextView. Desta forma, o cliente já consegue usar os métodos de TextView, por intermédio da classe adaptadora TextShape.

3.2.2 Composite

O padrão Composite serve para compor objectos em estruturas em árvore, permitindo assim a representação de hierarquias do tipo "parte de". Desta forma consegue-se tratar da mesma maneira objectos singulares (folhas da

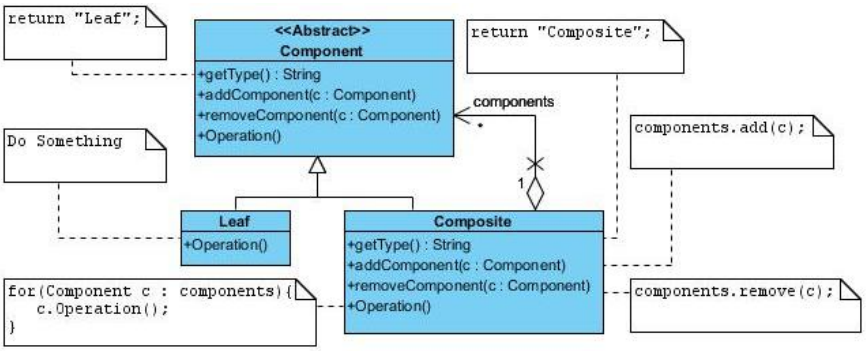

Figura 3.8: *Estrutura do Composite (adaptada de [Gamma et al., 1995])*

árvore), e também composições de objectos (ramos da árvore) [Dong et al., 2000]. Pode-se ainda ter composição recursiva, onde um ramo tem uma ou mais folhas/ramos [Seemann and von Gudenberg, 1998].

Em relação ao cliente, consegue-se que não se aperceba da diferença entre os elementos contidos e os seus contentores, tratando todos os objectos da mesma forma [Gamma et al., 1995].

Como se verifica através da sua observação da Figura 3.8, vemos que existe uma classe Component que representa o tipo dos objectos que vão constituir a estrutura a representar. Essa classe contém métodos que o cliente pode invocar, e não tem necessariamente que ser um interface. Neste caso não o é, pois as suas subclasses herdam todos os métodos, sendo depois redefinidos ao critério de cada uma delas (Leaf apenas redefiniu um desses métodos, ao passo que Composite os redefiniu a todos). No caso de Component ser um interface, aí as suas subclasses eram obrigadas a implementar todos os métodos nele presentes.

A subclasse Leaf representa um objecto singular, ou seja, uma folha da árvore; a classe Composite representa uma composição de objectos (ramos da árvore). Nessa classe existe uma referência para um conjunto de objectos do tipo Component, o que permite que instâncias de Composite contenham vários objectos do tipo Component (que podem na realidade ser do subtipo Leaf ou novamente Composite), permitindo assim a recursividade nesta estrutura. Os métodos redefinidos em Composite lidam com o conjunto de objectos contidos nessa subclasse.

Na Figura 3.9 está representado um exemplo de utilização do Composite, onde temos uma classe Tree que tem métodos para adicionar, remover e mostrar os seus elementos. Esses elementos podem ser instâncias de Leaf ou de Composite, mediante sejam um objecto simples ou uma composição de objectos do tipo Tree, respectivamente. O cliente apenas interage com a classe Tree a partir dos métodos nela apresentados.

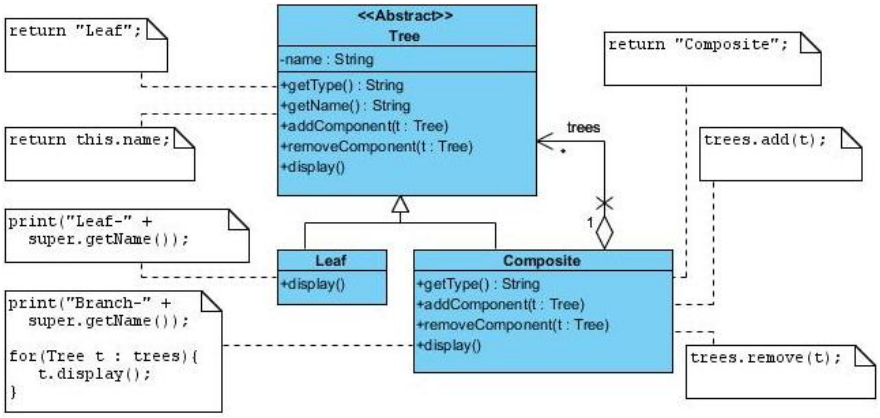

Figura 3.9: *Exemplo de Utilização do Composite (adaptada de [Gamma et al., 1995])*

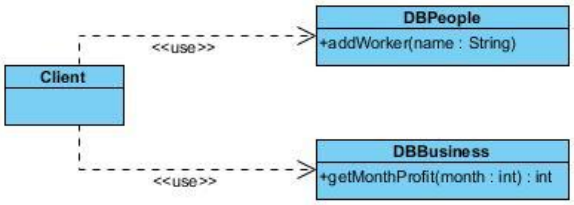

Figura 3.10: *Estrutura do Facade (adaptada de [Gamma et al., 1995])*

3.2.3 Facade

O padrão Facade oferece um interface único para um conjunto de interfaces, ou seja, o cliente em vez de comunicar com várias classes de uma aplicação, vai comunicar apenas com uma, que por sua vez comunica com as restantes. Fica assim definido um interface de alto nível para tornar o sistema mais simples de utilizar [Gamma et al., 1995].

Na Figura 3.10 verifica-se que o cliente tem de conhecer todas as classes que constituem o sistema, para poder interagir com ele. Esta situação é bastante complicada de gerir, obrigando o cliente a saber a que classes se dirigir mediante a operação que pretende invocar. A solução proposta pelo Facade é criar uma classe que disponibilize todos os métodos que o cliente precisa de conhecer para interagir com todo o sistema, ficando a cargo do Facade invocar os correspondentes métodos nas restantes classes.

Como exemplo de utilização deste padrão, temos a classe DBManager que funciona como a classe Facade, que incorpora os métodos presentes nas outras classes que compõem o sistema. Essas classes comunicam com a base de dados da aplicação, mas cada uma delas tem diferentes objectivos: por exemplo, a classe DBPeople trata dos métodos relacionados com os trabalhadores de uma empresa, ao passo que a classe DBBusiness trata das operações relacionadas

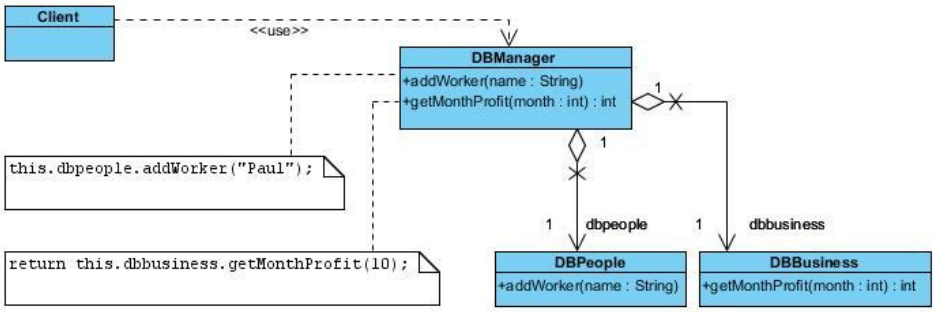

Figura 3.11: *Exemplo de Utilização do Facade (adaptada de [Gamma et al., 1995])*

com a contabilidade dessa empresa. Este exemplo pode ser visto na Figura 3.11.

Caso seja necessário acrescentar ou modificar métodos nas classes já definidas, ou mesmo adicionar novas classes com novos métodos na aplicação, teremos também de fazer a respectiva actualização da classe Facade. Será nestas situações que poderemos afirmar que a manutenção associada ao Facade não é muito simples, mas, porém, a sua utilização continua a ser uma mais-valia para as aplicações/classes cliente, que apenas necessitam de conhecer uma classe do sistema para comunicarem com ele.

3.3 Padrões Comportamentais

Nas subsecções seguintes serão analisados os padrões comportamentais Chain of Responsibility, Iterator e Strategy, que pertencem ao terceiro grupo dos padrões de concepção [Gamma et al., 1995].

3.3.1 Chain of Responsibility

O padrão comportamental Chain of Responsibility permite separar o cliente[1] do objecto que vai processar o seu pedido; apenas se indica ao cliente a porta de entrada para uma cadeia de objectos, e o cliente apenas sabe que o seu pedido vai ser atendido por algum dos objectos dessa cadeia [Vinoski, 2002].

Assim, o pedido do cliente vai passando de objecto em objecto, até que algum o trate (pode haver um ou mais objectos da cadeia capazes de processar cada pedido) [Gamma et al., 1995].

A cadeia de objectos tipicamente será representada por uma lista ligada de objectos, onde cada um conhece o seu sucessor, sendo o sucessor do último elemento o primeiro elemento da lista.

Como se pode verificar na Figura 3.12, existe uma classe Gate que contém um conjunto de objectos do tipo Handler. O cliente comunica com a classe

[1] classe ou aplicação que vai fazer um pedido

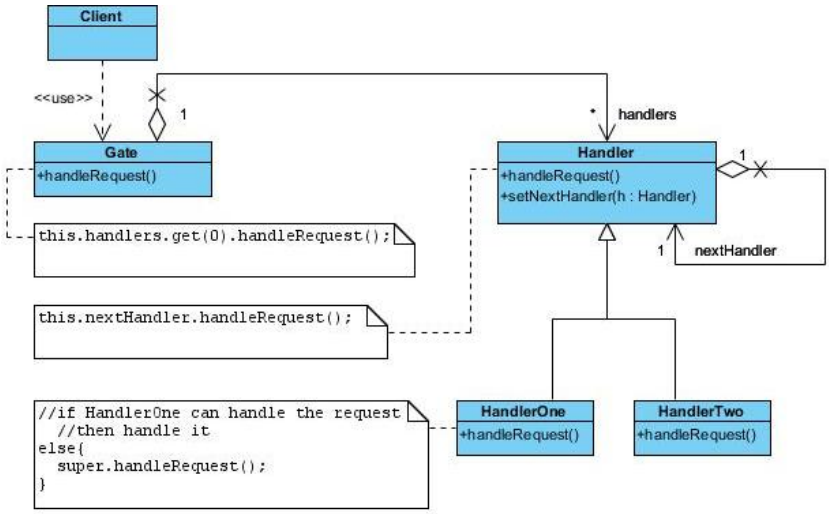

Figura 3.12: *Estrutura do Chain of Responsibility (adaptada de [Gamma et al., 1995])*

Gate, invocando nela o método handleRequest(). A classe Gate vai por sua vez chamar o mesmo método no primeiro elemento do conjunto que possui, para que o pedido seja processado.

A classe Handler é uma superclasse que conhece o seu sucessor, e possui o método handleRequest(), onde faz a mesma invocação, mas no seu sucessor. Temos depois várias sublcasses HandlerOne, HandlerTwo, etc., sendo que cada uma redefine o método handleRequest() herdado, para que cada uma saiba as condições para atender um pedido. Se não o puder atender, então invocará o mesmo método, mas na sua superclasse.

Através da observação do exemplo presente na Figura 3.13, verifica-se que é semelhante à estrutura entretanto descrita, sendo que apenas é feita uma concretização para o método handleRequest(), que passa a receber um número inteiro, e, mediante esse número, será atendido por uma das várias subclasses. Cada uma processa pedidos cujo valor esteja num intervalo de dez números inteiros[2].

3.3.2 Iterator

O padrão Iterator disponibiliza uma forma de aceder aos elementos de um objecto composto, de forma sequencial, sem ser necessário que se saiba qual a representação interna [Dong et al., 2000].

Devemos recorrer a este padrão quando se pretende aceder a um objecto agregador, mas sem expormos a sua implementação, e também quando é necessário suportar múltiplas travessias de um agregador. Por último, este pa-

[2] http://sourcemaking.com/design_patterns

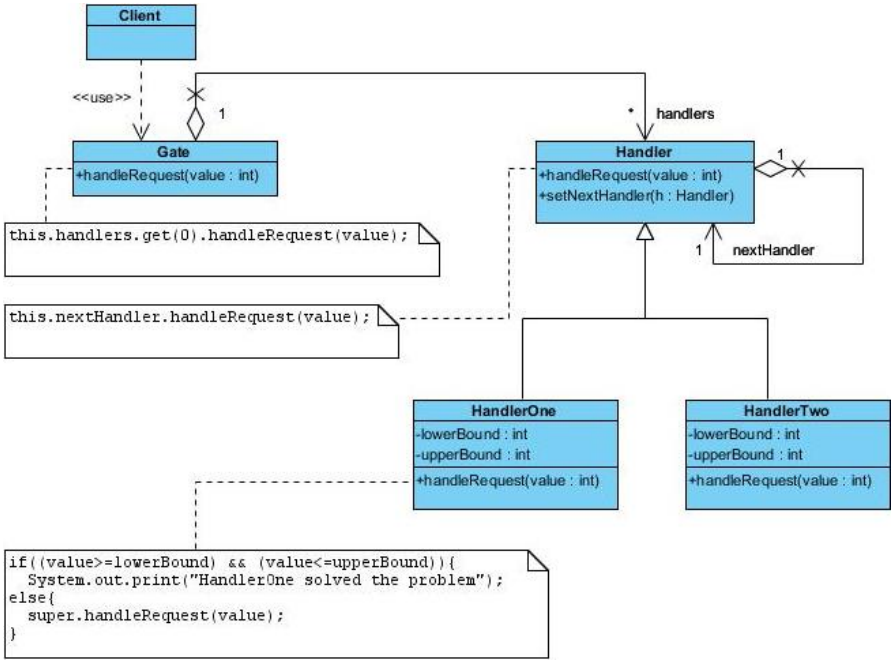

Figura 3.13: *Exemplo de Utilização do Chain of Responsibility (adaptada de [Gamma et al., 1995])*

drão é útil para se oferecer um interface uniforme, que permita efectuar diferentes travessias em diferentes estruturas agregadoras [Gamma et al., 1995].

Através da observação da Figura 3.14, vemos que o interface Aggregate contém o método createItereator(), que devolve uma instância de Iterator (concretizada numa das subclasses) para o cliente. A subclasse ConcreteAgregate implementa o método de instanciar um iterador, e é responsável por encapsular o modo como o conjunto de objectos está representado (por exemplo, um ConcreteAgregate pode utilizar um *array*, e outro ConcreteAgregate uma árvore).

O interface Iterator define os métodos que um iterador deve implementar para ser uma subclasse de Iterator. Cada um dos ConcreteIterator implementa um iterador apropriado para cada um dos vários ConcreteAgregate, ou seja, se ConcreteAgregate utiliza um *array*, então será instanciado um ConcreteIterator que saiba iterar sobre objectos representados por um *array*, por exemplo.

Quanto a um exemplo de utilização do Iterator, podemos considerar a situação representada na Figura 3.15.

Através da sua observação, podemos verificar que existe um conjunto de objectos do tipo TVChannel (contêm o seu nome e posição na grelha de canais), que vai ser implementado por um *array*, na subclasse de AggregateArray de Aggregate.

Temos também uma subclasse de Iterator, que é a classe IteratorArray,

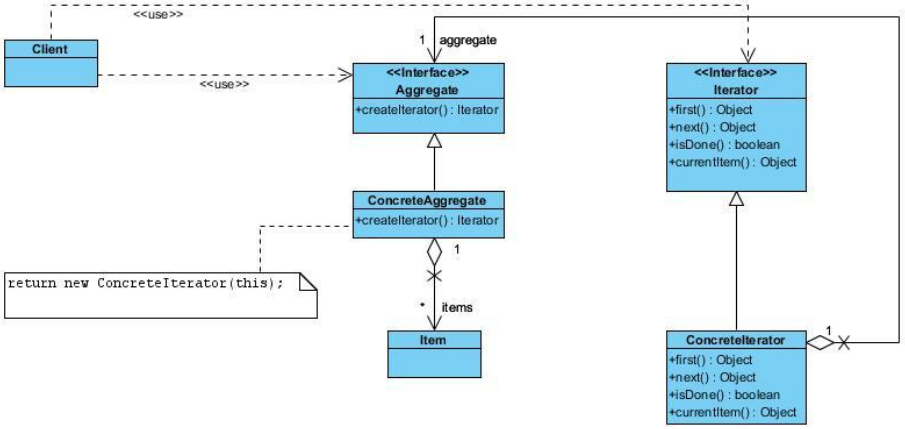

Figura 3.14: *Estrutura do Iterator (adaptada de [Gamma et al., 1995])*

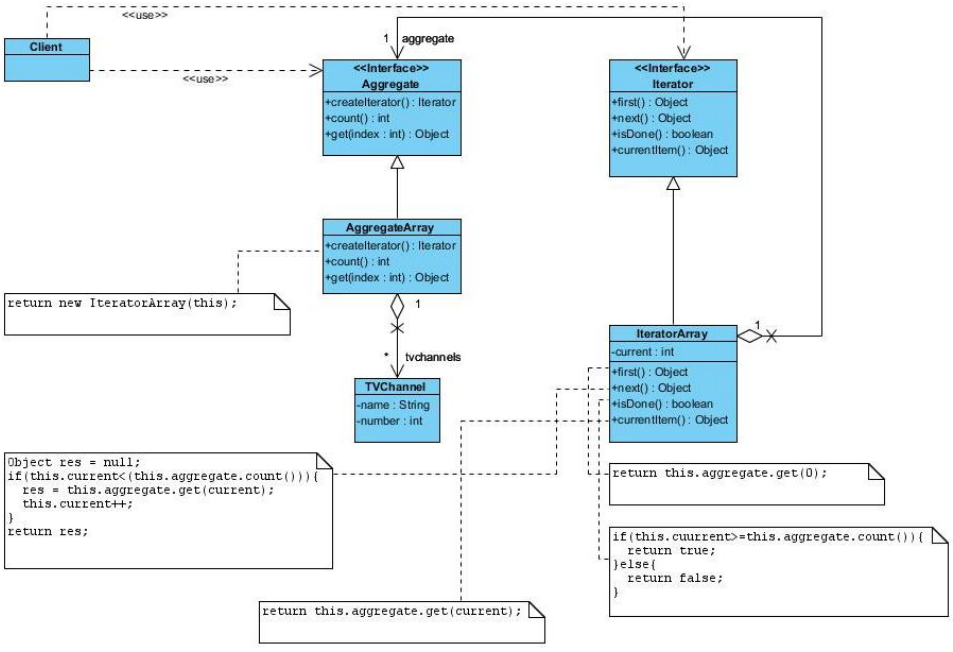

Figura 3.15: *Exemplo de Utilização do Iterator (adaptada de [Gamma et al., 1995])*

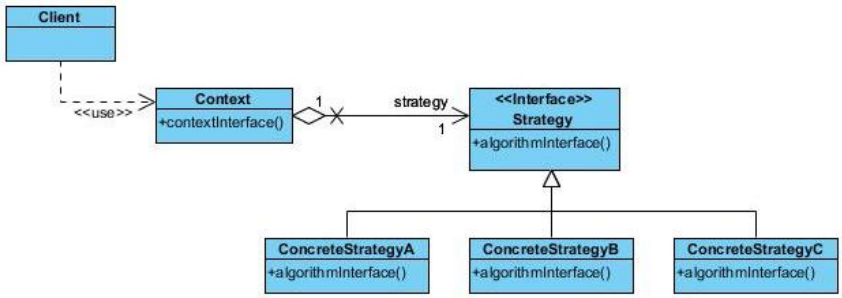

Figura 3.16: *Estrutura do Strategy (adaptada de [Gamma et al., 1995])*

para percorrer o conjunto de canais. O cliente limita-se a interagir com os interfaces Aggregate e Iterator, para ir percorrendo os vários canais do conjunto que possui.

3.3.3 Strategy

O padrão de comportamento Strategy possibilita que se defina uma família de algoritmos, encapsulando cada um deles, de modo a que sejam reutilizáveis. Consegue-se assim que clientes diferentes possam usar algoritmos diferentes [Gamma et al., 1995].

Este padrão deve ser utilizado quando temos muitas classes relacionadas, mas que são diferentes no comportamento, quando pretendemos definir variações de um dado algoritmo, ou mesmo quando um algoritmo usa dados que não queremos que sejam do conhecimento do cliente. Pode também ser utilizado quando uma classe pode ter muitos comportamentos [Gamma et al., 1995]. Será da responsabilidade do cliente decidir qual o algoritmo que pretende que seja utilizado [Aubert and Beugnard, 2001].

Como se pode ver na Figura 3.16, existe um interface Strategy com um método para as subclasses implementarem, dando-lhes assim liberdade para cada uma delas implementar o algoritmo concreto que pretender. A classe Context contém uma variável do tipo Strategy, recebida no seu construtor sobre a forma de algoritmo concreto, e vai ser sobre ela que será invocado o método algorithmInterface(), quando o cliente chamar o método contextInterface().

O cliente apenas comunica com a classe Context. Com esta estrutura elimina-se a opção de utilizar expressões condicionais na escolha do algoritmo, e é simples adicionar um novo algoritmo concreto à família de algoritmos, bastando implementá-lo numa nova sublcasse.

Como exemplo de utilização do Strategy pode-se considerar que o cliente possui um conjunto de números que definem um vector, sendo que pode escolher que esses sejam impressos todos numa só linha, separados por ";", ou então pode querer que surjam um por linha. Temos assim dois algoritmos diferentes para a operação de imprimir um conjunto de números. Este exemplo

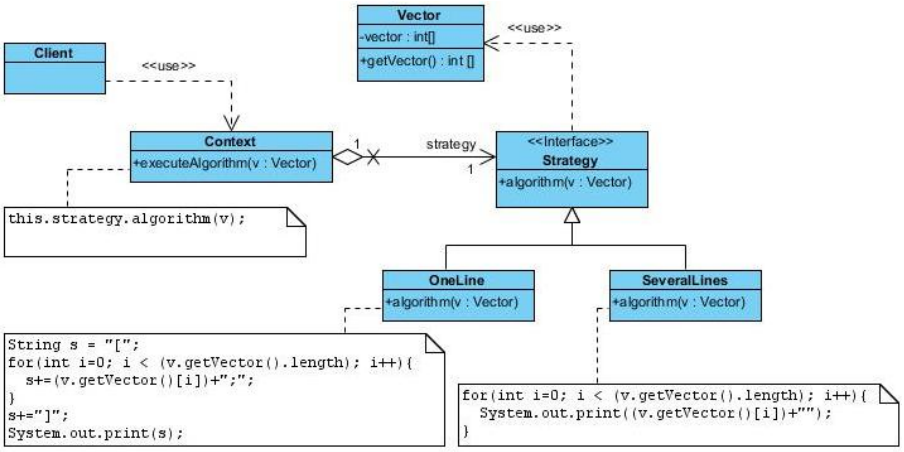

Figura 3.17: *Exemplo de Utilização do Strategy (adaptada de [Gamma et al., 1995])*

encontra-se representado com mais detalhe na Figura 3.17.

3.4 Descrição da Aplicação e Modelo do Domínio

Como já foi visto nas secções anteriores deste capítulo, os padrões de concepção têm a si associada uma implementação concreta, numa linguagem de programação orientada aos objectos. Assim sendo, é possível ter uma aplicação informática que faça a geração automática do código, numa linguagem orientada aos objectos, de um dado padrão de concepção, a partir de alguns dados fornecidos pelo utilizador, como por exemplo o nome de uma classe.

Com esta ideia, durante o desenvolvimento de uma aplicação informática, sempre que necessitarmos de utilizar um determinado padrão de concepção, em vez de estarmos a escrever o código associado, podemos recorrer a um programa que faça isso por nós. Consegue-se assim obter esse código mais rapidamente, e por um processo mais simples, utilizando uma interface gráfica.

Pode-se ainda conseguir gerar um código suficientemente genérico para uma dada situação, sendo que, dessa forma, esse código é capaz de ser reutilizado por várias aplicações informáticas, permitindo criar software a partir de software já existente, em vez de se partir sempre do zero no desenvolvimento de uma nova aplicação informática [Krueger, 1992].

Assim, em seguida encontra-se o Modelo do Domínio para a aplicação que se pretende elaborar, representado na Figura 3.18, onde estão representadas as entidades principais do domínio do problema, além das relações entre elas.

Através da observação da Figura 3.18, podemos verificar que a entidade Utilizador pode escolher uma Colecção de Padrões de Concepção para os quais pretende obter o código gerado pela ferramenta.

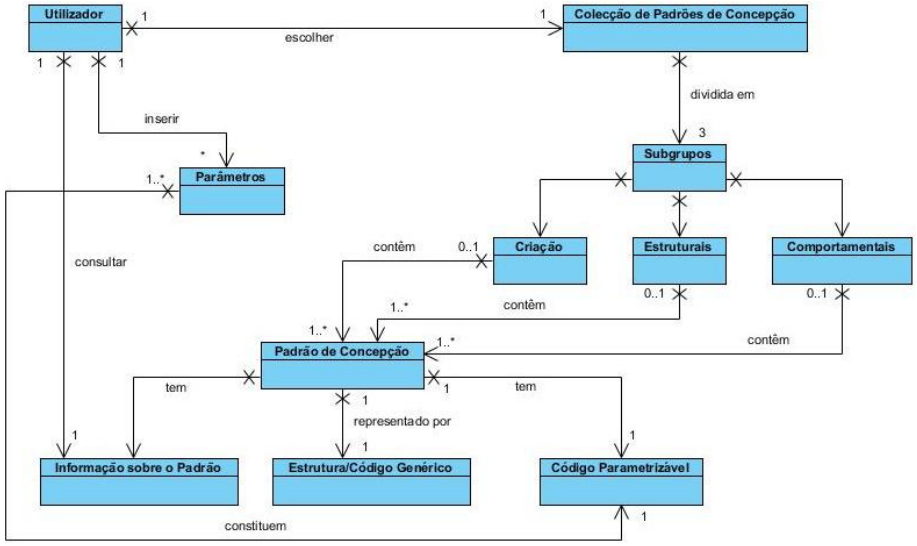

Figura 3.18: *Modelo do Domínio*

Durante o processo de escolha do padrão, o Utilizador pode consultar alguma informação essencial sobre os vários padrões de concepção, representada no Figura 3.18 pela entidade Informação sobre o Padrão, de modo a ter uma ideia mais clara de qual lhe poderá ser mais adequado nesse momento.

Além disto, o Utilizador deve inserir alguns Parâmetros importantes na geração de código, para que o Padrão de Concepção contenha, além da Estrutura/Código Genérico, também código adequado à situação pretendida pelo utilizador (entidade Código Parametrizável).

Sobre os vários Padrões de Concepção que a aplicação vai suportar, importa referir que são os considerados em [Gamma et al., 1995], que se encontram divididos em três Subgrupos, tal como se pode constatar pela observação da Figura 3.18.

Em relação à aplicação a desenvolver, tal como já mencionado, será uma ferramenta capaz de fazer a geração de código para os padrões de concepção, recorrendo para isso à sua estrutura genérica, juntamente com informação proveniente do utilizador, parametrizando assim a geração de código, tornando-a flexível às necessidades particulares do utilizador.

Para que os utilizadores possam interagir com o programa final, devem já possuir algum conhecimento e experiência sobre os vários padrões de concepção, como também dos problemas que pretendem resolver. Porém, é fundamental que a ferramenta a desenvolver disponibilize alguma informação importante e essencial sobre os padrões de concepção, de forma a elucidar os utilizadores quando trabalharem com a aplicação em questão.

Como resultado final, a aplicação deverá devolver um conjunto de ficheiros, contendo o código associado a cada método de cada classe, de acordo

com a estrutura do padrão de concepção escolhido.

Importa ainda dizer que a aplicação a desenvolver deverá ser *cross-platform*, isto é, deverá ser independente de qualquer sistema operativo, podendo assim ser utilizada num maior número de situações.

3.5 Análise de Requisitos

Nesta secção são analisados os requisitos que a ferramenta a desenvolver deve respeitar, de modo a poder ser uma mais-valia para quem estiver a escrever programas numa linguagem de programação orientada aos objectos.

Assim, os requisitos funcionais da aplicação são os seguintes:

- mostrar ao utilizador o conjunto de padrões de concepção, devidamente agrupados, para os quais pode gerar o código correspondente;
- mostrar alguma informação relacionada com cada um dos padrões de concepção. Essa informação é uma breve descrição do padrão, juntamente com o seu diagrama, que representa a sua estrutura;
- receber informação particular do utilizador. Essa informação pode ser constituída, por exemplo, por nomes de classes e nomes de métodos;
- incluir a informação particular recebida na geração do código para o padrão escolhido. Para a aplicação, esses dados inseridos pelo utilizador funcionam como parâmetros a ter em atenção no código final gerado;
- gerar o código associado ao padrão escolhido pelo utilizador, numa linguagem de programação orientada aos objectos, e incluir os parâmetros pretendidos.

Sobre os requisitos não funcionais do programa a desenvolver, importa salientar que deve:

- ser independente da plataforma;
- permitir que, futuramente, possam ser acrescentados novos padrões.

Quanto aos requisitos do utilizador, podem ser considerados os que se seguem:

- escolher um padrão de concepção a partir de um conjunto de padrões;
- consultar a informação relacionada com cada um dos padrões de concepção;
- fornecer à aplicação alguns parâmetros particulares (por exemplo, nomes de classes e nomes de métodos);
- obter o código do padrão escolhido, de acordo com os parâmetros indicados.

Capítulo 4

A aplicação PatGer

Ao longo deste capítulo será descrita a aplicação desenvolvida, ou seja, todo o programa responsável pela geração automática de código para os padrões de concepção [Gamma et al., 1995]. O nome escolhido para esta aplicação é *PatGer - Gerador Automático de Código para Padrões de Concepção*, podendo ser simplesmente chamado PatGer.

Na primeira secção será referida a especificação efectuada, quer ao nível da interface do PatGer, quer ao nível da sua lógica de negócio, apresentando, para o efeito, uma série de exemplos de esboços da interface, de diagramas de sequência e de diagramas de classes. Ao nível da interface serão apresentados vários esboços, relacionados com os componentes presentes na interface do sistema, e também alguns diagramas que ilustrem o que se passará após o utilizador clicar numa determinada opção. Já ao nível da lógica de negócio poderão ser visualizados vários packages, contendo cada um deles uma série de classes e métodos, devidamente agrupados de acordo com as funcionalidades que desempenham na implementação do PatGer.

Na segunda secção será dada importância ao processo de desenvolvimento do PatGer, após concluída e analisada a etapa de especificação. Serão divulgadas algumas decisões de implementação, importantes para a solução final, juntamente com a tecnologia adoptada. Haverá ainda espaço para mostrar um breve exemplo de utilização da aplicação elaborada, e também para indicar o que deverá ser efectuado para acrescentar novas funcionalidades ao programa final.

4.1 Especificação do PatGer

Esta secção encontra-se dividida em duas subsecções, sendo que na primeira será abordada a especificação do PatGer ao nível da interface, e na segunda ao nível da lógica de negócio.

Em relação à especificação da interface, importa dizer que serão analisados vários exemplos de esboços, os quais virão a fazer parte da solução

final. Além destes exemplos, serão também apresentados alguns diagramas de sequência, de modo a elucidar o que se passará com os dados introduzidos na interface, até que sejam utilizados pelo programa para a geração de código para os padrões de concepção.

No que diz respeito à especificação da lógica de negócio da aplicação, há que referir que a sua modelação foi baseada em diagramas de classes, que serão também apresentados e analisados, e onde se podem verificar algumas das várias classes que compõem o programa final, além dos seus métodos e relacionamentos.

4.1.1 Interface

Nesta subsecção vai ser descrita a modelação efectuada para a interface da aplicação a desenvolver. Tal como já anteriormente mencionado, a interface da aplicação será uma interface gráfica, contendo portanto caixas de texto para o utilizador introduzir os dados essenciais à geração de código para um dado padrão de concepção, além de botões para poder interagir com o programa.

Sobre a modelação efectuada para a interface do PatGer, interessa dizer que, inicialmente, foram desenhados os esboços gráficos, e, em seguida, foram desenhados alguns diagramas de funcionamento dos componentes da interface, para se saber como o programa trataria os dados introduzidos pelo utilizador, e o que deveria ser feito ao clicar em cada botão.

Em seguida serão apresentados e analisados alguns dos esboços elaborados para a interface da aplicação, podendo os restantes ser consultados no Anexo D. Uma vez que os esboços desenhados são, de certa forma, bastante semelhantes entre si, variando apenas nos dados que o utilizador tem de inserir para obter o código para diferentes padrões de concepção, não se justifica incluir todos os esboços na presente dissertação.

Os esboços considerados ao longo desta subsecção dizem respeito aos padrões de concepção Builder, Object Pool, Composite, Facade, Chain of Responsibility e Memento. Será efectuada uma análise mais detalhada ao esboço do padrão Builder, sendo que para os restantes padrões já não se irá entrar em tanto detalhe, devido às semelhanças existentes entre os esboços.

Terminada a análise dos referidos esboços, serão apresentados dois diagramas de sequência relacionados com os componentes da interface, de forma a mostrar como o programa processa os pedidos do utilizador, devolvendo o resultado pretendido. Os restantes diagramas, associados aos esboços que serão analisados em seguida, podem ser encontrados no Anexo E.

Builder O primeiro esboço a analisar é sobre o padrão Builder [Gamma et al., 1995], pertencente ao grupo dos padrões de criação. Vai ser analisado em detalhe para que se possa compreender qual o papel de cada um dos botões e caixas de texto nele presentes.

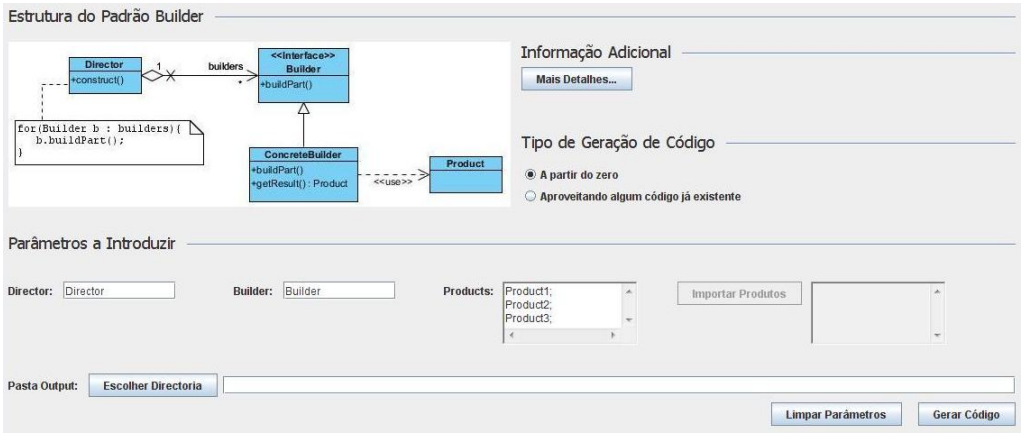

Figura 4.1: *Esboço da interface do Builder (a)*

Figura 4.2: *Esboço do botão "Mais Detalhes..."*

Através da observação da Figura 4.1 podemos verificar a existência de uma ilustração da estrutura associada ao padrão Builder, de modo a ajudar o utilizador a perceber o conjunto de ficheiros que o compõem, e que serão gerados pela aplicação. Existe também um botão "Mais Detalhes...", destacado na Figura 4.2, que deverá abrir uma outra janela, contendo alguma informação genérica relacionada com este padrão, para melhor esclarecer o papel de cada componente no Builder.

Além desta informação presente na interface, há a opção para o utilizador escolher entre obter o código "A partir do zero", ou "Aproveitando algum código já existente", como se verifica pela observação da Figura 4.3.

Caso a opção do utilizador seja a primeira, tal como representado na Figura 4.3, então deverá introduzir um conjunto de parâmetros, através da interface, como se pode verificar na Figura 4.4. Assim, a aplicação fará a geração de cada uma das classes do Builder, com os parâmetros fornecidos pelo utilizador. Neste caso concreto, vai ser gerada a classe Director, o interface Builder, e depois várias classes ConcreteBuilder e

Figura 4.3: *Esboço para a escolha do tipo de geração de código*

Figura 4.4: *Esboço para os parâmetros a introduzir*

Figura 4.5: *Esboço do botão "Escolher Directoria"*

Product, mediante a indicação do utilizador na caixa de texto Products (para cada Product presente nesta caixa de texto, será gerado um Product e o correspondente ConcreteBuilder). O botão "Escolher Directoria", salientado na Figura 4.5, deverá ser utilizado, neste caso, para escolher a directoria onde deverão ser colocados os ficheiros gerados pelo PatGer.

Se a opção do utilizador for a segunda, de acordo com o sugerido na Figura 4.6, isto é, "Aproveitando algum código já existente", então vai ter de introduzir os parâmetros respeitantes ao nome da classe Director e do interface Builder. Além disto, nesta segunda opção o botão "Importar Produtos" ficará activo, servindo para escolher vários Produtos já existentes (já entretanto criados pelo utilizador, e que serão incorporados na estrutura do Builder).

De realçar que os ficheiros que podem ser importados pela aplicação são ficheiros *.class*, ou seja, ficheiros resultantes de código Java já compilado. São estes os ficheiros que o PatGer é capaz de processar. Opcionalmente, o utilizador poderá também introduzir na caixa de texto Products alguns produtos, e a aplicação tratará de gerar os respectivos ficheiros.

Assim, para a escolha "Aproveitando algum código já existente", o PatGer vai gerar a classe Director, o interface Builder, e ainda um ConcreteBuilder associado a cada um dos produtos importados, e também um ConcreteBuilder relativo aos produtos colocados na caixa de texto Products. O botão "Escolher Directoria" serve para seleccionar o local para o programa colocar os ficheiros gerados.

Para terminar, resta dizer que "Limpar Parâmetros" vai apenas apagar os dados introduzidos pelo utilizador em todas as caixas de texto presentes na interface para o Builder, e "Gerar Código" vai, numa primeira fase, fazer a recolha e processamento dos dados inseridos na interface, e, em seguida, gerar os ficheiros que compõem a estrutura do Builder, de acordo com os dados retirados da interface.

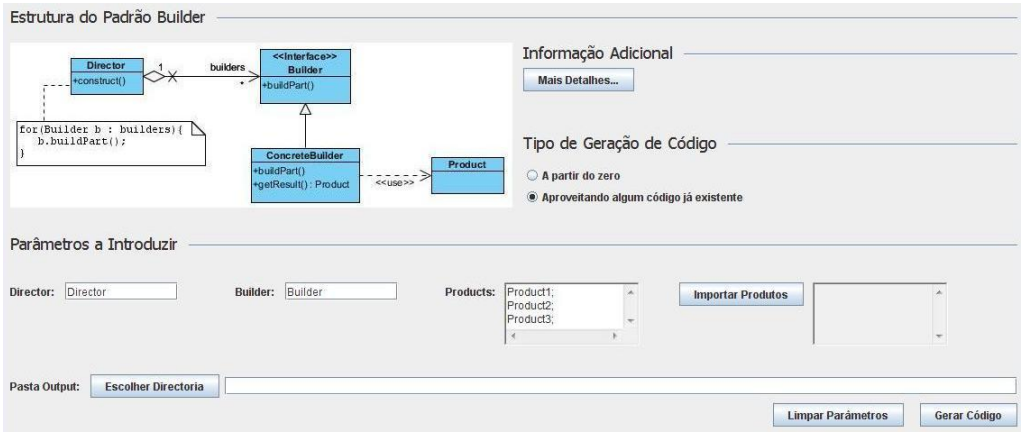

Figura 4.6: *Esboço da interface do Builder (b)*

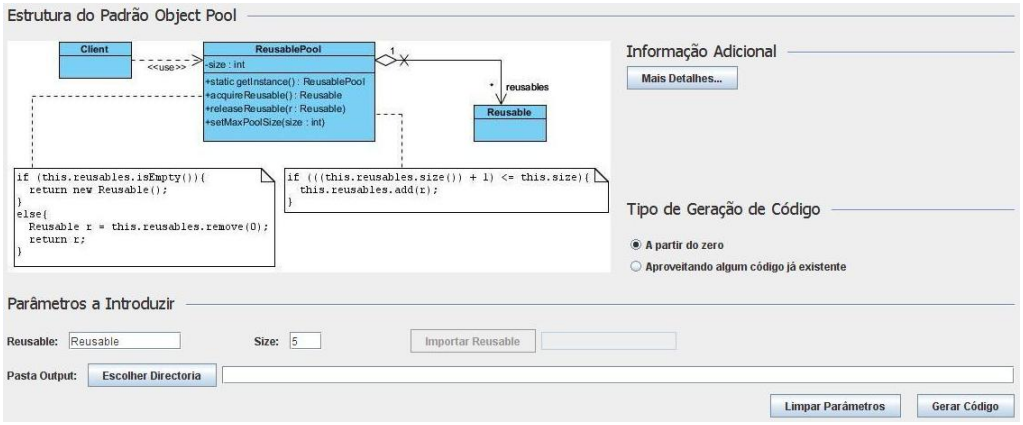

Figura 4.7: *Esboço da interface do Object Pool*

Object Pool O próximo esboço é sobre o padrão de criação Object Pool[1], que embora não tenha sido catalogado em [Gamma et al., 1995], pode ser encontrado em aplicações actuais, nomeadamente em situações onde é necessário utilizar uma base de dados. Este padrão serve então para, por exemplo, gerir um conjunto de ligações à referida base de dados, evitando que sempre que haja necessidade de efectuar um novo pedido à base de dados, tenha de ser estabelecida uma nova conexão.

Através da observação da Figura 4.7 pode-se verificar a presença da estrutura deste padrão, tendo por propósito elucidar o utilizador no momento de interagir com o PatGer. Nessa ilustração encontram-se também pequenos "pedaços" de código, relativos aos métodos da classe ReusablePool, que é, neste caso, a classe chave do padrão Object Pool. Uma vez mais, tal como no esboço anterior, existe um botão "Mais

[1] http://sourcemaking.com/design_patterns

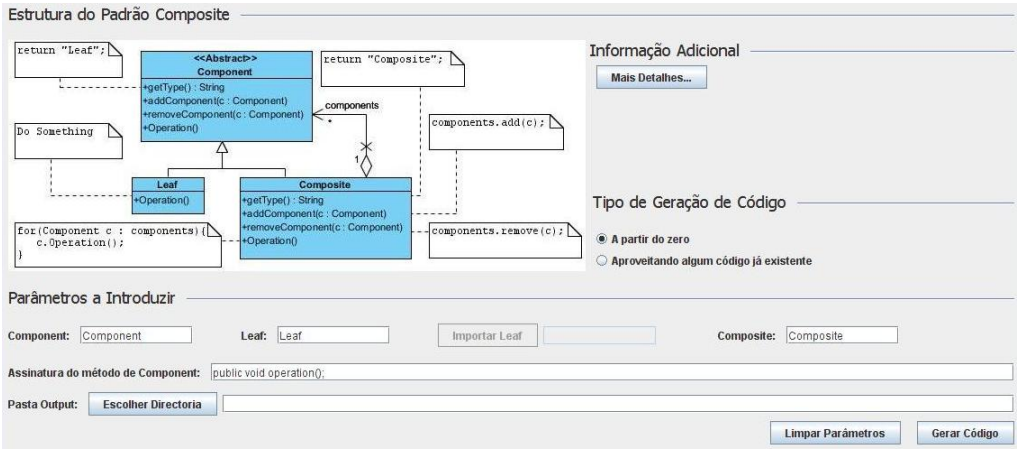

Figura 4.8: *Esboço da interface do Composite*

Detalhes..." para abrir uma nova janela com informação adicional sobre este padrão.

É também possível o utilizador escolher entre gerar o código para o Object Pool "A partir do zero", ou "Aproveitando algum código já existente". Caso opte pela primeira opção, deverá preencher os parâmetros solicitados, e escolher a directoria para o programa escrever os ficheiros gerados; caso opte pela segunda opção, então apenas deverá indicar o tamanho máximo da *pool*, e escolher o ficheiro que a aplicação vai considerar como sendo o objecto Reusable, clicando em "Importar Reusable", que ficará activo ao seleccionar a segunda opção.

Composite O padrão que se segue pertence aos padrões estruturais, segundo a classificação proposta em [Gamma et al., 1995]. Trata-se de um padrão que deve ser utilizado para lidar com estruturas em árvore, de forma recursiva.

De acordo com o esboço representado na Figura 4.8, pode-se verificar que é em tudo semelhante aos dois esboços já entretanto apresentados. A única diferença prende-se com o facto de, neste caso concreto, o utilizador poder definir a assinatura do método Operation da classe Component. O PatGer tratará de respeitar essa informação proveniente do utilizador, incluindo-a na geração de código efectuada.

Facade O esboço que se segue diz respeito ao padrão estrutural Facade, responsável por apresentar os métodos de um conjunto de classes de um sistema numa única classe, servindo assim esta classe de ponto de comunicação único com uma aplicação cliente [Gamma et al., 1995].

Como se observa na Figura 4.9, na parte correspondente à estrutura associada ao Facade, existem, neste exemplo, duas classes (ClassA e

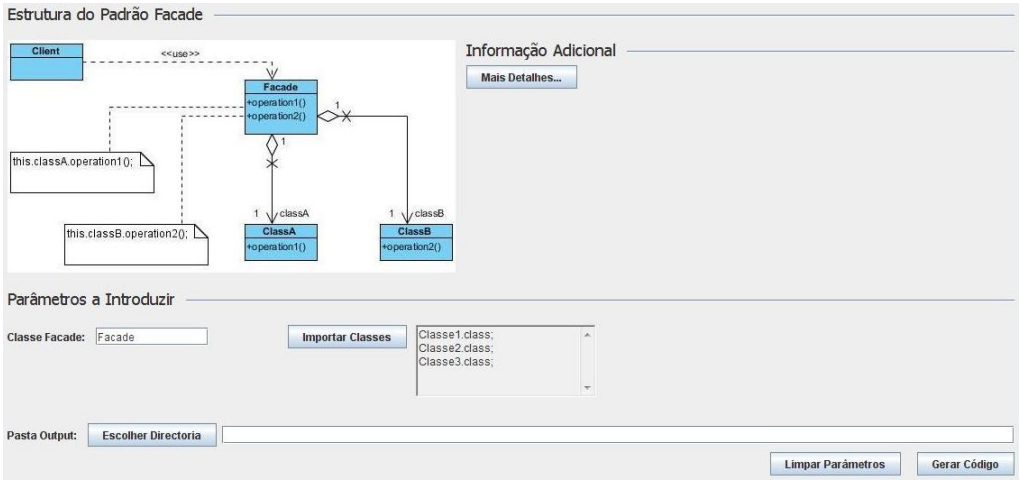

Figura 4.9: *Esboço da interface do Facade*

ClassB), cujos métodos vão fazer parte da classe Facade. Claro que o número de classes a considerar pela aplicação não é relevante, uma vez que o programa apenas vai, em cada uma delas, extrair as assinaturas dos métodos para gerar a classe Facade pretendida.

O utilizador apenas deverá indicar um nome para a classe Facade, seleccionar as classes para as quais pretende aplicar este padrão estrutural, através do botão "Importar Classes", e, por fim, escolher a directoria para onde deverá ser escrito o ficheiro resultante.

Para este padrão não existe a opção de gerar o código "A partir do zero", como nos outros casos já entretanto apresentados, pois este é um padrão que deve ser aplicado apenas quando já existem classes desenvolvidas, que serão então incorporadas no Facade.

Chain of Responsibility Na Figura 4.10 encontra-se representado o esboço para a interface do padrão Chain of Responsibility, pertencente aos padrões comportamentais, de acordo com a classificação proposta em [Gamma et al., 1995]. Este padrão deve ser utilizado quando se pretende efectuar um pedido a um conjunto de objectos, não se sabendo qual o objecto do conjunto que o irá processar.

O esboço apresentado para este padrão é muito semelhante aos esboços já analisados nesta subsecção da dissertação. Assim sendo, contém uma representação da estrutura associada ao Chain of Responsibility, e um botão para mostrar uma janela com informação auxiliar, relativa ao papel de cada entidade presente na estrutura deste padrão de comportamento.

Além disso, são pedidos vários parâmetros para o programa poder efectuar a geração dos ficheiros de código deste padrão. Se for para efec-

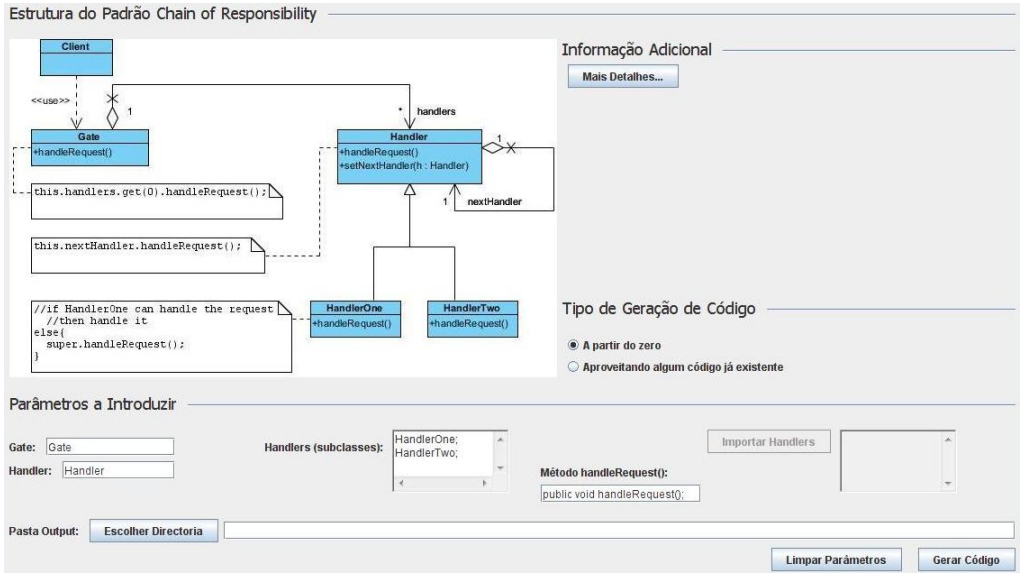

Figura 4.10: *Esboço da interface do Chain of Responsibility*

tuar a geração "A partir do zero", interessa salientar que é o utilizador a definir a assinatura do método handleRequest, e ainda as várias subclasses (Handlers) desejadas. Se a opção for a de aproveitar código já existente, então o PatGer vai receber as classes que farão o papel de Handlers (subclasses), através do botão "Importar Handlers", contendo já uma implementação de handleRequest, que a aplicação extrairá e colocará na classe Handler. Poderão também ser gerados os ficheiros para novos Handlers, presentes na caixa de texto Handlers (subclasses).

Memento É um padrão de comportamento que deve ser utilizado quando se pretende extrair o estado de um objecto, e guardá-lo noutro objecto, para que, mais tarde, esse estado possa ser reposto no objecto que o originou [Gamma et al., 1995]. Na Figura 4.11 pode ser visto o esboço da interface associada ao Memento.

Este exemplo é em tudo parecido com os exemplos anteriormente descritos, tendo também uma ilustração da estrutura associada ao padrão, além de pedir ao utilizador que insira um conjunto de parâmetros, necessários para a correcta geração dos ficheiros com o respectivo código.

A novidade presente neste esboço é a existência de uma caixa para o utilizador inserir as variáveis de instância, consideradas como o estado do objecto que interessa guardar. Assim, o PatGer vai encapsular essas variáveis numa classe State, que estará presente como uma variável de instância de Originator, e poderá ser extraída e guardada, na classe Caretaker, quando necessário.

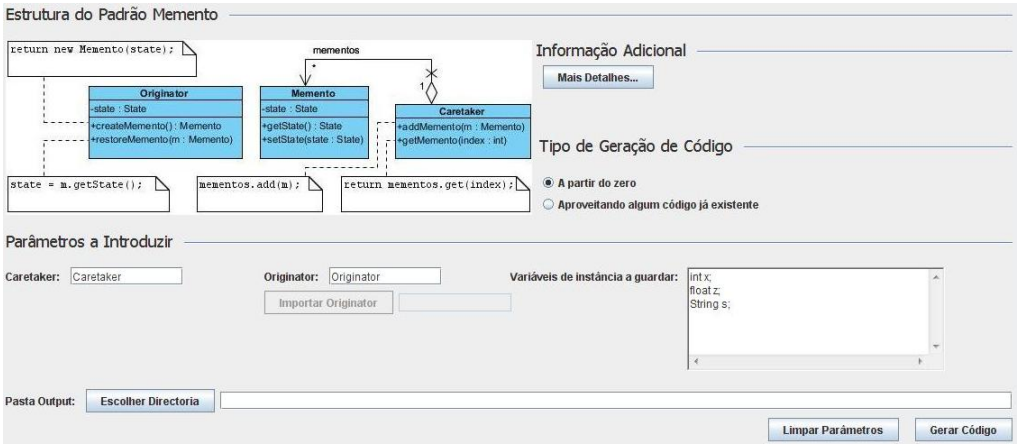

Figura 4.11: *Esboço da interface do Memento*

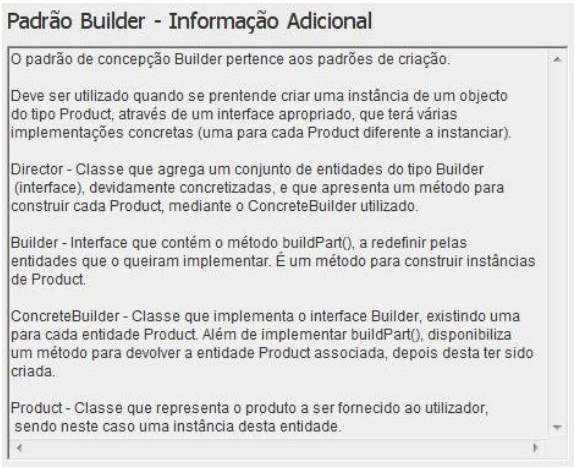

Figura 4.12: *Esboço da informação adicional do Builder*

O próximo esboço a apresentar está relacionado com o botão "Mais Detalhes...", presente nos esboços já descritos e analisados. Ao clicar neste botão, a aplicação vai abrir uma nova janela, contendo alguma informação adicional relativa ao padrão de concepção seleccionado, nomeadamente uma breve descrição da sua funcionalidade, e do papel de cada entidade na sua estrutura.

Para ilustrar esta ideia apenas é necessário apresentar um exemplo concreto, uma vez que todos os outros são muito parecidos, variando apenas no texto escrito para cada um deles. Na Figura 4.12 encontra-se a informação adicional para o padrão de concepção Builder, e, como se pode verificar, inicialmente é feita uma breve descrição da sua funcionalidade, e depois é dito o papel de cada entidade que o constitui.

Terminada a apresentação e descrição de alguns esboços para a interface da aplicação a desenvolver, é também interessante mostrar como é que o

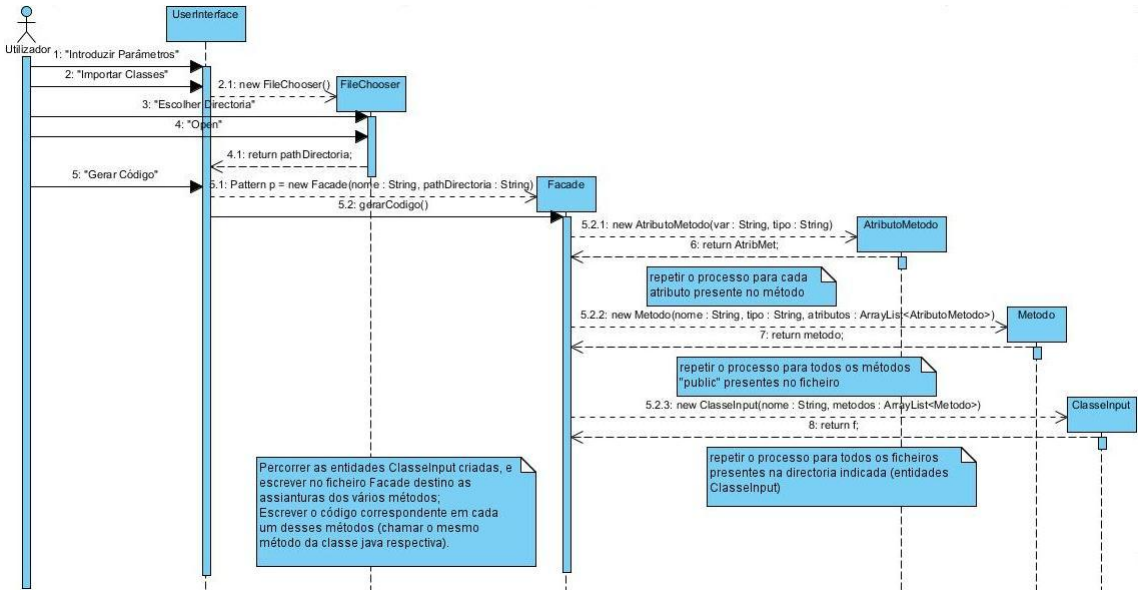

Figura 4.13: *Diagrama de Sequência do Facade*

PatGer vai lidar com os dados introduzidos pelo utilizador, até à geração do respectivo código.

Desta forma, foram desenvolvidos vários diagramas de sequência para clarificar o comportamento do programa, após o utilizador mandar gerar o código para um determinado padrão de concepção, e também após escolher visualizar a informação adicional de um dado padrão.

Dado que os diagramas desenvolvidos são semelhantes entre si, vai ser apenas descrito, de forma detalhada, um diagrama de sequência relativo à geração de código, e outro relacionado com a visualização de informação adicional, salientando a lógica subjacente ao processamento dos dados do utilizador até à geração do código. Os restantes diagramas de sequência directamente associados aos esboços já analisados na presente dissertação podem ser encontrados no Anexo E, tal como já mencionado no início desta subsecção.

O diagrama de sequência presente na Figura 4.13 diz respeito ao padrão estrutural Facade, quando o utilizador clica no botão "Gerar Código".

Como se pode verificar pela observação do diagrama, vemos que o utilizador vai poder comunicar directamente com a entidade UserInterface, que representa a janela onde o PatGer está a correr. Em primeiro lugar, o utilizador vai introduzir os parâmetros necessários para a geração de código do Facade, e, em seguida, vai importar as classes cujos métodos devem ser extraídos para se obter o Facade pretendido.

Assim, ao clicar em "Importar Classes", a classe UserInterface lança uma pequena janela para o utilizador proceder à procura e selecção da directo-

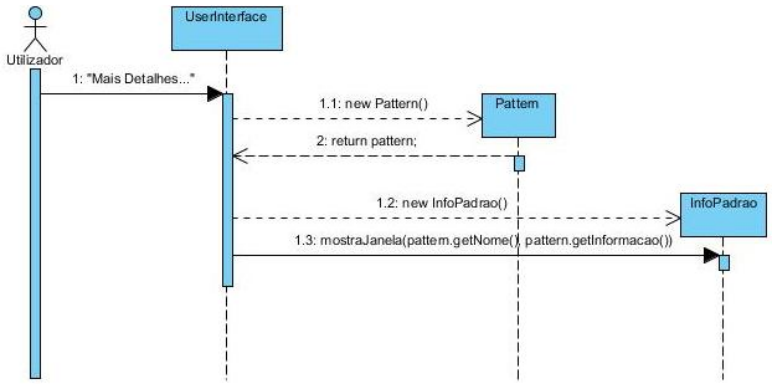

Figura 4.14: *Diagrama de Sequência do botão "Mais Detahes..."*

ria onde se encontram as classes que farão parte do Facade. Essa pequena janela foi representada no diagrama da Figura 4.13 como sendo a entidade FileChooser. Após clicar em "Open", será retornado para UserInterface o *path* da respectiva directoria.

Em seguida, o utilizador pode então clicar em "Gerar Código", e a classe UserInterface vai criar uma instância de Facade, que é uma classe que herda da superclasse Pattern (na subsecção 4.1.2 podem ser encontrados mais detalhes relativos à lógica de negócio subjacente à aplicação a desenvolver). Será depois chamado o método gerarCodigo() na instância de Facade entretanto obtida.

Quanto ao comportamento associado a este método, como se verifica na Figura 4.13, para cada ficheiro presente na directoria especificada, será criada uma instância de ClasseInput, contendo um conjunto de classes Metodo (uma para cada método presente na classe sobre a qual o programa está a iterar), que por sua vez contém, cada uma, várias instâncias de AtributoMetodo, uma para cada atributo presente no método sobre o qual o programa está a iterar. No final deste processo, a classe Facade vai conter um conjunto de instâncias de ClasseInput, que serão percorridas para ser gerado o código para a obtenção do padrão Facade.

Terminada a descrição do diagrama de sequência exemplificativo do processo de geração de código para um padrão de concepção, vai ser analisado agora o diagrama de sequência relacionado com a visualização de informação adicional, para um determinado padrão de concepção.

Assim sendo, na Figura 4.14 encontra-se um diagrama de sequência genérico para o caso em que o utilizador clica sobre o botão "Mais Detalhes..." na interface do PatGer e, como se pode verificar no diagrama da Figura 4.14, inicialmente o utilizador vai clicar em "Mais Detalhes...", e a entidade UserInterface vai processar o pedido do cliente. Dito isto, vai ser criada uma instância de Pattern (supertipo), de acordo com o padrão de concepção em causa.

Em seguida, vai ser instanciado um objecto InfoPadrao, com a instância de Pattern entretanto criada. De referir que a instância de InfoPadrao será uma janela, podendo assim ser mostrada ao utilizador, contendo um pequeno texto com a informação adicional associada ao padrão de concepção pretendido.

4.1.2 Lógica de Negócio

Ao longo desta subsecção vai ser descrita e analisada a modelação feita para a lógica de negócio do PatGer. Da lógica de negócio de uma aplicação de software fazem parte as classes, os relacionamentos entre elas, e também os métodos e variáveis de instância nelas presentes. Além disto, para tornar mais perceptível a organização das classes que constituem o sistema a desenvolver, estas serão agrupadas em packages, ou seja, conjuntos de classes com funcionalidades semelhantes.

Desta forma, pode-se dizer que a arquitectura lógica do PatGer se trata de uma arquitectura de duas camadas, separando assim a camada da interface da camada de negócio.

Na camada da interface estarão as classes que tratam de implementar a interface do programa, ou seja, a parte do programa "visível" para o utilizador, servindo de ponto de entrada no sistema. Estarão também as classes que tratam do processamento dos dados introduzidos pelo utilizador. Desta camada faz parte o package UserInterface, como se poderá constatar mais à frente.

Da camada de negócio farão parte as classes responsáveis por implementar toda a lógica de negócio presente na aplicação, que neste caso se trata de, a partir de dados fornecidos pelo utilizador, e tratados pelas classes da camada da interface, gerar os ficheiros que compõem a estrutura do padrão de concepção seleccionado pelo utilizador. Nesta camada está presente o package ProgramaPatterns, entre outros, como se verá mais adiante.

Sobre os packages especificados há que referir que, em primeiro lugar, foi especificado um package para conter as classes responsáveis pela interface da aplicação (UserInterface), e outro package para agrupar as classes utilizadas na geração de código de qualquer um dos padrões de concepção (ProgramaPatterns), como por exemplo a classe que trata de escrever num ficheiro.

Também em relação à camada de negócio do PatGer, interessa salientar a existência de vários outros packages, nomeadamente um para cada um dos vários padrões de concepção suportados pelo sistema.

Para explicar melhor os vários packages e classes desenvolvidos na especificação da lógica de negócio do programa, em seguida serão analisados, com algum detalhe, os packages UserInterface e ProgramaPatterns, e logo depois três packages relativos a três dos padrões de concepção suportados pelo PatGer (Builder, Facade e Memento). No Anexo F podem ser encontrados mais

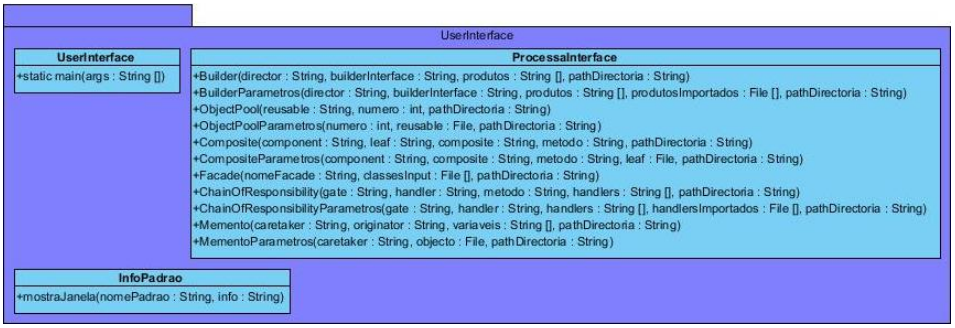

Figura 4.15: *Package UserInterface*

alguns diagramas, relativos a mais alguns packages do PatGer. Não serão todos incluídos em anexo pois são algo semelhantes entre si.

UserInterface Este é o package que contém as classes responsáveis pela interface do sistema, ou seja, as classes que representam as janelas apresentadas ao utilizador, e também que tratam de processar os pedidos do utilizador, isto é, capturam os dados por ele introduzidos, passando-os depois para a camada de negócio.

Como se verifica através da observação da Figura 4.15, há três classes presentes no package UserInterface: a classe UserInterface, a classe InfoPadrao, e por fim a classe ProcessaInterface.

A classe UserInterface representa a janela principal do PatGer, que permite a interacção do utilizador com a aplicação; a classe InfoPadrao representa a janela apresentada quando há a necessidade de mostrar a informação adicional sobre um determinado padrão de concepção.

A classe ProcessaInterface é utilizada para processar os dados inseridos pelo utilizador, e tratar, em seguida, da geração de código para o padrão de concepção em questão. Como se pode ver na Figura 4.15, nessa classe estão os métodos para a geração de código dos padrões apresentados durante a subsecção 4.1.1.

ProgramaPatterns Este package é constituído por várias classes que são utilizadas, de forma recorrente, no processo de geração de código para qualquer um dos padrões de concepção. São classes que fazem parte da camada de negócio da aplicação.

Na Figura 4.16 pode ser visto o package ProgramaPatterns, juntamente com todas as classes nele presentes, além dos respectivos métodos. Assim sendo, importa para já abordar a classe Pattern, que é uma classe que contém um conjunto de métodos comuns a todas as classes que representam um qualquer padrão de concepção. Desta forma, como se vai poder ver mais à frente, cada padrão de concepção está representado

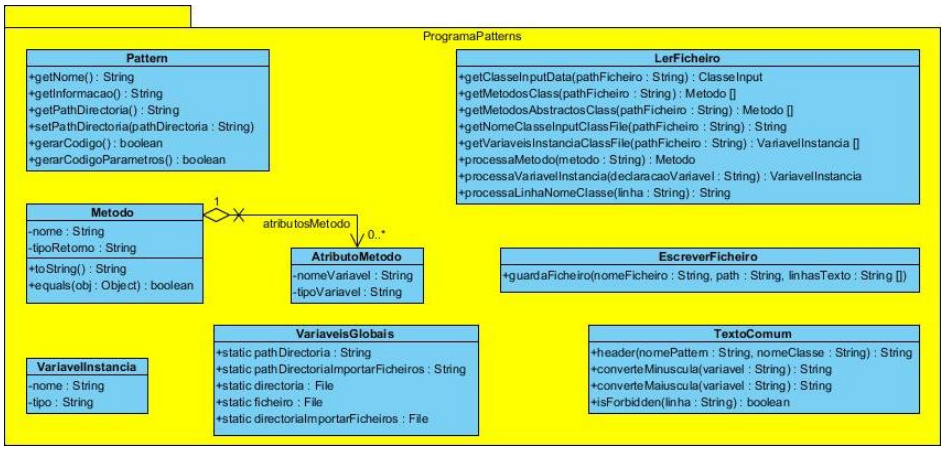

Figura 4.16: *Package ProgramaPatterns*

por um package, e nesse package existe sempre uma classe que vai herdar desta classe Pattern, redefinindo os métodos de acordo com o padrão em causa.

Existem também as classes Metodo e AtributoMetodo, que são as entidades que representam um método da linguagem Java nesta aplicação. É uma especificação bastante simplificada em relação à definição de classe, atributo e método, em UML, como se pode constatar pela observação da Figura 4.17.

De acordo com este metamodelo, a entidade Class é subclasse de Classifier, contendo esta vários objectos Feature e Parameter. Feature representa funcionalidades associadas à classe, podendo ser StructuralFeature e/ou BehavioralFeature. Caso seja StructuralFeature, pode ser considerada como um atributo (entidade Attribute), e caso seja BehavioralFeature poderá ser do subtipo Operation ou Method, sendo portanto um método constituído por entidades Operation e Method, tal como representado na Figura 4.17.

Voltando agora às classes Metodo e AtributoMetodo, presentes na Figura 4.16, em relação aos atributos presentes em Metodo, temos um para o nome do método, outro para o seu tipo de retorno, e ainda a hipótese de haver várias entidades AtributoMetodo para representar os respectivos atributos de um método, ou seja, os seus parâmetros (variáveis com nome e tipo). De forma semelhante foi especificada a classe VariavelInstancia, uma classe bastante simples para representar uma variável de instância, com o seu nome e tipo.

Quanto à classe VariaveisGlobais, há que dizer que é uma classe utilizada pelo PatGer para gerir a informação relacionada com a directoria de onde é preciso ler um conjunto de ficheiros input, escrever os ficheiros

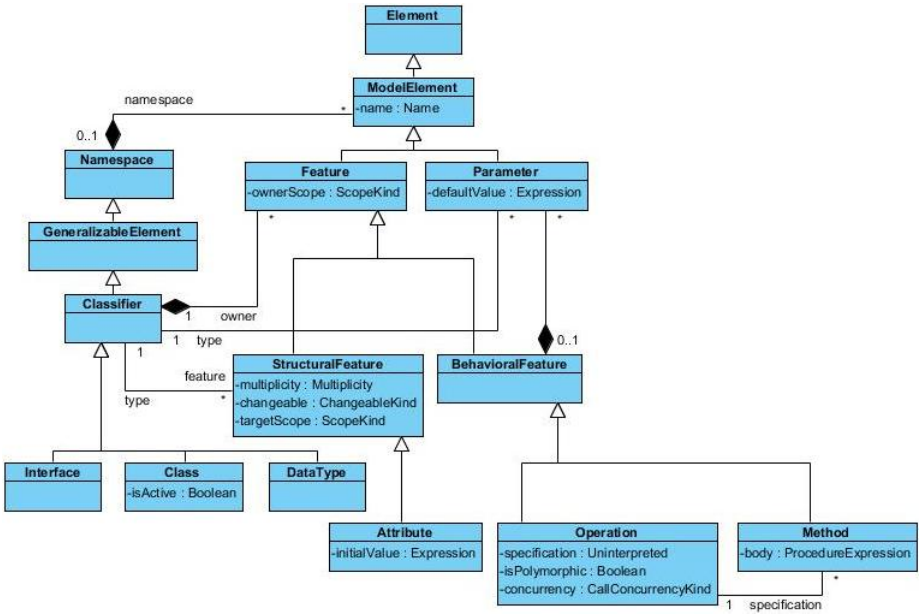

Figura 4.17: *Metamodelo UML para classes (adaptada de [Object Management Group, 2006])*

automaticamente gerados, entre outras informações.

Sobre as classes LerFicheiro e EscreverFicheiro, interessa mencionar que servem para fazer leituras aos ficheiros de input, e também para escrever os ficheiros gerados pela aplicação, respectivamente. Quanto à classe LerFicheiro, há que referir que faz o processamento de ficheiros *.class*, ou seja, ficheiros *.java* compilados. Deste modo, qualquer ficheiro de input introduzido pelo utilizador terá de respeitar este formato.

Para fazer o processamento de ficheiros *.class*, de modo a poder extrair de lá o nome da classe, as suas variáveis de instância e assinaturas de métodos, devolvendo esses resultados sob a forma de entidades presentes neste package (Figura 4.16), nomeadamente Metodo e VariavelInstancia, foram especificados os métodos presentes em LerFicheiro. Estes métodos vão recorrer a uma API[2] já existente para processar ficheiros *.class*.

Usando esta API é então possível extrair dados importantes de ficheiros *.class*, de uma forma bastante simples. Os métodos de LerFicheiro apenas terão que os trabalhar e converter para os respectivos tipos de retorno especificados.

Para terminar a descrição das classes deste package, resta abordar a classe TextoComum, que engloba métodos também utilizados na gera-

[2] http://jakarta.apache.org/bcel/

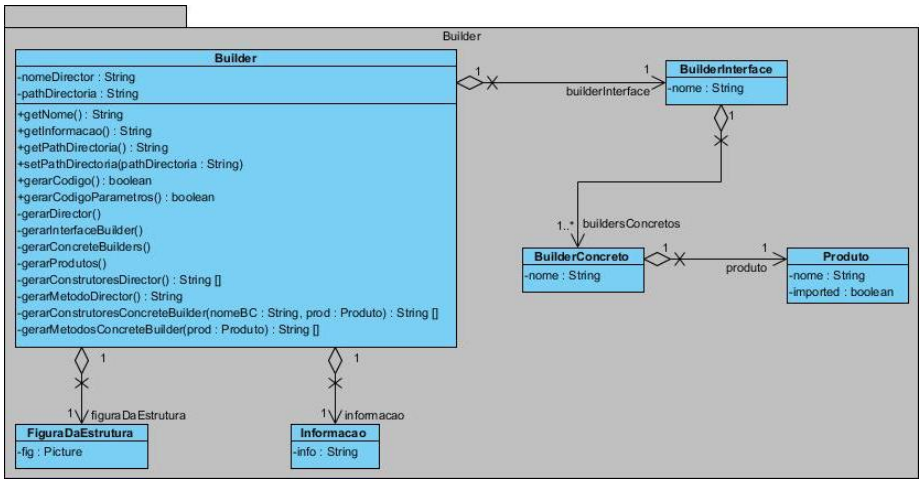

Figura 4.18: *Package Builder*

ção de código para qualquer um dos padrões de concepção. Para não estar a descrever todos os métodos, vai ser apenas referido o método header(nomePattern : String, nomeClasse : String) : String, que é utilizado para qualquer um dos padrões de concepção do PatGer. Então, pode-se afirmar que é um método que serve para se obter um cabeçalho comum para ser colocado em qualquer ficheiro gerado, indicando qual o padrão a que pertence, e ainda a informação de que foi gerado de forma automática.

Builder Na Figura 4.18 encontra-se representado o package relativo ao padrão de criação Builder, com todas as classes que engloba e que são importantes na representação interna deste padrão de concepção.

Em relação às classes aqui presentes, importa desde já referir que a classe Builder é uma subclasse da classe Pattern, que faz parte do package ProgramaPatterns, já anteriormente analisado.

Portanto, esta classe herda os métodos presentes na superclasse Pattern, embora esse detalhe não seja visível na Figura 4.18, por causa destas classes pertencerem a diferentes packages. Ainda sobre a classe Builder, pode-se dizer que contém como variáveis de instância o nome da entidade Director (faz parte da estrutura deste padrão de criação), o *path* da directoria onde os ficheiros a gerar deverão ser escritos, uma instância de BuilderInterface, e também uma figura com a sua estrutura (já visível no esboço mostrado na subsecção 4.1.1), juntamente com a informação adicional relativa a este padrão (também brevemente analisada na subsecção 4.1.1).

Quanto aos métodos existentes na classe Builder, apenas alguns são de acesso público, entre os quais importa salientar getInformacao() : String,

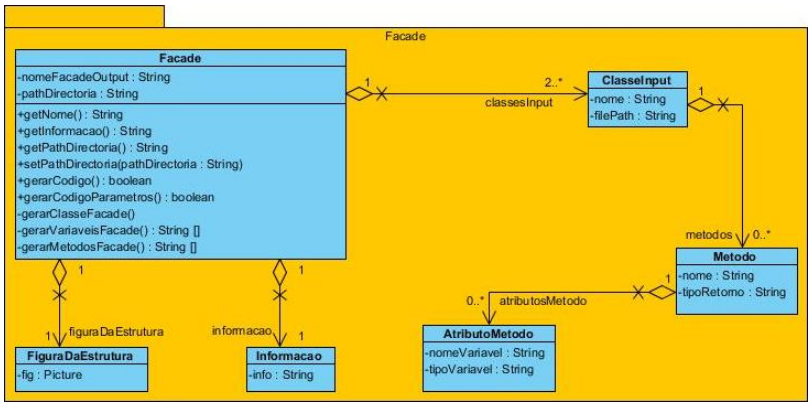

Figura 4.19: *Package Facade*

gerarCodigo() : boolean e gerarCodigoParametros() : boolean. Como os próprios nomes indicam, o primeiro método devolve a informação adicional respeitante ao Builder, o segundo trata de gerar os ficheiros quando a opção do utilizador for "A partir do zero", e o terceiro trata de gerar os ficheiros se a opção do utilizador for "Aproveitando algum código já existente". Estes dois métodos que geram o código para os ficheiros constituintes do Builder devolvem um valor do tipo boolean, mediante os ficheiros tenham sido gerados com sucesso ou não.

Sobre os métodos privados da classe Builder, pode-se verificar que estão todos relacionados com a geração das várias entidades que formam a estrutura deste padrão de criação, sendo apenas utilizados como métodos auxiliares dos métodos públicos, para que haja uma melhor organização do código presente nesta classe.

As restantes classes especificadas no package presente na Figura 4.18 servem para completar a lógica de negócio subjacente ao Builder, como sendo a existência de um interface para o Builder, o qual contém vários elementos do tipo BuilderConcreto, contendo cada um destes últimos um Produto.

Facade Este padrão pertence aos padrões estruturais, de acordo com a classificação sugerida em [Gamma et al., 1995]. Na Figura 4.19 encontra-se representado o package com as classes que tratam da geração de código associada ao Facade.

Através da sua observação podemos encontrar a classe Facade que é a classe principal deste package. É também uma subclasse de Pattern (package ProgramaPatterns) herdando portanto os métodos nela definidos. Contém como variáveis de instância o nome indicado para o ficheiro resultante da aplicação do Facade, além do *path* da directoria para onde esse ficheiro deve ser escrito, e também um conjunto de en-

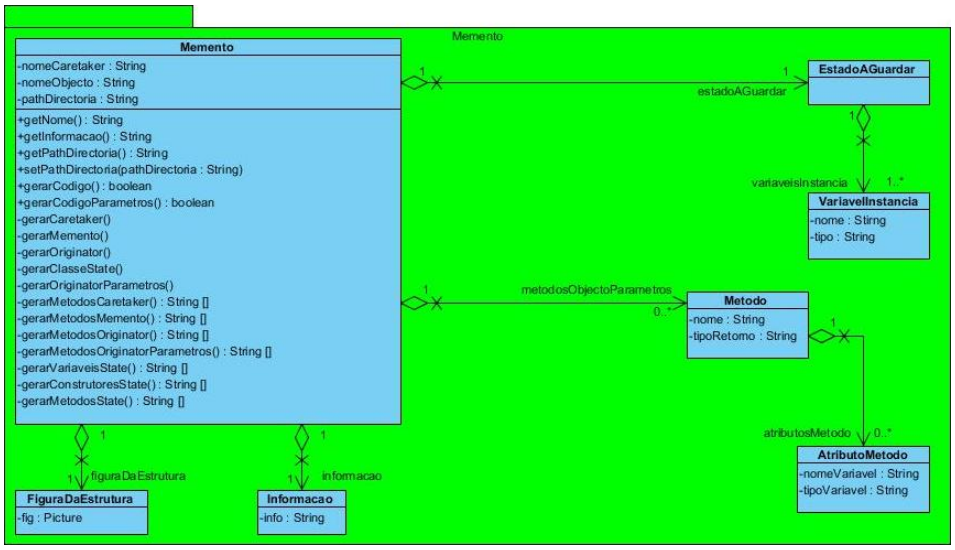

Figura 4.20: *Package Memento*

tidades ClasseInput (pelo menos duas) que representam os ficheiros aos quais o Facade será aplicado. Contém também uma figura associada à estrutura deste padrão, como aliás pode ser visto no esboço do Facade na subsecção 4.1.1, e também a informação adicional relativa ao padrão em questão.

À semelhança da descrição efectuada para o package Builder, também aqui na classe Facade existem definições de métodos públicos e privados, servindo estes últimos apenas como métodos auxiliares dos métodos públicos.

Por fim, a entidade ClasseInput tem como variáveis de instância o nome associado ao ficheiro que representa, bem como o seu *path*, de modo a que o PatGer saiba onde encontrar o ficheiro, para o poder ler e processar. A outra variável de instância desta classe é a classe Metodo, que na verdade pertence ao package ProgramaPatterns, mas, no entanto, encontra-se desenhada neste package apenas para tornar mais perceptível o relacionamento com ClasseInput, evitando-se assim ter lado a lado ambos os packages para tornar visível este relacionamento. Pela mesma razão está também incluída neste package a classe AtributoMetodo, que na realidade pertence ao package ProgramaPatterns.

Memento O package que segue diz respeito ao padrão de concepção Memento, pertencente aos grupo dos padrões comportamentais, de acordo com [Gamma et al., 1995].

Na Figura 4.20 está representado o package com as classes especificadas para a geração de código associada ao Memento. Pode-se encontrar a

classe Memento, que é a classe principal deste package, e que é também uma subclasse da classe Pattern (package ProgramaPatterns).

Como variáveis de instância, Memento apresenta os nomes associados às entidades que compõem a sua estrutura, juntamente com o *path* da directoria onde os ficheiros gerados pelo PatGer devem ficar. Além destas, existe ainda a classe EstadoAGuardar, formada por várias instâncias de VariavelInstancia, que na realidade pertence ao package ProgramaPatterns, mas apesar disso encontra-se representada neste package apenas para tornar mais perceptível o relacionamento entre estas classes. Por igual motivo, também as instâncias de Metodo e AtributoMetodo estão representadas neste package, embora também pertençam a ProgramaPatterns. As restantes variáveis de instância dizem respeito à figura relativa à estrutura do Memento, bem como à informação adicional para este padrão de comportamento.

Tal como já mostrado nos packages anteriores, também neste exemplo a classe Memento apresenta métodos públicos e privados, sendo que os métodos privados são utilizados como auxiliares dos métodos públicos.

4.2 Implementação do PatGer

Ao longo desta secção vai ser descrito o processo de implementação do PatGer, ou seja, tudo o que foi feito a partir da especificação, entretanto analisada, até se obter o programa final, num estado funcional, respeitando os requisitos propostos. Além disso, haverá ainda espaço para mostrar um exemplo simples de utilização da aplicação elaborada.

Assim sendo, nesta secção estarão presentes três subsecções para destacar aspectos importantes da solução obtida, como algumas decisões importantes tomadas antes de se passar ao seu desenvolvimento, além da tecnologia a utilizar e de um exemplo de utilização, juntamente com alguma informação pertinente para quem pretenda acrescentar mais funcionalidades ao programa elaborado.

4.2.1 Decisões de Implementação e Desenvolvimento

Nesta parte da dissertação vão ser descritas algumas decisões de implementação da solução, juntamente com o processo de desenvolvimento do PatGer.

Uma vez terminada a etapa de especificação da aplicação a desenvolver, e antes de começar propriamente a programar, foi necessário tomar algumas decisões, nomeadamente em relação à linguagem de programação a adoptar e ao resultado final a apresentar.

Assim, tal como sugerido pelos requisitos da aplicação, esta deveria ser independente da plataforma, e, por isso, a escolha recaiu sobre o desenvolvimento de um programa na linguagem de programação Java, cujo resultado

final é um ficheiro executável *.jar*, independente de qualquer plataforma. Pode assim ser executado num ambiente Windows, Linux ou MacOS, por exemplo, sem qualquer problema, e sem ser necessário voltar a compilar o código fonte.

Além deste executável mencionado, surgiu também a ideia de desenvolver uma outra versão desta mesma aplicação, em tudo idêntica à anterior, mas que pudesse ser utilizada como um plugin para o Ambiente de Desenvolvimento Integrado NetBeans[3].

A opção escolhida foi a de desenvolver estas duas versões aqui descritas, conseguindo-se assim que o PatGer possa ser utilizado em vários sistemas operativos, e que tenha ainda a mais-valia de poder funcionar como um plugin para o NetBeans, agradando àqueles que desenvolvem código Java neste IDE.

No que diz respeito ao desenvolvimento da aplicação proposta, em primeiro lugar iniciou-se a implementação da primeira versão aqui mencionada, ou seja, o desenvolvimento do ficheiro executável *.jar*. Para o efeito foi utilizado o Ambiente de Desenvolvimento Integrado NetBeans, pois é um IDE que permite a programação de aplicações Java, e com o qual já tive oportunidade de trabalhar em projectos anteriores.

Assim, o ponto de partida para a implementação, em Java, do PatGer, foi a especificação já entretanto apresentada. Desta forma, os vários packages e correspondentes classes já estavam pensados e modelados, simplificando bastante a etapa de programação.

O método utilizado para a programação deste sistema de software consistiu na implementação, de forma separada e faseada, dos vários packages entretanto descritos e analisados.

Numa primeira fase, foram codificadas as classes representadas nos packages UserInterface e ProgramaPatterns, relacionadas com a camada de interface e com a camada de negócio do PatGer, respectivamente.

Na segunda fase foram codificadas as classes presentes nos packages associados aos padrões suportados pelo programa. Foram também efectuados alguns testes de verificação que permitissem concluir que os ficheiros gerados estavam de acordo com a estrutura do padrão em causa, tendo em conta os parâmetros fornecidos pelo utilizador.

Interessa aqui dizer que foram sempre analisados os ficheiros gerados pela aplicação, tendo em atenção os nomes dos métodos e entidades neles presentes, garantindo que iam de encontro aos dados introduzidos pelo utilizador. Também se verificou se, nos casos de classes que implementam um dado interface, estavam a ser redefinidos todos os métodos nele presentes, por exemplo.

Uma vez terminado o desenvolvimento desta primeira versão do PatGer, ou seja, do ficheiro executável *.jar*, passou-se então ao desenvolvimento

[3] http://netbeans.org/

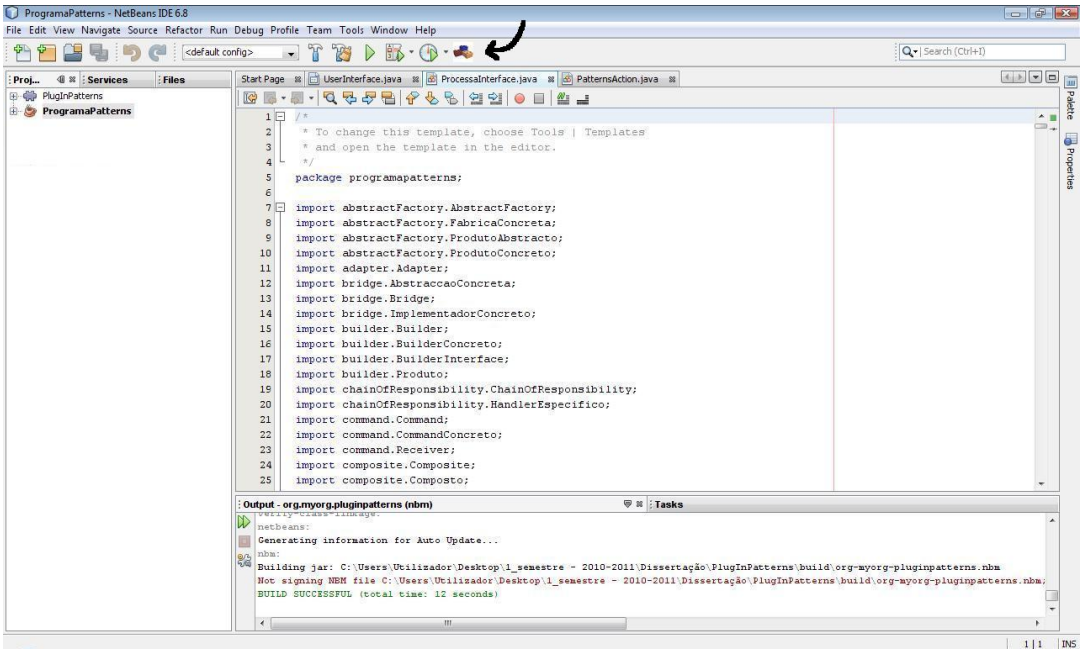

Figura 4.21: *Localização do ícone do PatGer (versão plugin) no NetBeans*

da segunda versão, isto é, do plugin para o NetBeans. Para esta versão do programa o objectivo passou por tentar aproveitar ao máximo o código já implementado na primeira versão. Assim sendo, o ideal seria que o plugin se tratasse apenas de um botão, presente no NetBeans, e que chamasse o ficheiro *.jar* entretanto desenvolvido. Isto foi conseguido porque foi possível importar para o plugin o ficheiro *.jar* desenvolvido, sendo considerado como um *wrapped JAR* pelo NetBeans.

Assim sendo, na Figura 4.21 pode ser visto o NetBeans com o ícone do PatGer, versão plugin, bem assinalado pela seta preta. O ícone é uma pequena figura composta por legos, que como se sabe podem ser utilizados para criar grandes estruturas, através da sua junção. De uma maneira semelhante, temos os padrões de concepção, que juntos podem formar partes importantes de uma aplicação de software, daí a escolha dos legos para ícone do PatGer.

Uma vez importado o ficheiro *.jar* para o plugin, a partir desse momento tornou-se possível que qualquer método de qualquer classe do plugin acedesse aos métodos e classes do ficheiro *.jar*.

Desta forma, na classe principal do plugin, no método chamado ao clicar no ícone do plugin, apenas foi necessário chamar o método main do ficheiro *.jar*, e, a partir daí, ao clicar no ícone do plugin automaticamente surgia a aplicação elaborada. Quanto ao plugin há ainda a referir que é um ficheiro *.nbm*, gerado pelo NetBeans, e que pode ser facilmente instalado neste IDE, ficando colocado no local assinalado na Figura 4.21.

Na Figura 4.22 pode-se ver o aspecto final do PatGer, quando utilizado

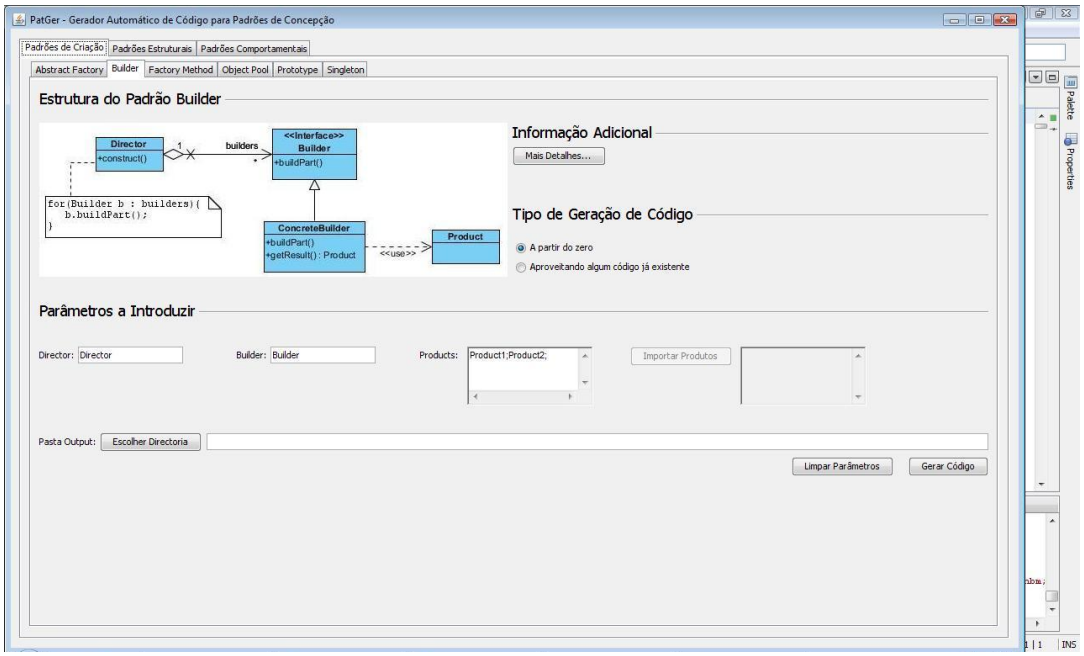

Figura 4.22: *Escolha do Builder no PatGer*

na versão plugin, após se clicar no botão que o inicializa, e escolhendo o padrão Builder. Facilmente se conclui que é em tudo idêntico aos esboços apresentados na subsecção 4.1.1. No Anexo G podem ser encontradas mais imagens relativas ao aspecto final da versão plugin do PatGer.

4.2.2 Exemplo de Utilização

Durante esta subsecção vai ser mostrado, de forma breve e sucinta, um exemplo muito simples de utilização do PatGer. É um exemplo que serve apenas para mostrar que o programa está a funcionar correctamente, para o caso considerado.

Assim, para já interessa referir que o exemplo de utilização é sobre o padrão estrutural Facade, que necessita de receber como entrada duas ou mais classes Java, em formato já compilado, ou seja, no formato *.class*. Serão portanto mostrados os ficheiros recebidos, que neste caso são três, e o ficheiro gerado como output.

Para que se possa perceber o conteúdo dos ficheiros recebidos como input terão de ser aqui utilizados os ficheiros *.java*, caso contrário não se perceberia o conteúdo dos respectivos ficheiros *.class* (código Java compilado). Deste modo, nas Transcrições 4.1, 4.2 e 4.3 encontra-se o código associado aos três ficheiros *.java* utilizados, que são os ficheiros Classe1, Classe2 e Classe3.

```
1 public class Classe1 {
2     public Classe1(){
```

```
3
4       }
5
6       public int metodoC11(int x, int y){
7           if(x > y){
8               return x;
9           }
10          else{
11              return y;
12          }
13      }
14
15      public String[] metodoC12(){
16          return new String[10];
17      }
18  }
```

Transcrição 4.1: *Classe1.java*

```
1  public class Classe2 {
2
3       private int x = 0;
4
5       public Classe2(){
6
7       }
8
9
10      public void metodoC21(){
11          this.x = 10;
12      }
13
14      public void metodoC22(int y){
15           this.x += y;
16      }
17  }
```

Transcrição 4.2: *Classe2.java*

```
1  public class Classe3 {
2
3       public Classe3(){
4
5       }
6
7       public int metodoC31(int x, int y, int z, int[] array){
8           return (array[1] + x - y + z);
9       }
10 }
```

Transcrição 4.3: *Classe3.java*

Na Figura 4.23 encontra-se uma imagem da versão plugin do PatGer, para o padrão Facade, onde foram escolhidos como input os já referidos ficheiros *.class*, Classe1, Classe2 e Classe3. Logo de seguida foi seleccionada a directoria para a qual irá o ficheiro gerado automaticamente pelo programa.

Após feita a escolha da pasta output, foi então dada a ordem de geração do ficheiro FacadeExemplo, neste caso concreto. A aplicação colocou o ficheiro

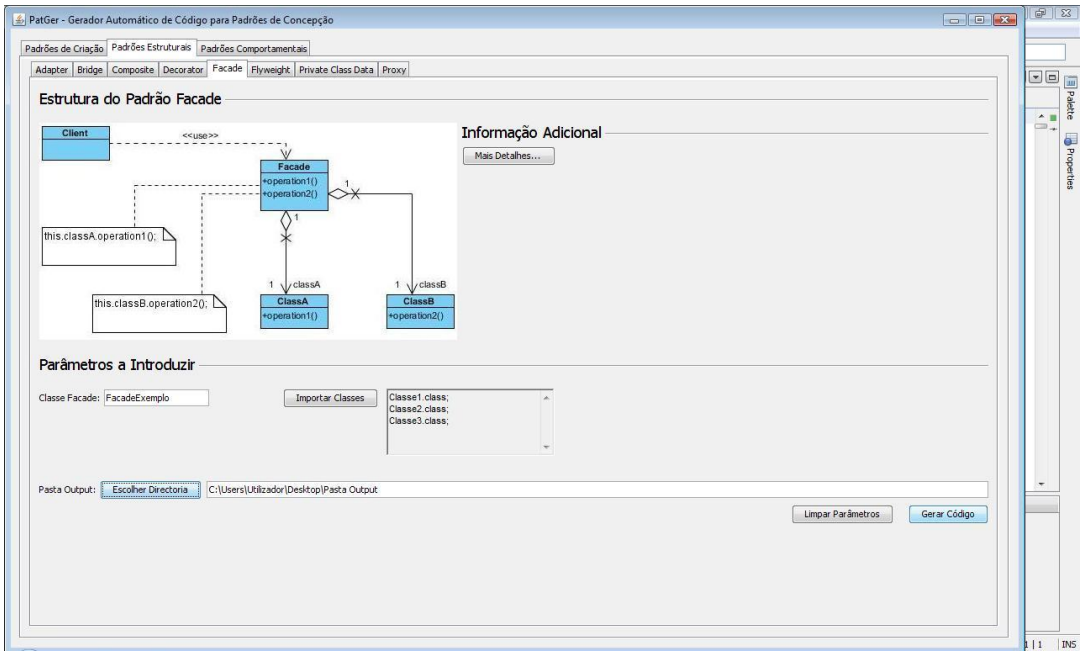

Figura 4.23: *Exemplo de utilização do PatGer*

gerado na directoria escolhida e, quanto ao seu conteúdo, é o que se encontra na Transcrição 4.4.

```
1  /**
2  *
3  * Pattern Facade
4  * FacadeExemplo gerada automaticamente pela aplicacao
5  *
6  */
7  public class FacadeExemplo{
8      private Classe1 classe1 = new Classe1();
9      private Classe2 classe2 = new Classe2();
10     private Classe3 classe3 = new Classe3();
11
12     public FacadeExemplo() {
13
14     }
15
16     public int metodoC11(int x, int y){
17         return this.classe1.metodoC11(x, y);
18     }
19
20      public String[] metodoC12(){
21         return this.classe1.metodoC12();
22      }
23
24      public void metodoC21(){
25         this.classe2.metodoC21();
26      }
27
28      public void metodoC22(int y){
29         this.classe2.metodoC22(y);
```

```
30      }
31
32      public int metodoC31(int x, int y, int z, int[] array){
33          return this.classe3.metodoC31(x, y, z, array);
34      }
35 }
```

Transcrição 4.4: *FacadeExemplo.java*

Como se verifica através da análise do código do ficheiro FacadeExemplo.java, gerado pelo PatGer e representado na Transcrição 4.4, todos os métodos presentes nos ficheiros utilizados como input foram capturados e colocados no ficheiro output, tal como o padrão estrutural Facade sugere.

Para terminar, também o conteúdo de cada um dos métodos presentes na Transcrição 4.4 foi gerado automaticamente, tendo em consideração o tipo de retorno do método original, e a entidade no qual está realmente definido, para que seja invocado de forma correcta.

4.2.3 Como acrescentar novos padrões

Ao longo desta subsecção vai ser descrito como é que deverá ser feito, para, de futuro, serem adicionadas funcionalidades à aplicação elaborada, nomeadamente dar suporte a outros padrões.

Desta forma, como já mencionado ao longo da subsecção 4.1.2, a lógica de negócio do PatGer está dividida em vários packages, de acordo com as funções que desempenham. Assim, existe um package para as classes directamente responsáveis pela implementação da interface da aplicação, além de um package associado a classes que são frequentemente utilizadas na geração de código para os vários padrões de concepção. Além disto, o código Java associado à geração de código para os padrões está também dividido em vários packages, sendo um para cada padrão.

Assim, quando se pretender adicionar um novo padrão, as classes que o implementam deverão ser também agrupadas num novo package. Este novo package deverá, de preferência, ter o mesmo nome do novo padrão, por uma questão de coerência com os outros packages já existentes no PatGer. Cumprindo esta regra será possível ter o código do programa devidamente dividido e de fácil compreensão, simplificando a manutenção da aplicação.

Nesse novo package é fundamental que exista uma classe que herde os métodos definidos na classe Pattern, presente no package ProgramaPatterns. O nome dessa classe deverá ser idêntico ao do novo padrão, de novo por uma questão de coerência com o restante código já desenvolvido.

A referida classe deverá redefinir os métodos herdados que forem considerados convenientes, como por exemplo os métodos directamente ligados à geração de código. Claro que a existência ou não de métodos privados auxiliares, além de classes auxiliares para implementar as entidades que fazem parte da estrutura do novo padrão, são decisões a tomar por quem for codificar essa nova funcionalidade.

Outro aspecto importante que falta mencionar prende-se com a informação adicional que diga respeito ao novo padrão. Sugere-se que seja um pequeno texto que indique o papel de cada uma das classes que formam a estrutura do novo padrão, elucidando assim o utilizador na hora de escolher esse novo padrão.

Até aqui foram analisadas as tarefas respeitantes à introdução de um novo package no sistema, juntamente com as classes que deve conter, faltando agora dizer o que deve ser feito ao nível das classes do package UserInterface, e também ao nível da interface da aplicação.

Em primeiro lugar deverá ser desenhada a interface para o novo padrão, juntamente com os seus componentes, numa nova aba a juntar ao painel onde se encontram os outros padrões de concepção. Importa dizer que deverá existir pelo menos um botão para a geração de código, para a escolha da directoria de output, e também para mostrar a informação adicional. Além disso, convém colocar uma ilustração da estrutura do novo padrão, que é sempre importante para esclarecer o utilizador no momento de recorrer ao programa elaborado. Tudo isto deverá ser efectuado na classe UserInterface, pertencente ao package UserInterface, também já analisado na subsecção 4.1.2.

Em seguida, há que codificar o comportamento associado aos botões acima referidos. Desta forma, deverá ser tido em consideração o processamento dos dados recebidos do utilizador, sendo em seguida passados para a classe ProcessaInterface (package UserInterface), em métodos a definir por quem estiver a codificar. O passo seguinte é programar na classe ProcessaInterface os métodos que recebem os dados de UserInterface. Estes métodos terão agora de chamar os métodos apropriados para a geração de código pretendida, entretanto definidos pelo programador no package respeitante ao novo padrão.

Para terminar, é ainda importante referir que as alterações em classes já desenvolvidas pressupõem que quem o for fazer tem acesso ao código fonte do programa, ou seja, ao código Java que compilado originará a versão *.jar* do PatGer. Caso se pretenda acrescentar estas mesmas funcionalidades na versão plugin, então apenas será necessário ter acesso ao código fonte do plugin, e importar o novo *.jar* entretanto desenvolvido.

Capítulo 5

Casos de Estudo

Neste capítulo vão ser analisadas algumas possibilidades de utilização do PatGer em aplicações Java já desenvolvidas, quer para modificar a arquitectura dessas aplicações, com recurso aos padrões de concepção, quer para acrescentar novas funcionalidades, recorrendo também aos padrões de concepção.

Na primeira secção deste capítulo vai ser abordado o programa TEO (Sistema de Gestão de Serviço de Televisão (*pay per view*)), que é uma aplicação Java resultante de um trabalho de grupo elaborado durante o terceiro ano da Licenciatura em Engenharia Informática, em 2009.

Na segunda secção será tido em consideração o programa CleanSheets[1], desenvolvido em Java, e que implementa as mesmas funcionalidades que o Microsoft Excel.

Para ambos os programas serão brevemente descritas as suas funcionalidades, juntamente com os packages e classes mais importantes para o desenvolvimento dos respectivos casos de estudo.

No final do capítulo haverá ainda uma secção onde serão discutidos os casos de estudo efectuados, fazendo uma análise crítica aos resultados obtidos com a utilização do PatGer.

5.1 TEO

Nesta secção vai ser brevemente descrita a aplicação TEO (Sistema de Gestão de Serviço de Televisão (*pay per view*)), juntamente com algumas alterações acrescentadas no decorrer do desenvolvimento do caso de estudo, mostrando assim alguma aplicabilidade prática da aplicação de software entretanto elaborada.

O TEO, como já referido, foi um projecto de grupo, elaborado no terceiro ano da Licenciatura em Engenharia Informática, no âmbito da Unidade Curricular de Desenvolvimento de Sistemas de Software. Em linhas gerais,

[1] http://java-source.net/open-source/finance/cleansheets

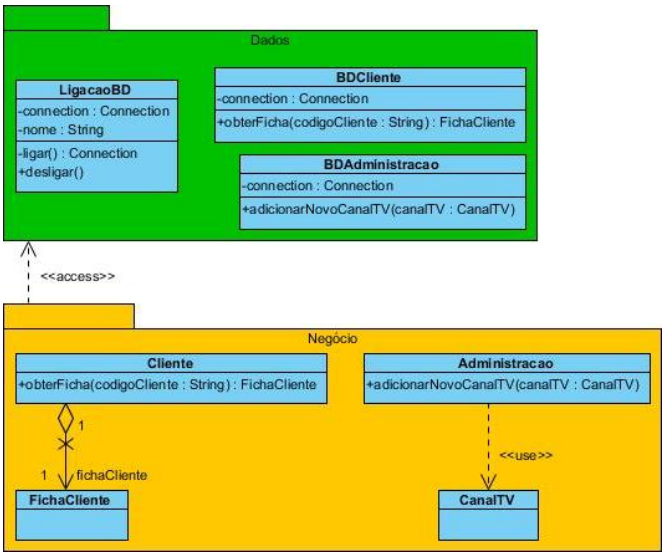

Figura 5.1: *Excerto do Diagrama de Classes do TEO*

trata-se de um programa escrito em Java que é responsável por implementar um sistema de gestão para um serviço de televisão, semelhante ao MEO[2], por exemplo. Permite assim a existência de clientes no sistema, os quais podem escolher um pacote de canais para subscrever, juntamente com canais *premium*, além do aluguer de filmes. Para além destas funcionalidades existe uma parte reservada para a administração e para os técnicos, onde é possível adicionar novos clientes no sistema, e registar pedidos de manutenção, por exemplo.

Sobre a arquitectura de classes do TEO, ou seja, a sua lógica de negócio, é importante dizer que as várias classes estão agrupadas em três packages, um para cada uma das camadas de software existentes: Interface, Negócio e Dados.

Relativamente a modificações a efectuar no TEO, após alguma análise e inspecção do código escrito há dois anos atrás, foi possível encontrar de imediato algo que suscitou o interesse de ser melhorado. Trata-se da lógica de negócio relativa aos acessos à base de dados do TEO, onde estava a ser constantemente criada uma nova conexão à base de dados sempre que algum método de alguma dessas classes tivesse necessidade de fazer uma *query*. Esta situação encontra-se ilustrada na Figura 5.1, onde temos representado um pequeno excerto do diagrama de classes deste programa.

Como se verifica pela observação da Figura 5.1, existe no package Dados uma classe LigacaoBD, onde é feita uma nova conexão à base de dados, no construtor dessa entidade; há ainda neste exemplo duas outras classes, responsáveis pelos métodos relacionados com os clientes e com a administração

[2]http://www.meo.pt/Pages/homepage.aspx

do sistema, que são BDCliente e BDAdministracao, respectivamente.

As classes do package Negócio comunicam com as classes de Dados, sendo que, por exemplo, no método definido em Cliente é criada uma nova instância de LigacaoBD e de BDCliente, recebendo esta última a variável connection já inicializada com uma ligação à base de dados. Em seguida, nas Transcrições 5.1, 5.2 e 5.3 pode ser visto um excerto do código de Cliente, LigacaoBD e BDCliente, para que se possa ter uma ideia do que acontece ao chamar o método obterFicha(codigoCliente : String) : FichaCliente. De realçar que o que se passa com a classe Administracao e o método adicionarNovoCanalTV(canalTV : CanalTV) é idêntico, daí não ser necessário mostrar o código para essa classe.

```
1 public class Cliente {
2     private FichaCliente fichaCliente;
3
4     public Cliente(){
5     }
6
7     public FichaCliente getFichaCliente() {
8         return fichaCliente;
9     }
10
11    public void setFichaCliente(FichaCliente fichaCliente) {
12        this.fichaCliente = fichaCliente;
13    }
14
15    public FichaCliente obterFicha(String codigoCliente){
16        LigacaoBD ligacaoBD = new LigacaoBD();
17        BDCliente bdCliente = new BDCliente(ligacaoBD.
              getConnection());
18        return bdCliente.obterFicha(codigoCliente);
19    }
20 }
```

Transcrição 5.1: *Cliente.java*

```
1 public class LigacaoBD {
2     private Connection con;
3     private String nome = "DSS_TEO";
4
5     public LigacaoBD(){
6         this.con = this.ligar();
7     }
8
9     public LigacaoBD(String nome) {
10        this.nome = nome;
11        this.con = this.ligar();
12    }
13
14    public Connection getConnection() {
15        return this.con;
16    }
17
18    public String getNome() {
19        return nome;
20    }
21
22    public void setNome(String nome) {
23        this.nome = nome;
```

```
24      }
25
26      private Connection ligar() {
27          try {
28              Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
29          }catch (ClassNotFoundException e) {
30              e.printStackTrace();
31          }
32
33          try {
34              String ligacao = ("jdbc:odbc:" + this.nome);
35              return DriverManager.getConnection(ligacao);
36          }catch (SQLException e) {
37              e.printStackTrace();
38              return null;
39          }
40      }
41
42      public void desligar() {
43          try {
44              this.con.close();
45          }catch (SQLException e) {
46              e.printStackTrace();
47          }
48      }
49  }
```

Transcrição 5.2: *LigacaoBD.java*

```
1  public class BDCliente {
2      private Connection conn;
3
4      public BDCliente(Connection conn) {
5          this.conn = conn;
6      }
7
8      public FichaCliente obterFicha(String codigoCliente) {
9          FichaCliente f = new FichaCliente();
10         String nome_tabela = "ClientesInfo";
11         Statement st;
12         ResultSet res = null;
13         try {
14             String sql = "SELECT Nome,Username,Morada,Idade,Email,
                   ContaServicos FROM " + nome_tabela + " WHERE
                   Codigo='" + codigoCliente + "'";
15             st = this.conn.createStatement();
16             res = st.executeQuery(sql);
17                 //iterar sobre o ResultSet e construir o objecto
                       FichaCliente
18             return f;
19          }
20          catch (SQLException e) {
21              e.printStackTrace();
22              return null;
23          }
24     }
25 }
```

Transcrição 5.3: *BDCliente.java*

Para resolver esta situação descrita pode-se recorrer ao padrão de criação

Object Pool, criando-se assim uma *pool* de ligações à base de dados, e, sempre que necessário, apenas se terá de pedir à *pool* uma conexão, poupando-se no tempo levado pelo programa a estabelecer uma nova conexão com a base de dados. Essa conexão poderá depois ser passada por parâmetro para as restantes classes de Dados que precisem dela, como por exemplo a classe BDCliente.

Além disto, de modo a evitar que as classes do package Negócio tenham de comunicar com várias classes do package Dados, pode-se utilizar o padrão estrutural Facade, que permite agregar numa única classe vários métodos de várias classes. Desta forma, as classes do package Negócio apenas comunicarão com essa nova classe, e claro está, com a classe que implementa a *pool* de ligações à base de dados, sempre que seja necessário interagir com o package Dados.

Então, após idealizada a solução acima mencionada, passou-se à sua implementação propriamente dita. Para o efeito, recorreu-se à versão plugin para o NetBeans do PatGer, elaborado no decurso da presente dissertação, de modo a implementar os respectivos padrões de concepção para este caso concreto.

Em primeiro lugar tratou-se da geração automática de código para o padrão Object Pool, utilizando como input o ficheiro LigacaoBD.class, que se trata da classe Java LigacaoBD já compilada. O código Java correspondente foi mostrado um pouco atrás, na Transcrição 5.2. Na Figura 5.2 encontra-se a execução do PatGer de forma a gerar o código para implementar o padrão Object Pool.

Como resultado foi obtida uma classe Java, chamada ObjectPool.java, que implementa o referido padrão para a classe usada como input, ou seja, a classe LigacaoBD. O código obtido de forma automática pode ser visto na Transcrição 5.4.

```
1 /**
2  *
3  * Pattern Object Pool
4  * ObjectPool gerada automaticamente pela aplicacao
5  *
6  */
7 public class ObjectPool {
8     private ArrayList<LigacaoBD> reusables;
9     private int maximo = 5;
10    private String nome;
11
12    private ObjectPool(String nome) {
13        this.nome = nome;
14        reusables = new ArrayList<LigacaoBD>();
15        for (int i = 0; i < this.maximo; i++) {
16            LigacaoBD reusable = new LigacaoBD(nome);
17            this.reusables.add(reusable);
18        }
19    }
20
21    private static class ObjectPoolHolder {
22        private static final ObjectPool INSTANCE = new ObjectPool(
```

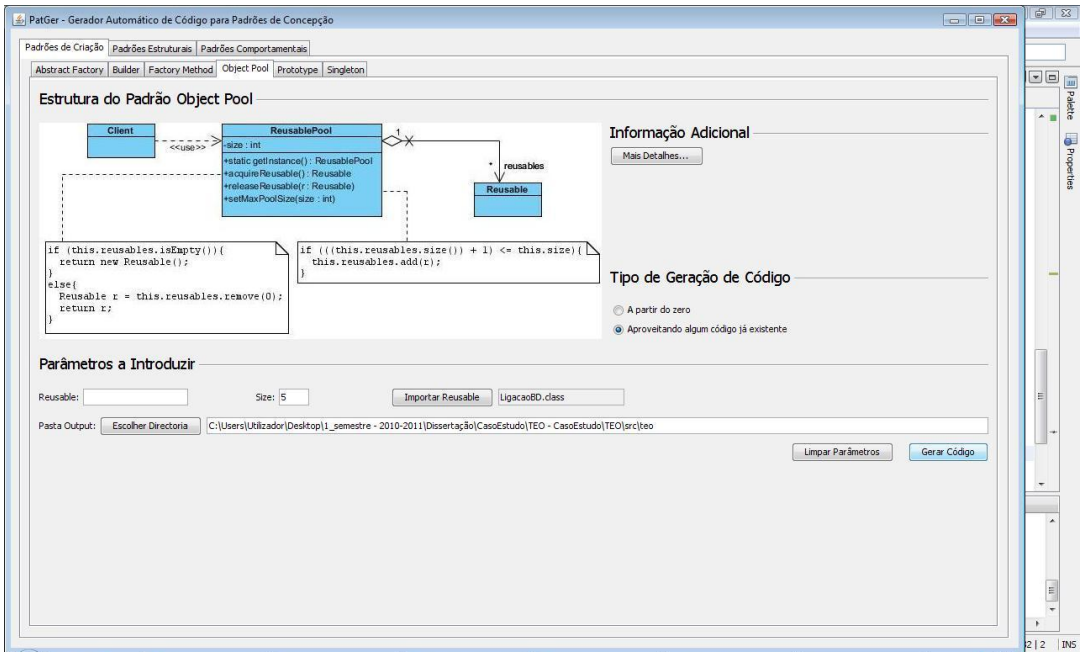

Figura 5.2: *Utilização do PatGer para o Object Pool na aplicação TEO*

```
            "DSS_TEO");
23      }
24
25      public static ObjectPool getInstance() {
26          return ObjectPoolHolder.INSTANCE;
27      }
28
29      public LigacaoBD acquireLigacaoBD() {
30          if (this.reusables.isEmpty()) {
31              return new LigacaoBD(this.nome);
32          } else {
33              LigacaoBD res = this.reusables.remove(0);
34              return res;
35          }
36      }
37
38      public void releaseLigacaoBD(LigacaoBD reusable) {
39          if (((this.reusables.size()) + 1) <= this.maximo) {
40              this.reusables.add(reusable);
41          }
42      }
43
44      public void setMaxPoolSize(int max) {
45          int size = this.reusables.size();
46          int diff = max - size;
47          if (diff > 0) {
48              for (int i = size; i < max; i++) {
49                  LigacaoBD res = new LigacaoBD(this.nome);
50                  this.reusables.add(res);
51              }
52          }
53          if (diff < 0) {
```

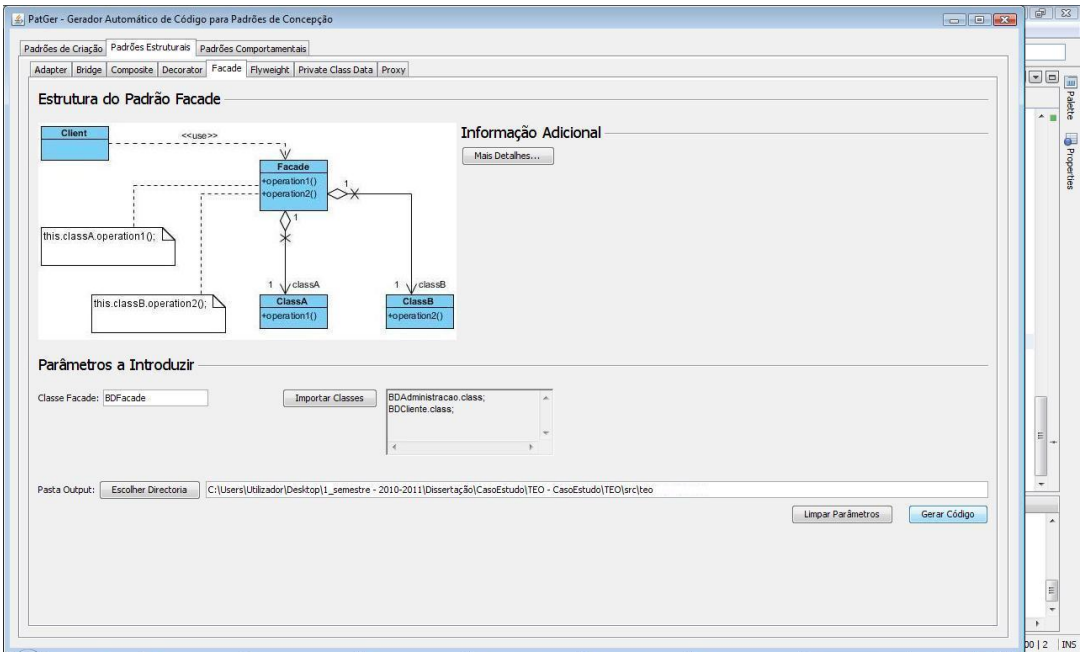

Figura 5.3: *Utilização do PatGer para o Facade na aplicação TEO*

```
54                 for (int i = (size - 1); i >= max; i--) {
55                     this.reusables.remove(i);
56                 }
57             }
58             this.maximo = max;
59         }
60 }
```

Transcrição 5.4: *ObjectPool.java*

Uma vez gerado o código entretanto apresentado, passou-se à geração automática de código para o padrão Facade. Importa aqui dizer que apenas foram utilizados como input dois ficheiros *.class*, correspondentes às classes Java BDCliente e BDAdministracao do package Dados, para se poder assim mostrar um exemplo mais pequeno. Portanto, na Figura 5.3 pode ser visto em execução o PatGer, de modo a gerar o código para a implementação de um Facade para dois ficheiros *.class*.

Como resultado da aplicação do plugin foi então obtido o ficheiro BDFacade.java, cujo conteúdo se encontra representado na Transcrição 5.5.

```
1 /**
2  *
3  * Pattern Facade
4  * BDFacade gerada automaticamente pela aplicacao
5  *
6  */
7 public class BDFacade {
8     private Connection conn;
9     private BDCliente bdCliente;
```

```
10      private BDAdministracao bdAdministracao;
11
12      public BDFacade(Connection conn) {
13          this.conn = conn;
14      }
15
16      public void adicionarNovoCanalTV(CanalTV canalTV) {
17          this.bdAdministracao = new BDAdministracao(this.conn);
18          this.bdAdministracao.adicionarNovoCanalTV(canalTV);
19      }
20
21      public FichaCliente obterFicha(String codigoCliente) {
22          this.bdCliente = new BDCliente(this.conn);
23          return bdCliente.obterFicha(codigoCliente);
24      }
25  }
```

Transcrição 5.5: *BDFacade.java*

Terminada a geração automática de código apenas foi necessário rever os métodos das entidades de Negócio, que são Cliente e Administracao. Teve de ser alterado o código relativo ao método obterFicha(codigoCliente : String) : FichaCliente e adicionarNovoCanalTV(canalTV : CanalTV). Como o procedimento adoptado foi semelhante, apenas vai aqui ser mostrado o novo código para o método obterFicha(codigoCliente : String) : FichaCliente (ver Transcrição 5.6), utilizando já a *pool* de ligações à base de dados, e a classe BDFacade entretanto criada.

```
1  public FichaCliente obterFicha(String codigoCliente) {
2      ObjectPool pool = ObjectPool.getInstance();
3      LigacaoBD ligacao = pool.acquireLigacaoBD();
4      Connection conn = ligacao.getConnection();
5      BDFacade bdFacade = new BDFacade(conn);
6      FichaCliente res = bdFacade.obterFicha(codigoCliente);
7      pool.releaseLigacaoBD(ligacao);
8      return res;
9  }
```

Transcrição 5.6: *Excerto de Cliente.java*

Para melhor se visualizar a nova arquitectura de classes do TEO, após as alterações introduzidas, na Figura 5.4 encontra-se um excerto do diagrama de classes modificado.

Terminada esta melhoria da lógica de negócio do TEO surgiu uma outra ideia, um pouco diferente da anterior. Assim, tal como o MEO disponibiliza um conjunto de serviços além dos canais de televisão e do aluguer de filmes, como por exemplo possibilitar aos utilizadores saber o estado do tempo numa dada região, porque não implementar algo do género no TEO?

Desta forma foi então pensado em dotar o TEO da capacidade para revelar o estado do tempo para um conjunto de cidades portuguesas à escolha, e também ser capaz de mostrar a chave do último sorteio do Euromilhões. Para isto foram programadas duas classes Java, que são as classes Tempo e Euromilhoes, e foram colocadas num novo package, denominado FuncionalidadesExtra. Importa aqui referir que estas classes, para serem capazes de

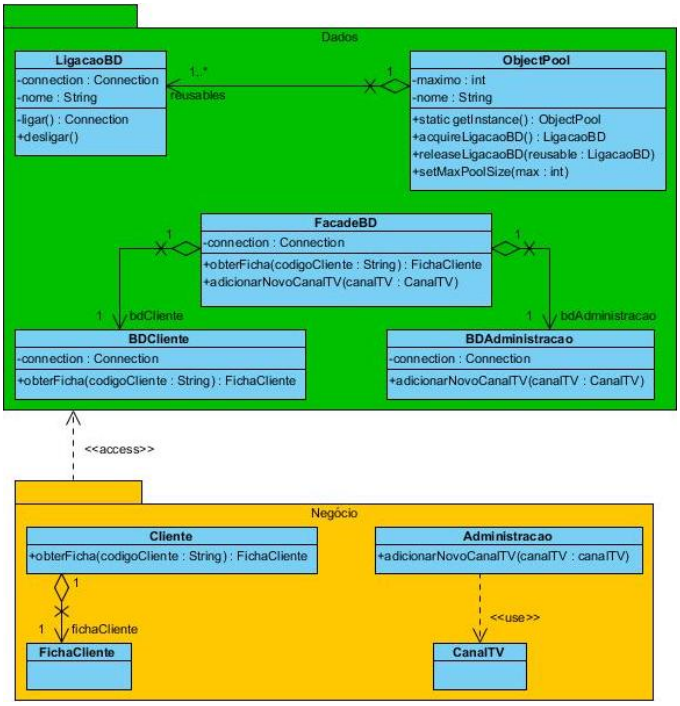

Figura 5.4: *Excerto do Diagrama de Classes modificado do TEO*

devolver quer o estado do tempo de uma cidade, quer a chave do último sorteio do Euromilhões, ligam-se via *web* a páginas[3][4] que devolvem esses mesmos dados. Depois é apenas necessário fazer *parsing* do texto html recebido, encontrando assim os dados pretendidos para estes casos concretos. Apresenta-se na Transcrição 5.7 o código para a classe Tempo.java, e como o código de Euromilhoes.java é bastante semelhante, apenas vai ser mostrado um destes exemplos.

```
1 public class Tempo {
2     private String cidade;
3     private String estadoTempo;
4     private String temperaturaCidade;
5
6     public Tempo() {
7     }
8
9     public Tempo(String cidade, String codigo) {
10        this.cidade = cidade;
11        this.procurarEstadoTempo(codigo);
12    }
13
14    public String getCidade() {
15        return cidade;
```

[3] http://weather.yahooapis.com/forecastrss?w=codigo&u=c, onde *codigo* é substituído pelos números que identificam a cidade desejada

[4] http://www.euromilhoes.com/index_intro.php

```
16      }
17
18      public void setCidade(String cidade) {
19          this.cidade = cidade;
20      }
21
22      public String getEstadoTempo() {
23          return estadoTempo;
24      }
25
26      public void setEstadoTempo(String estadoTempo) {
27          this.estadoTempo = estadoTempo;
28      }
29
30      public String getTemperaturaCidade() {
31          return temperaturaCidade;
32      }
33
34      public void setTemperaturaCidade(String temperaturaCidade) {
35          this.temperaturaCidade = temperaturaCidade;
36      }
37
38      private void procurarEstadoTempo(String codigoCidade) {
39          try {
40              URL url = new URL("http://weather.yahooapis.com/
                    forecastrss?w=" + codigoCidade + "&u=c");
41              InputStream is = url.openStream();
42              DataInputStream dis = new DataInputStream(new
                    BufferedInputStream(is));
43              String res = "";
44              String aux = "";
45              while ((aux = dis.readLine()) != null) {
46                  res += aux;
47              }
48              String[] res1 = res.split("<b>");
49              String[] res2 = res1[1].split(",");
50              String[] condicaoTempo = res2[0].split("/>");
51              String[] temperatura = res2[1].split("<");
52              this.estadoTempo = condicaoTempo[1];
53              this.temperaturaCidade = temperatura[0];
54          }
55          catch (Exception e) {
56              e.printStackTrace();
57          }
58      }
59
60      @Override
61      public String toString() {
62          String res = "";
63          res += "Previsao do Estado do Tempo para " + cidade + "\n\
                    n";
64          res += "Condicao Geral: " + this.estadoTempo + "\n";
65          res += "Temperatura: " + this.temperaturaCidade;
66          return res;
67      }
68  }
```

Transcrição 5.7: *Tempo.java*

Posteriormente, utilizando o PatGer, foi gerado automaticamente o código correspondente ao padrão de criação Factory Method, que englobou as

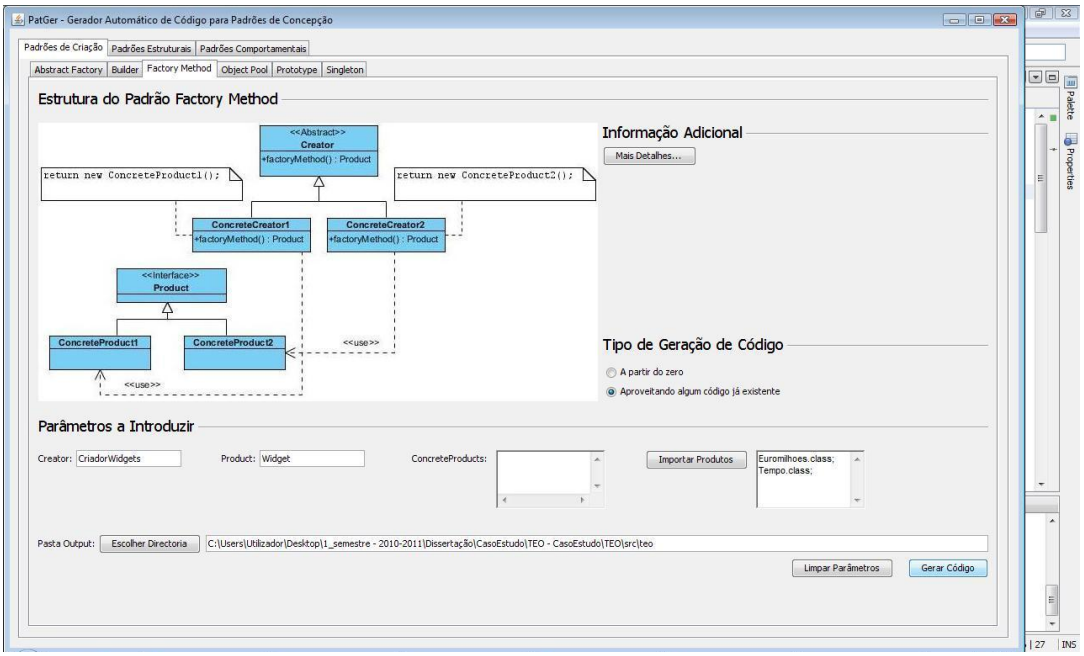

Figura 5.5: *Utilização do PatGer para o Factory Method na aplicação TEO*

classes Tempo e Euromilhoes, já codificadas e compiladas (ficheiros *.class*), como fazendo parte de um interface Widget, e gerou também as restantes classes que completam a estrutura deste padrão. Na Figura 5.5 encontra-se o plugin no momento da geração de código para este caso.

Uma vez que o código gerado é semelhante para os dois ficheiros utilizados como input, vai ser apenas mostrado o código directamente associado à classe Euromilhoes, presente nas classes CriadorWidgets, ConcreteCreator_Euromilhoes e Widget, que pode ser visto nas Transcrições 5.8, 5.9 e 5.10.

```
1  /**
2  *
3  * Pattern Factory Method
4  * CriadorWidgets gerada automaticamente pela aplicacao
5  *
6  */
7 public abstract class CriadorWidgets {
8
9     public CriadorWidgets() {
10
11    }
12
13
14    public abstract Widget factoryMethod();
15
16 }
```

Transcrição 5.8: *CriadorWidgets.java*

```
1 /**
2  *
3  * Pattern Factory Method
4  * ConcreteCreator_Euromilhoes gerada automaticamente pela
       aplicacao
5  *
6  */
7 public class ConcreteCreator_Euromilhoes extends CriadorWidgets {
8
9     @Override
10    public Widget factoryMethod() {
11        return new Euromilhoes();
12    }
13 }
```

Transcrição 5.9: *ConcreteCreator_Euromilhoes.java*

```
1 /**
2 *
3 * Pattern Factory Method
4 * Widget gerada automaticamente pela aplicacao
5 *
6 */
7 public interface Widget {
8
9     @Override
10    public String toString();
11 }
```

Transcrição 5.10: *Widget.java*

Após gerado o código para implementar o padrão de criação Factory Method para as duas entidades já referidas, foi então desenvolvida uma janela, muito simples, para ser lançada a partir do TEO. Essa janela vai mostrar o estado do tempo para uma região portuguesa à escolha, dentro das regiões suportadas pelo programa, e vai mostrar também a chave do último sorteio do Euromilhões, mediante a opção escolhida pelo utilizador. Nas Figuras 5.6 e 5.7 pode ser vista a janela implementada, para cada um dos casos.

O código associado à acção desencadeada por um clique no botão "EuroMilhoes" pode ser visto na Transcrição 5.11, sendo igualmente semelhante caso a opção seja o botão "Tempo".

```
1 private void jButton2ActionPerformed(java.awt.event.ActionEvent
      evt) {
2     CriadorWidgets cw = new ConcreteCreator_Euromilhoes();
3     Widget euromilhoes = cw.factoryMethod();
4     textArea1.setText(euromilhoes.toString());
5 }
```

Transcrição 5.11: *Excerto de janelaAdicional.java*

Quanto à estrutura de classes pertencentes ao novo package desenvolvido, já com a aplicação do padrão Factory Method, pode ser encontrada na Figura 5.8.

Figura 5.6: *Janela adicional para o TEO (a)*

Figura 5.7: *Janela adicional para o TEO (b)*

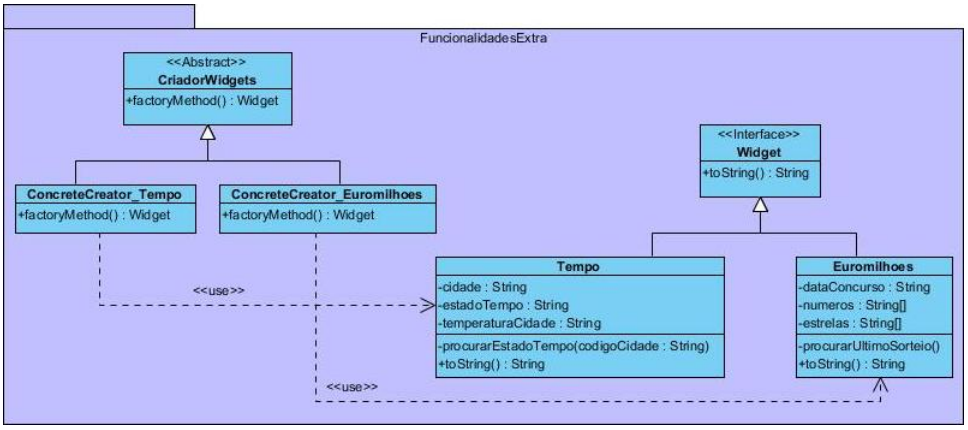

Figura 5.8: *Packge FuncionalidadesExtra do TEO*

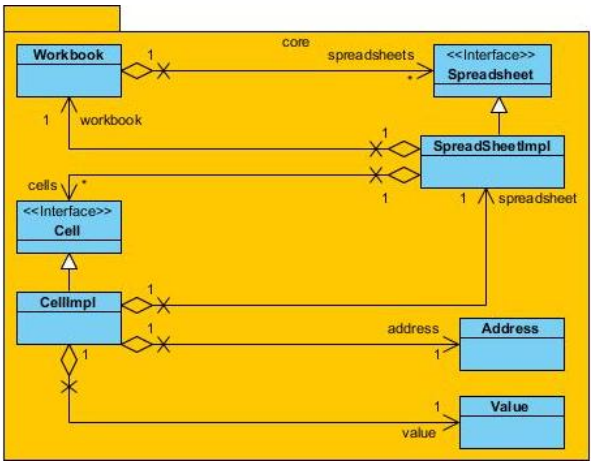

Figura 5.9: *Package core da aplicação CleanSheets*

5.2 CleanSheets

Ao longo desta secção será brevemente analisada a aplicação CleanSheets, que se trata de uma aplicação idêntica ao Microsoft Excel, desenvolvida em Java[5]. É um programa *open source*, sendo facultado o acesso não só ao programa final (ficheiro *.jar*), como também ao seu código fonte[6]. Além disto, vai ser ainda descrita uma possível utilização do PatGer para a modificação de algum código do CleanSheets, recorrendo aos padrões de concepção.

Relativamente ao CleanSheets, interessa ainda mencionar o recurso a um outro ficheiro *.jar*, que é o ANTLR[7], com o qual é feito o *parsing* e interpretação das fórmulas utilizadas no programa.

Para melhor se compreender a estrutura de classes do CleanSheets, em seguida apresentam-se duas figuras com os packages principais do programa, que são core e io, sendo que existem várias classes e/ou outros packages dentro dos representados nas Figuras 5.9 e 5.10. Para não tornar confuso o diagrama optou-se por não o representar com todas essas entidades, mas sim com as essenciais para a compreensão global do problema.

Na Figura 5.9 está representado o package core que contém as entidades principais de negócio, como a classe Workbook, que representa uma folha de cálculo, contendo várias células (subclasse CellImpl do interface Cell) e estando associada a uma SpreadSheet. Além disso, uma instância de CellImpl tem associado um endereço (Address) e um valor (Value).

Na Figura 5.10 está presente o package io, onde existem classes responsáveis pelo input/output do CleanSheets. Permitem assim que se possa serializar para disco todas as alterações efectuadas numa folha de cálculo

[5] http://java-source.net/open-source/finance/cleansheets
[6] http://sourceforge.net/projects/csheets/files/csheets/1.4b/
[7] http://java-source.net/open-source/finance/cleansheets

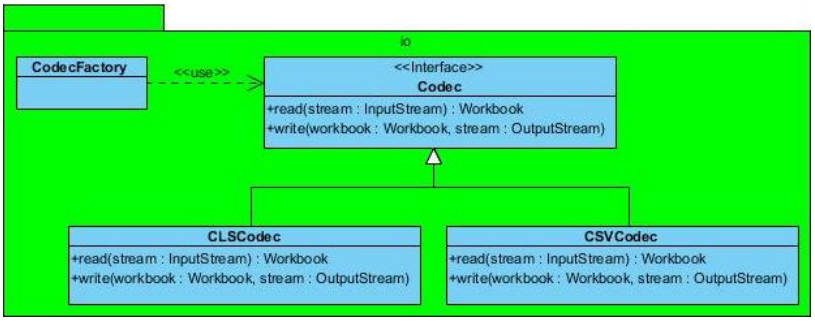

Figura 5.10: *Package io da aplicação CleanSheets*

(Workbook), e carregar de disco para o programa uma folha de cálculo previamente modificada e guardada.

Além dos packages referidos, existem outros, sendo alguns deles subpackages de core, por exemplo, entre outros. São responsáveis por implementar o *parsing* e processamento das fórmulas suportadas, por exemplo, e também pelas classes relacionadas com a interface do CleanSheets. No entanto, para o que é necessário para este caso de estudo é suficiente trabalhar sobre os packages já analisados.

Assim, uma vez que este programa é bastante mais complexo que o TEO, e para não estar a fazer muitas modificações na sua arquitectura, optou-se por trabalhar sobre o package io, que é o package mais simples de todo o CleanSheets, uma vez que as suas classes apenas suportam as operações de leitura e escrita, usando *streams*.

Desta forma, olhando para as classes de io e para o correspondente código, surgiu a ideia de aplicar o padrão de comportamento Strategy, que serve para definir comportamentos diferentes, para um mesmo método, usando para isso um interface e várias implementações desse interface, onde cada uma delas redefine os métodos da forma desejada. É semelhante ao que está a acontecer com o interface Codec, onde as classes que o implementam (CLSCodec e CSVCodec) definem os métodos de read(stream : InputStream) : Workbook e write(workbook : Workbook, stream : OutputStream) de forma diferente, conforme pode ser visto em seguida no código de Codec, CLSCodec e CSVCodec, representado nas Transcrições 5.12, 5.13 e 5.14, respectivamente.

```
1 package csheets.io;
2 import java.io.IOException;
3 import java.io.InputStream;
4 import java.io.OutputStream;
5 import csheets.core.Workbook;
6 /**
7 * An interface for classes capable of reading and writing
      workbooks.
8 * @author Einar Pehrson
9 */
10 public interface Codec {
11
```

```
12      /**
13      * Reads a workbook from the given input stream.
14      * @param stream the input stream from which the workbook
            should be read
15      * @throws IOException if the workbook could not be read
            correctly
16      * @throws ClassNotFoundException If the class of a serialized
            object could not be found
17      */
18      public Workbook read(InputStream stream) throws IOException,
            ClassNotFoundException;
19
20      /**
21      * Writes a workbook to the given output stream.
22      * @param stream the output stream to which the workbook should
            be written
23      * @throws IOException if the workbook could not be written
            correctly
24      */
25      public void write(Workbook workbook, OutputStream stream)
            throws IOException;
26  }
```

Transcrição 5.12: *Codec.java*

```
1  package csheets.io;
2  import java.io.IOException;
3  import java.io.InputStream;
4  import java.io.ObjectInputStream;
5  import java.io.ObjectOutputStream;
6  import java.io.OutputStream;
7  import csheets.core.Workbook;
8  import csheets.ext.ExtensionManager;
9  /**
10 * A codec for the native CleanSheets format that uses Java
       Serialization.
11 * @author Einar Pehrson
12 */
13 public class CLSCodec implements Codec {
14
15      /**
16      * Creates a new CleanSheets codec.
17      */
18      public CLSCodec() {}
19
20
21      public Workbook read(InputStream stream) throws IOException,
            ClassNotFoundException {
22          ObjectInputStream ois = new DynamicObjectInputStream(
                stream, ExtensionManager.getInstance().getLoader());
23          return (Workbook)ois.readObject();
24      }
25
26      public void write(Workbook workbook, OutputStream stream)
            throws IOException {
27          ObjectOutputStream oos = new ObjectOutputStream(stream);
28          oos.writeObject(workbook);
29          oos.flush();
30      }
31 }
```

Transcrição 5.13: *CLSCodec.java*

```
1  package csheets.io;
2  import java.io.BufferedReader;
3  import java.io.BufferedWriter;
4  import java.io.IOException;
5  import java.io.InputStream;
6  import java.io.InputStreamReader;
7  import java.io.OutputStream;
8  import java.io.OutputStreamWriter;
9  import java.io.PrintWriter;
10 import java.io.Reader;
11 import java.util.LinkedList;
12 import java.util.List;
13 import csheets.core.Spreadsheet;
14 import csheets.core.Workbook;
15
16 /**
17 * A codec for comma-separated files.
18 * @author Einar Pehrson
19 */
20 public class CSVCodec implements Codec {
21     /** The string used to separate the content of different cells
          */
22     public static final String SEPARATOR = ";";
23
24     /**
25     * Creates a new CSV codec.
26     */
27     public CSVCodec() {}
28
29     public Workbook read(InputStream stream) throws IOException {
30         // Wraps stream
31         Reader streamReader = new InputStreamReader(stream);
32         BufferedReader reader = new BufferedReader(streamReader);
33         // Reads content of rows
34         String line;
35         int columns = 0;
36         List<String[]> rows = new LinkedList<String[]>();
37         while ((line = reader.readLine()) != null) {
38             String[] row = line.split(SEPARATOR);
39             rows.add(row);
40             if (row.length > columns)
41                 columns = row.length;
42         }
43         // Builds content matrix
44         String[][] content = new String[rows.size()][columns];
45         int i = 0;
46         for (String[] row : rows)
47             content[i++] = row;
48         // Frees resources
49         reader.close();
50         streamReader.close();
51         stream.close();
52         return new Workbook(content);
53     }
54
55     public void write(Workbook workbook, OutputStream stream)
           throws IOException {
56         System.out.println("Writing!");
57         // Wraps stream
58         PrintWriter writer = new PrintWriter(new BufferedWriter(
               new OutputStreamWriter(stream)));
```

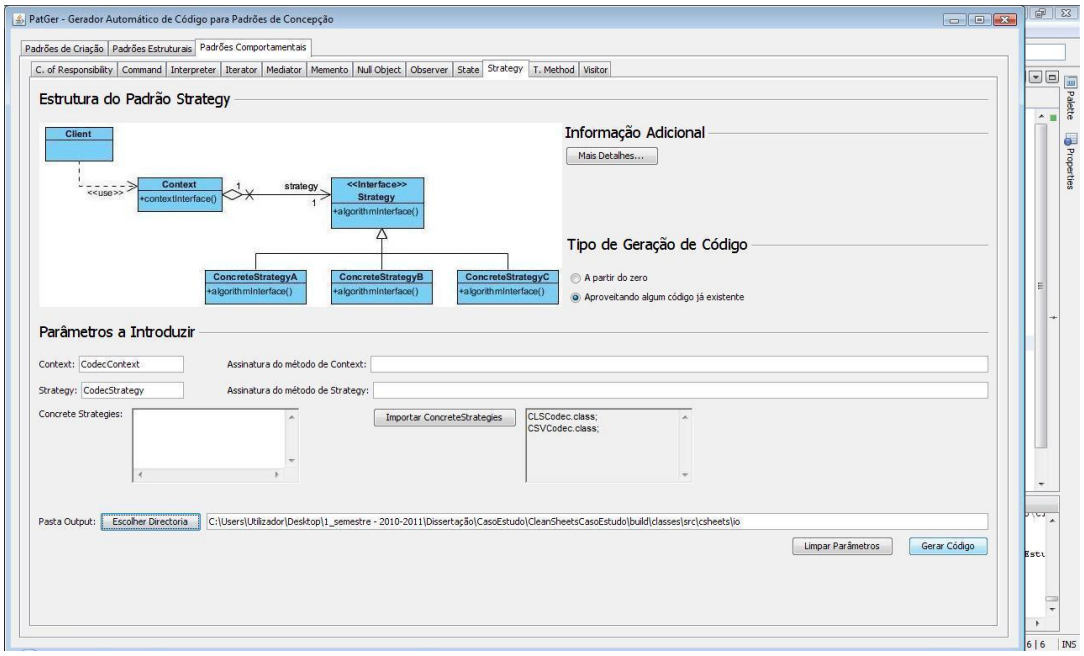

Figura 5.11: *Utilização do PatGer para o Strategy na aplicação CleanSheets*

```
59          // Writes content of rows
60          Spreadsheet sheet = workbook.getSpreadsheet(0);
61          for(int row = 0; row < sheet.getRowCount(); row++) {
62              for(int column = 0;  column < sheet.getColumnCount();
                    column++)
63                  if(column + 1 < sheet.getColumnCount())
64                      writer.print(sheet.getCell(column,row).
                            getContent() + SEPARATOR);
65                  if(row + 1 < sheet.getRowCount())
66                      writer.println();
67          }
68          // Frees resources
69          writer.close();
70          stream.close();
71          System.out.println("Done!");
72      }
73  }
```

Transcrição 5.14: *CSVCodec.java*

Em seguida, na Figura 5.11, encontra-se uma imagem da utilização do PatGer para a geração de código para o padrão comportamental Strategy, utilizando como input os ficheiros *.class* correspondentes às classes Java CLSCodec e CSVCodec.

Ao nível do código gerado pelo PatGer, para este exemplo concreto, apenas interessa mostrar a classe CodecContext, dado que o código dos ficheiros usados como input não foi alterado. No caso do interface CodecStrategy, como o programa apenas coloca nessa classe as assinaturas dos métodos presentes nos ficheiros *.class* usados como input, acaba também por não ser alterado;

a única coisa que mudou foi o seu nome, tal como se pode ver nas opções introduzidas, na Figura 5.11. Em seguida, na Transcrição 5.15, pode ser visto o código para CodecContext, gerado pelo PatGer.

```
1 /**
2 *
3 * Pattern Strategy
4 * CodecContext gerada automaticamente pela aplicacao
5 *
6 */
7 public class CodecContext{
8     private CodecStrategy codecStrategy;
9
10    public CodecContext(CodecStrategy codecStrategy){
11        this.codecStrategy = codecStrategy;
12    }
13
14    public CodecStrategy getCodecStrategy(){
15        return this.codecStrategy;
16    }
17
18    public void setCodecStrategy(CodecStrategy codecStrategy){
19        this.codecStrategy = codecStrategy;
20    }
21
22    public Workbook read(InputStream stream) throws IOException,
          ClassNotFoundException {
23        return this.codecStrategy.read(stream);
24    }
25
26    public void write(Workbook workbook, OutputStream stream)
          throws IOException {
27        this.codecStrategy.write(workbook, stream);
28    }
29 }
```

Transcrição 5.15: *CodecContext.java*

Com esta abordagem apenas se pretende tornar mais clara a estrutura de classes presente no package io, evidenciando a existência do padrão de comportamento Strategy, recorrendo para isso ao PatGer. Na verdade, apenas faltava incorporar a classe CodecContext para que, juntamente com as outras classes, se estivesse na presença deste padrão de concepção.

Resta agora modificar o código de CleanSheets para que, em vez de utilizar directamente os métodos de read(stream : InputStream) : Workbook ou de write(workbook : Workbook, stream : OutputStream), numa instância de Codec fornecida por CodecFactory, passe a utilizar essa mesma instância como parâmetro de CodecContext, e aí sim é que serão invocados os métodos read(stream : InputStream) : Workbook ou write(workbook : Workbook, stream : OutputStream).

Pode agora ser visto, na Figura 5.12, a nova arquitectura de classes para o package io, já com a classe CodecContext nele presente.

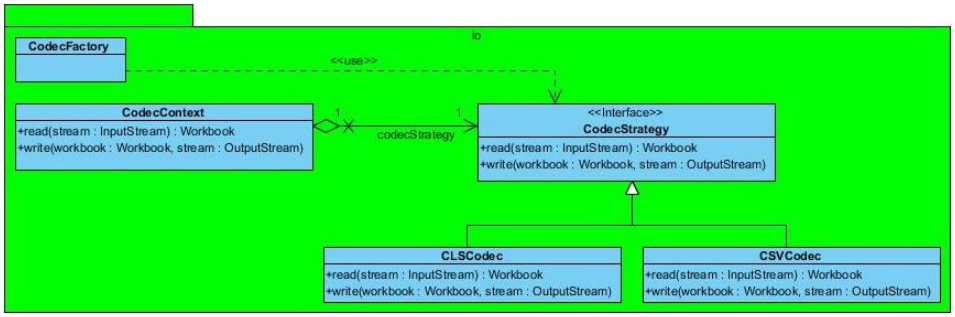

Figura 5.12: *Package io modificado da aplicação CleanSheets*

5.3 Discussão dos Casos de Estudo

Nesta secção será efectuada uma análise crítica aos dois casos de estudo apresentados nas secções anteriores.

O primeiro caso de estudo englobou uma aplicação Java de pequena/média dimensão, o TEO, sendo que o PatGer foi utilizado, com sucesso, quer para a modificação de uma parte da estrutura de classes do TEO, quer para lhe acrescentar novas funcionalidades.

Sobre a modificação de parte da estrutura de classes do TEO, nomeadamente nas classes do package Dados, importa dizer que o PatGer alterou essa estrutura com a introdução de uma *pool* de ligações à base de dados (padrão Object Pool); além disso, com a posterior geração de um Facade para encapsular as várias classes que lidavam com a base de dados, tornou-se mais simples a comunicação com o package Dados.

Como a geração de código para a implementação da *pool* e para a implementação do Facade foi feita de forma automática pelo PatGer, pode-se concluir que simplificou bastante essa tarefa. Caso tivesse sido efectuada sem se recorrer ao PatGer, teria de ser programada, de forma manual, toda a *pool* de ligações à base de dados, juntamente com o comportamento associado aos métodos nela presentes.

Já para o Facade, caso a sua implementação fosse efectuada sem utilizar o PatGer, seria necessário saber todas as assinaturas dos métodos das classes incorporadas no Facade, e replicar esses métodos no Facade, fazendo-os interagir com as respectivas classes incorporadas. Portanto, com o PatGer foi bastante simples modificar a estrutura de classes do package Dados do TEO, sem se escrever nenhum código de forma manual.

Com os exemplos de utilização do PatGer entretanto analisados, é possível concluir que esta aplicação de software pode ajudar a desenvolver código Java para aplicações já existentes, fazendo-o de uma forma simples e rápida. Pode por isso ser bastante útil quando for necessário acrescentar código a um programa já desenvolvido, minimizando o tempo gasto para o fazer.

Quanto às novas funcionalidades acrescentadas ao TEO, após terem sido

programadas as classes Java Tempo e Euromilhoes, com recurso ao PatGer foi possível gerar, de forma automática, toda a estrutura e código associados ao padrão Factory Method, englobando as classes acima referidas, ficando a fazer parte de um interface. Assim, ficou mais simples a interacção com estas novas classes. Caso não tivesse sido utilizado o PatGer para gerar automaticamente o código relativo ao Factory Method, seria necessário fazê-lo manualmente, tendo que se definir o comportamento para todos os métodos pertencentes às classes do Factory Method.

Relativamente a este último exemplo descrito, se se considerar as novas funcionalidades acrescentadas ao TEO como sendo uma nova aplicação, então pode-se afirmar que o PatGer pode ser utilizado no desenvolvimento de novas aplicações Java, contribuindo com código gerado automaticamente para a implementação de padrões de concepção. Desta forma, o PatGer pode também ser uma ferramenta importante na programação de novas aplicações Java, reduzindo o tempo necessário para o seu desenvolvimento.

O segundo caso de estudo teve como base uma aplicação Java de maior dimensão que o TEO, que foi o CleanSheets[8], tendo sido utilizado o PatGer para a modificação de uma pequena parte da sua estrutura de classes.

A mudança ocorreu nas classes do package io, onde, recorrendo ao PatGer, se implementou o padrão de comportamento Strategy, sobre as classes CLSCodec e CSVCodec, aproveitando as assinaturas dos métodos nelas definidos. Como a geração do código das classes que constituem o Strategy foi feita de forma automática pelo PatGer, esse processo foi bastante simples e rápido. Caso se optasse por não utilizar o PatGer para a implementação do Strategy, então seria necessário programar, manualmente, todo o código das classes e métodos que compõem a sua estrutura, sendo portanto uma tarefa mais complicada e demorada.

Este exemplo demonstra que o PatGer pode também ser utilizado sobre aplicações Java de maior dimensão, gerando código de forma automática para os padrões de concepção, podendo por isso simplificar, de forma significativa, a manutenção/modificação de aplicações deste tipo, reduzindo o tempo gasto nesse processo.

[8] http://java-source.net/open-source/finance/cleansheets

Capítulo 6

Conclusões

6.1 Síntese

Nesta secção vai ser efectuada uma breve síntese do que foi desenvolvido durante a elaboração desta dissertação, analisando brevemente cada um dos capítulos existentes.

Assim, no capítulo 1 foi feita a introdução ao tema da dissertação, que é o desenvolvimento de uma aplicação de software para a geração automática de código para os padrões de concepção, identificados em [Gamma et al., 1995], numa linguagem de programação orientada aos objectos.

Foi também referida a motivação para o desenvolvimento dessa aplicação, o PatGer, que passa essencialmente por evitar a codificação de componentes semelhantes (padrões de concepção), durante a programação de qualquer aplicação de software. Esses componentes passam a ser gerados de forma automática pelo PatGer. Além disso, foram também apresentados vários objectivos a atingir durante a realização da dissertação.

O capítulo 2 abordou o estado da arte associado ao tema da dissertação, isto é, os conceitos teóricos fundamentais para a sua realização, como os vários padrões de concepção catalogados em [Gamma et al., 1995]. Foram também apresentados outros padrões de software, nomeadamente os propostos em [Buschmann et al., 1996] e em [Brown et al., 1998], servindo para mostrar outros pontos de vista, diferentes da abordagem tomada nesta dissertação.

Também durante este capítulo foi feita uma análise da aplicabilidade actual de alguns dos padrões de concepção de [Gamma et al., 1995], de forma a demonstrar que podem ser utilizados no desenvolvimento de soluções de software algo recentes. No final deste capítulo foram ainda referidas, de forma breve, algumas aplicações de software semelhantes ao PatGer, e que foram úteis para retirar ideias para o seu desenvolvimento.

No capítulo 3 foi abordado o problema relacionado com a dissertação, ou seja, os padrões de concepção [Gamma et al., 1995] para os quais o PatGer

trata de gerar o respectivo código. Foram escolhidos alguns padrões para os quais foi feita uma breve descrição das situações onde podem ser aplicados, apresentando uma ilustração respeitante à sua estrutura, com as várias classes e métodos que o constituem, além de um exemplo de utilização.

No final deste terceiro capítulo foi apresentado o Modelo do Domínio desenvolvido, juntamente com a descrição de cada uma das entidades nele representadas, e também dos relacionamentos entre cada uma delas. Foram ainda indicados os requisitos funcionais e não funcionais que o PatGer teve de cumprir.

No capítulo 4 foi descrita a aplicação de software programada, ou seja, toda a especificação subjacente ao PatGer, juntamente com o processo relacionado com a sua implementação. Assim, neste capítulo foram apresentados alguns esboços elaborados para a interface do PatGer, bem como alguns diagramas de sequência para explicar o que acontece quando o utilizador interage com o programa, ao nível dos métodos invocados. Foram também descritos alguns diagramas de classes elaborados na etapa de especificação do PatGer, para melhor se entender a lógica de negócio presente no programa.

Também durante este capítulo foram referidas algumas decisões tomadas antes de se passar à codificação do PatGer, nomeadamente a opção de desenvolver duas versões do programa, sendo uma delas um ficheiro executável *.jar* e a outra um plugin para o Ambiente de Desenvolvimento Integrado NetBeans. Além disso foi incluído um breve exemplo de utilização da aplicação elaborada, juntamente com informação importante sobre como acrescentar novas funcionalidades ao PatGer.

No capítulo 5 foram apresentados dois casos de estudo, ou seja, exemplos reais de utilização do PatGer, de modo a provar que a aplicação desenvolvida pode ser utilizada, com bons resultados, num contexto real.

Desta forma, em primeiro lugar, foi tida em consideração uma aplicação de pequena/média dimensão, sobre a qual se recorreu ao PatGer para modificar um pouco a sua arquitectura de classes, e também para acrescentar novas funcionalidades. Em seguida, foi utilizada uma aplicação de software de maior dimensão, onde também se recorreu ao PatGer para modificar a arquitectura de classes de uma pequena parte dessa aplicação.

Para terminar, resta dizer que os anexos elaborados apresentam informação relacionada com os padrões de concepção não abordados no terceiro capítulo, além de mais alguns esboços da interface do PatGer, e também mais diagramas de sequência e de classes, juntamente com algumas imagens relativas ao PatGer. Trata-se de informação complementar que pode também ser consultada para uma melhor compreensão da dissertação desenvolvida.

6.2 Discussão

No decorrer desta secção será efectuada uma análise crítica sobre os aspectos mais relevantes do trabalho desenvolvido durante a presente dissertação.

Assim, há que referir que foi muito importante a pesquisa relacionada com o tema da dissertação, tendo permitido não só aprofundar conhecimentos sobre os padrões de concepção [Gamma et al., 1995], como também ficar a conhecer padrões orientados à arquitectura de software [Buschmann et al., 1996] e ainda *antipatterns* [Brown et al., 1998]. Além disto, as ferramentas encontradas, e capazes de gerar código para os padrões de concepção, possibilitaram ter uma noção mais concreta da aplicação de software programada no decurso da dissertação.

A pesquisa mais aprofundada sobre os padrões de concepção [Gamma et al., 1995] contribuiu de forma decisiva para saber ao certo a sua estrutura de classes, juntamente com os métodos fundamentais para a sua correcta implementação no PatGer. Assim, uma vez terminada a pesquisa foi possível entrar na fase de especificação do PatGer, onde se começou por desenhar o Modelo do Domínio e definir os requisitos funcionais e não funcionais a respeitar.

Durante a fase de desenvolvimento do PatGer há que salientar a utilidade dos esboços, diagramas de sequência e diagramas de classes desenhados durante a fase de especificação, permitindo que a sua programação fosse mais simples, na medida em que como já estava tudo especificado, apenas foi necessário seguir essa mesma especificação.

É importante mencionar também a decisão de desenvolver duas versões do PatGer, procurando maximizar os seus cenários de utilização. Assim, o ficheiro executável *.jar* permite que o PatGer seja independente da plataforma, ao passo que o plugin para o NetBeans possibilita um acesso mais rápido ao PatGer, para quem recorra a esse IDE.

Relativamente aos dois casos de estudo desenvolvidos, interessa dizer que serviram para provar a aplicabilidade do PatGer em aplicações já existentes, tornando possível a modificação da sua arquitectura de classes e também a adição de novas funcionalidades, recorrendo aos padrões de concepção.

Quanto aos objectivos inicialmente propostos para a dissertação, há que referir que podem ser considerados cumpridos, pois os tópicos neles presentes foram sendo abordados durante os vários capítulos elaborados. Além disso, o objectivo principal da dissertação foi cumprido, visto que foi desenvolvido o PatGer, que é uma aplicação de software que gera o código, de forma automática, para os padrões de concepção, de acordo com o pretendido.

Por fim, pode-se concluir que o PatGer vem simplificar a utilização dos padrões de concepção, durante o desenvolvimento de uma aplicação de software, uma vez que permite que o código associado aos padrões seja gerado de forma automática e parametrizada, de acordo com as necessidades do utilizador.

Espera-se então que o PatGer possa vir a ser uma peça importante durante o desenvolvimento de novas aplicações de software.

6.3 Trabalho Futuro

Nesta secção são referidas algumas ideias sobre o que poderá ser efectuado, como trabalho futuro, ao nível da aplicação de software elaborada no decorrer desta dissertação, tendo por objectivo melhorar o programa final.

Uma ideia que pode vir a ser desenvolvida, como trabalho futuro, está relacionada com a apresentação da estrutura de classes para o padrão de concepção, escolhido pelo utilizador, já com a inclusão dos parâmetros por ele fornecidos. Isto é, em vez do PatGer mostrar apenas uma figura genérica, com as classes que formam a estrutura de um dado padrão, poderá passar a mostrar, além disso, uma nova figura com as classes, e respectivos métodos, já definidos pelo utilizador, antes de fazer a geração de código associada.

Esta funcionalidade acrescentada permitirá, ao utilizador, ter uma noção mais clara de como vai ficar a estrutura de classes para o padrão de concepção escolhido, tendo em consideração os parâmetros por ele definidos.

Em relação à geração de código para os padrões de concepção, uma vez que o PatGer apenas o faz na linguagem de programação Java, interessa aqui salientar a possibilidade de o vir a fazer noutras linguagens de programação orientadas aos objectos. Desta forma, uma ideia para trabalho futuro passa por tornar o PatGer capaz de gerar o código para os padrões de concepção, por exemplo, em C#.

O programa final pode continuar a ser disponibilizado como um ficheiro executável *.jar*, mantendo-se assim independente da plataforma. Pondo em prática esta sugestão, certamente que a utilidade do PatGer pode vir a aumentar, pois passa a gerar o código para os padrões de concepção em mais do que uma linguagem de programação orientada aos objectos.

Uma outra ideia para trabalho futuro diz respeito à forma como o PatGer é disponibilizado para o utilizador. Por agora, existem duas versões, sendo uma delas um ficheiro executável *.jar* e a outra um plugin para o Ambiente de Desenvolvimento Integrado NetBeans. Assim, de futuro poderá ser desenvolvida uma versão *web* do PatGer, continuando o programa a ser independente da plataforma. Com esta nova versão do PatGer deixa de ser necessário que o utilizador possua o ficheiro executável *.jar*, passando a poder interagir com a aplicação através de um *browser*.

Bibliografia

Aubert, O. and Beugnard, A. 2001. Adaptive strategy design pattern.

Avgeriou, P. and Zdun, U. 2005. Architectural patterns revisited - a pattern language. In *EuroPLoP*, A. Longshaw and U. Zdun, Eds. UVK - Universitaetsverlag Konstanz, 431–470.

Bosch, J. 1998. Design patterns as language constructs. *JOOP 11,* 2, 18–32.

Brown, W. J., Malveau, R. C., McCormick III, H. W., and Mowbray, T. J. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* Wiley.

Budinsky, F. J., Finnie, M. A., Vlissides, J. M., and Yu, P. S. 1996. Automatic code generation from design patterns. *IBM Systems Journal 35,* 2, 151–171.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. 1996. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley, Chichester, UK.

Büttner, F., Radfelder, O., Lindow, A., and Gogolla, M. 2004. Digging into the visitor pattern. In *Proc. IEEE 16th Int. Conf. Software Engineering and Knowlege Engineering (SEKE2004). IEEE, Los Alamitos.* 135–141.

Carvalho, S. T. 2002. Um design pattern para configuração de arquiteturas de software. M.S. thesis, Universidade Federal Fluminense, Niterói, RJ, Brasil.

Coad, P., North, D., and Mayfield, M. 1995. *Object models: strategies, patterns, applications.* Yourdon Press, Upper Saddle River, NJ, USA.

Dong, J., Alencar, P., and Cowan, D. 2000. Ensuring structure and behavior correctness in design composition. In *Engineering of Computer Based Systems, 2000. (ECBS 2000) Proceedings. Seventh IEEE International Conference and Workshopon the.* 279 –287.

Duffy, E. B., Gibson, J. P., and Malloy, B. A. 2003. Applying the decorator pattern for profiling object-oriented software. In *IWPC*. IEEE Computer Society, 84–93.

Ellis, B., Stylos, J., and Myers, B. 2007. The factory pattern in api design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*. ICSE '07. IEEE Computer Society, Washington, DC, USA, 302–312.

Ferenc, R., Beszedes, A., Fulop, L., and Lele, J. 2005. Design pattern mining enhanced by machine learning. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. 295 – 304.

Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. M. 1993. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP*, O. Nierstrasz, Ed. Lecture Notes in Computer Science, vol. 707. Springer, 406–431.

Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. M. 1995. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, United States of America.

Gestwicki, P. V. 2007. Computer games as motivation for design patterns. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*. SIGCSE '07. ACM, New York, NY, USA, 233–237.

Hammouda, I. and Harsu, M. 2004. Documenting maintenance tasks using maintenance patterns. *Software Maintenance and Reengineering, European Conference on 0*, 37.

Johnson, R. E. 1997. Frameworks = (components + patterns). *Commun. ACM 40,* 10, 39–42.

Kampffmeyer, H. and Zschaler, S. 2007. Finding the pattern you need: The design pattern intent ontology. In *Model Driven Engineering Languages and Systems*, G. Engels, B. Opdyke, D. Schmidt, and F. Weil, Eds. Lecture Notes in Computer Science, vol. 4735. Springer Berlin / Heidelberg, 211–225.

Kärpijoki, V. 2001. Antipatterns.

Khomh, F., Vaucher, S., Guéhéneuc, Y.-G., and Sahraoui, H. 2009. A bayesian approach for the detection of code and design smells. *Quality Software, International Conference on 0*, 305–314.

King, G., Bauer, C., Andersen, M. R., Bernard, E., and Ebersole, S. 2010. *HIBERNATE - Relational Persistence for Idiomatic Java*, 3.5.1 ed.

KNIESEL, G., RHO, T., AND HANENBERG, S. 2004. Evolvable pattern implementations need generic aspects. In *Proc. of ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution at ECOOP 2004*. 111–126.

KRUEGER, C. W. 1992. Software reuse. *ACM Comput. Surv. 24,* 2, 131–183.

LAPLANTE, P., HOFFMAN, R. R., AND KLEIN, G. 2007. Antipatterns in the creation of intelligent systems. *Intelligent Systems, IEEE 22,* 1 (January-February), 91–95.

LAPLANTE, P. A. 2004. Staying clear of boiling-frog syndrome [work environment]. *IT Professional 6,* 2 (March-April), 56–58.

LAUDER, A. AND KENT, S. 1998. Precise visual specification of design patterns. In *ECOOP'98 - Object-Oriented Programming*, E. Jul, Ed. Lecture Notes in Computer Science, vol. 1445. Springer Berlin / Heidelberg, 114–134. 10.1007/BFb0054089.

MAK, J. K. H., CHOY, C. S. T., AND LUN, D. P. K. 2004. Precise modeling of design patterns in uml. In *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. IEEE Computer Society, Washington, DC, USA, 252–261.

MOHA, N., GUÉHÉNEUC, Y.-G., LE MEUR, A.-F., AND DUCHIEN, L. 2008. A domain analysis to specify design defects and generate detection algorithms. In *Fundamental Approaches to Software Engineering*, J. Fiadeiro and P. Inverardi, Eds. Lecture Notes in Computer Science, vol. 4961. Springer Berlin / Heidelberg, 276–291.

MOHA, N., GUÉHÉNEUC, Y.-G., AND LEDUC, P. 2006. Automatic generation of detection algorithms for design defects. *Automated Software Engineering, International Conference on 0*, 297–300.

NOBLE, J. 2000. Iterators and encapsulation. In *Technology of Object-Oriented Languages, 2000. TOOLS 33. Proceedings. 33rd International Conference on*. 431 –442.

OBJECT MANAGEMENT GROUP, O. 2006. *Unified Modeling Language: Infrastructure*.

PALSBERG, J. AND JAY, C. 1998. The essence of the visitor pattern. In *Computer Software and Applications Conference, 1998. COMPSAC '98. Proceedings. The Twenty-Second Annual International*. 9 –15.

PIVETA, E. K. AND ZANCANELLA, L. C. 2003. Observer pattern using aspect-oriented programming.

Pohl, H. W. and Gerlach, J. 2003. Using the bridge design pattern for osgi service update. In *Proceedings of the 8th European Conference on Pattern Languages of Programs*. Irsee, Germany.

Reiss, S. P. 2000. Working with patterns and code. In *HICSS*.

Rogers, J. and Pheatt, C. 2009. Integrating antipatterns into the computer science curriculum. *J. Comput. Small Coll. 24*, 183–189.

Seemann, J. and von Gudenberg, J. W. 1998. Pattern-based design recovery of java software. In *SIGSOFT FSE*. 10–16.

Suzuki, J. and Yamamoto, Y. 1998. The reflection pattern in the immune system. In *Proc. of OOPSLA '98, workshop on Non-Software Examples of Software Architecture*.

Tokuda, L. and Batory, D. 1995. Automated software evolution via design pattern transformations. In *Proceedings of the 3rd International Symposium on Applied Corporate Computing*.

van Gurp, J. and Bosch, J. 2002. Design erosion: problems and causes. *Journal of Systems and Software 61,* 2, 105 – 119.

Vinoski, S. 2002. Chain of responsibility. *Internet Computing, IEEE 6,* 6 (November-December), 80 – 83.

Welicki, L., Lovelle, J. M. C., and Aguilar, L. J. 2006a. Meta-specification and cataloging of software patterns with domain specific languages and adaptive object models. In *EuroPLoP*, U. Zdun and L. B. Hvatum, Eds. UVK - Universitaetsverlag Konstanz, 359–392.

Welicki, L., Lovelle, J. M. C., and Aguilar, L. J. 2006b. Patterns meta-specification and cataloging: towards knowledge management in software engineering. In *OOPSLA Companion*, P. L. Tarr and W. R. Cook, Eds. ACM, 679–680.

Wirfs-Brock, R. 2007. Toward design simplicity. *Software, IEEE 24,* 2 (March-April), 9 –11.

Zdun, U., Kircher, M., and Völter, M. 2004. Remoting patterns. *IEEE Internet Computing 8,* 6, 60–68.

Zimmer, W. 1994. Relationships between design patterns. In *PATTERN LANGUAGES OF PROGRAM DESIGN*. Addison-Wesley, 345–364.

Anexo A

Padrões de Criação

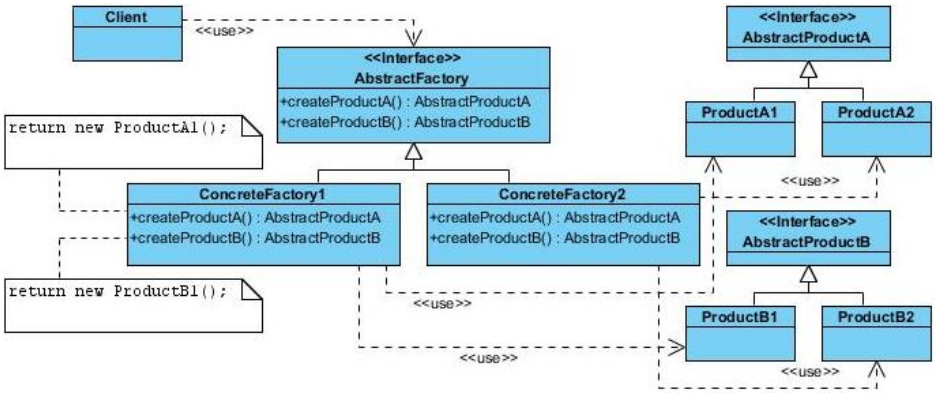

Figura A.1: *Estrutura do Abstract Factory (adaptada de [Gamma et al., 1995])*

Abstract Factory

O padrão Abstract Factory oferece um interface para que sejam criados objectos genéricos do tipo produto. Remove a dependência entre as classes concretas dos produtos e os objectos do tipo produto, criados pelo cliente [Gamma et al., 1993].

Na Figura A.1 podemos ver a estrutura associada ao padrão Abstract Factory. Como se verifica pela sua observação, vemos que a classe AbstractFactory (é uma classe totalmente abstracta, ou seja, um interface) contém os métodos abstractos para a criação dos produtos A e B, neste caso. As classes ConcreteFactory1 e ConcreteFactory2 implementam os métodos da classe AbstractFactory, de acordo com os produtos concretos a instanciar a partir delas. Assim, um produto instanciado a partir de ConcreteFactory1 será um produto A1 ou B1, de acordo com o método utilizado.

Assim, quando o cliente[1] pretender, por exemplo, um produto do tipo A, o sistema dar-lhe-á um produto ou A1 ou A2, mas o cliente não se apercebe disso, pois para ele apenas lhe foi dado um produto do tipo A. Isto acontece porque as classes AbstractProductA eAbstractProductB são apenas interfaces; as implementações concretas dos produtos da fábrica abstracta são os objectos ProductA1, ProductA2, ProductB1 e ProductB2, neste exemplo. Quanto ao cliente, apenas comunica com o interface AbstractFactory, invocando um dos métodos lá definidos, e recebe como resultado o respectivo interface AbstractProduct, desconhecendo o seu subtipo.

Como um exemplo da utilização deste padrão podemos considerar o descrito em [Gamma et al., 1995].

Como se verifica pela observação da Figura A.2, vemos que a classe Abstract Factory é agora uma WidgetFactory, e tem duas subclasses que implementam os métodos nela definidos. Os métodos são createScrollBar() e createWindow(). Agora, no caso de o cliente ser um cliente Windows, ao solicitar

[1] pode ser uma classe do programa, ou mesmo uma aplicação externa

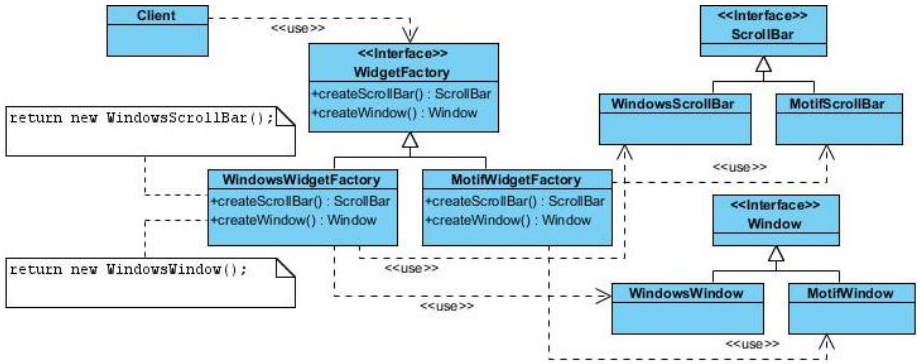

Figura A.2: *Exemplo de Utilização do Abstract Factory (adaptada de [Gamma et al., 1995])*

uma Window à WidgetFactory, vai-lhe ser dada uma WindowsWindow (porém o cliente vai vê-la como o interface Window, desconhecendo o seu subtipo), instanciada com o método createWindow() da classe WindowsWidgetFactory. De uma forma semelhante obterá uma ScrollBar.

Assim sendo, como explicado em [Tokuda and Batory, 1995], dado que uma combinação de, por exemplo, uma WindowsScrollBar com uma MotifWindow não irá funcionar muito bem, é importante que o cliente não venha a sofrer este problema. Então, para o evitar, o padrão Abstract Factory serve perfeitamente, pois faz com que o cliente apenas se dirija à classe WidgetFactory, e a partir dela obterá as instâncias de Window e ScollBar que precisar, com a garantia que funcionarão bem em conjunto. Isto é, apenas poderá ter WindowsWidgets ou MotifWidgets.

Então, a partir deste exemplo constata-se que o Abstract Factory é útil quando pretendemos ter um sistema independente da forma como os seus produtos são criados e compostos, ou ainda quando queremos que um sistema seja configurado para mais do que uma família de produtos, e finalmente quando se quer fornecer uma biblioteca de classes de produtos, mas apenas queremos dar a conhecer a sua interface, ou seja, a sua API [Gamma et al., 1995].

Quanto à capacidade para passar a criar os produtos já existentes (Window e ScrollBar) para outra plataforma (Linux, por exemplo), apenas teremos de programar uma subclasse de WidgetFactory, a classe LinuxWidgetFactory, neste caso, e concretizar nessa nova classe os métodos abstractos de WidgetFactory. É um processo bastante simples, sendo uma vantagem da utilização do padrão Abstract Factory.

Quanto à capacidade para suportar novos tipos de produtos com interfaces diferentes (por exemplo, uma SideBar), para tornar isto possível, teríamos de adicionar um novo método abstracto na classe WidgetFactory (por exemplo, createSidebar()), e implementar esse novo método em todas as suas subclasses.

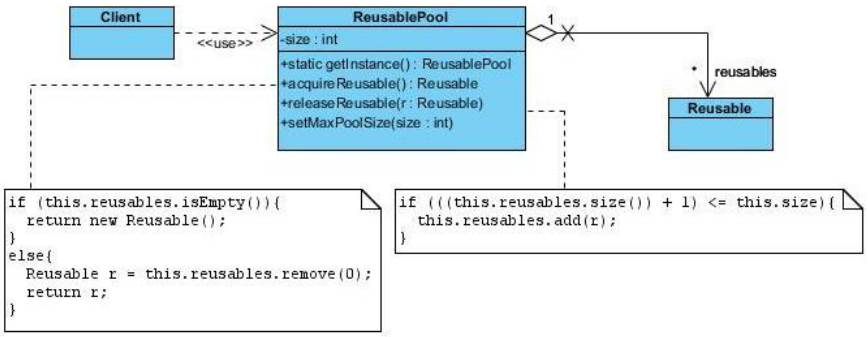

Figura A.3: *Estrutura do Object Pool*

Object Pool

O Padrão Object Pool tem por objectivo melhorar o desempenho de aplicações onde o custo de criar instâncias de uma classe é elevado, e onde essas instâncias estão muito frequentemente a ser criadas. No entanto, as instâncias criadas são utilizadas por pouco tempo pela aplicação[2].

Assim, recorrendo a *pools* de objectos, o cliente não precisa de estar sempre a criar uma nova instância, pois apenas necessita de pedir à *pool* de objectos uma instância já criada. Na *pool* de objectos ficam algumas instâncias entretanto criadas, e depois de serem utilizadas pelo cliente voltam para a *pool*, para poderem mais tarde ser reutilizadas. No caso de o cliente pedir um objecto e a *pool* estar vazia, então será nesse momento criada uma nova instância e será dada ao cliente[3].

A *pool* de objectos é então responsável por inicializar a *pool* com um número definido de instâncias, e pode ainda ter uma opção de manter a *pool* sempre com esse mesmo número de instâncias, o que faz com que a *pool* cresça até um limite, devido às instâncias que voltam para a *pool* após terem sido utilizadas[4].

Importa ainda referir que a *pool* de objectos deverá ser implementada de acordo com o padrão Singleton (ver mais à frente), permitindo assim que exista apenas uma instância da *pool* de objectos na aplicação.

Assim, através da observação da Figura A.3, vemos que a classe ReusablePool (implementada com o padrão Singleton) contém uma variável reusables, que é o conjunto de instâncias já criadas e que se podem dar ao cliente, quando este solicitar um objecto da *pool*. O método getInstance() devolve ao cliente uma instância (única) de ReusablePool, sendo que o método acquireReusable() lhe devolve um objecto da *pool*. Quando já não necessitar desse objecto, deverá devolvê-lo à *pool*, usando o método releaseReusable(). O método setMaxPoolSize(size : int) permite definir o número máximo de objectos

[2]http://sourcemaking.com/design_patterns
[3]http://sourcemaking.com/design_patterns
[4]http://sourcemaking.com/design_patterns

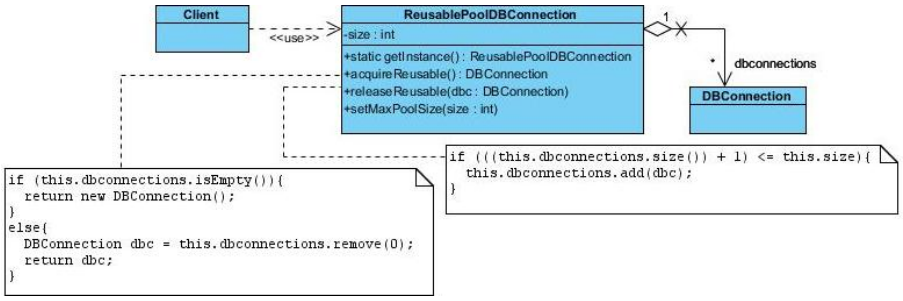

Figura A.4: *Exemplo de Utilização do Object Pool*

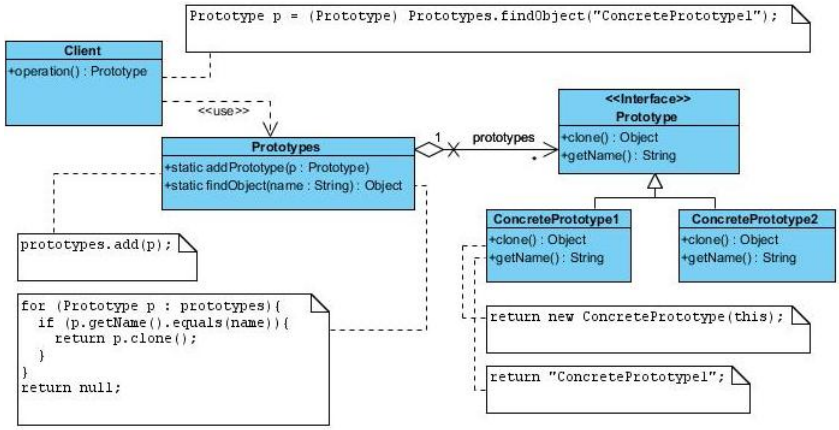

Figura A.5: *Estrutura do Prototype (adaptada de [Gamma et al., 1995])*

a manter na *pool*.

Como exemplo de utilização do padrão Object Pool temos uma *pool* de ligações a uma base de dados. Nesse caso, há sempre um número limitado de ligações à base de dados activas, e guardadas na *pool*. Quando o cliente pretender efectuar um pedido à base de dados, em vez de estar a criar a ligação, apenas precisa de ir buscar uma à *pool*, ganhando assim o tempo necessário para consumar uma nova ligação à base de dados. Este exemplo encontra-se representado na Figura A.4.

Prototype

O padrão de criação Prototype (que pode também ser conhecido por Exemplar [Tokuda and Batory, 1995]) especifica os tipos de objectos a criar partindo de uma instância protótipo, e cria os novos objectos fazendo cópias do protótipo escolhido. Desta forma, o cliente não precisa de utilizar o operador *new* para obter uma nova instância do produto que deseja, mas vai sim utilizar o método clone() a partir do protótipo [Gamma et al., 1995].

Através observação da Figura A.5, vemos que o interface Prototype define

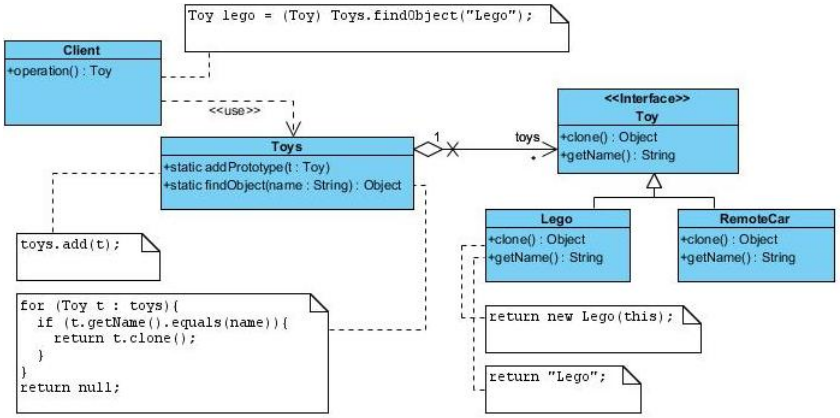

Figura A.6: *Exemplo de Utilização do Prototype (adaptada de [Gamma et al., 1995])*

o método clone(), que será implementado pelas várias subclasses. O interface Prototype serve de supertipo para as instâncias de produtos concretos a criar. Esses produtos estão representados por ConcretePrototype1 e ConcretePrototype2, e em cada um está implementado o respectivo método clone(), que devolve uma cópia de si mesmo.

Vemos também a classe Prototypes que contém um conjunto de objectos do tipo Prototype, em que cada um desses objectos corresponde a uma instância de ConcretePrototype, dentro dos vários existentes na hierarquia de classes (neste exemplo são dois). O cliente tem o método operation(), onde vai chamar o método findObject(name : String) de Prototypes, que lhe vai devolver uma instância (cópia) do protótipo escolhido.

De acordo com o exemplo representado na Figura A.6, existe a classe Toys que contém um conjunto de objectos do tipo Toy, e, disponibiliza o método findObject(name : String) para dar ao cliente uma cópia da instância de Toy que o cliente pretende, de acordo com o nome passado por parâmetro. O interface Toy tem como subclasses Lego e RemoteCar, que implementam os métodos clone() e getName(), que devolvem uma *deep copy* da instância Lego ou RemoteCar, no caso do clone(), e devolvem o nome da instância, no caso de getName().

Singleton

O padrão Singleton (ou também Solitaire [Tokuda and Batory, 1995]) permite que uma dada classe tenha apenas uma única instância, garantindo um ponto de acesso global para essa instância. Deve então ser utilizado quando a aplicação a desenvolver necessita ter uma e só uma instância de um determinado objecto, e que seja globalmente acessível [Gamma et al., 1995].

Além disso, é também importante referir que este padrão deve ser utili-

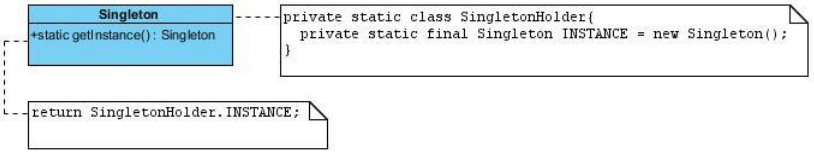

Figura A.7: *Estrutura do Singleton (adaptada de [Gamma et al., 1995])*

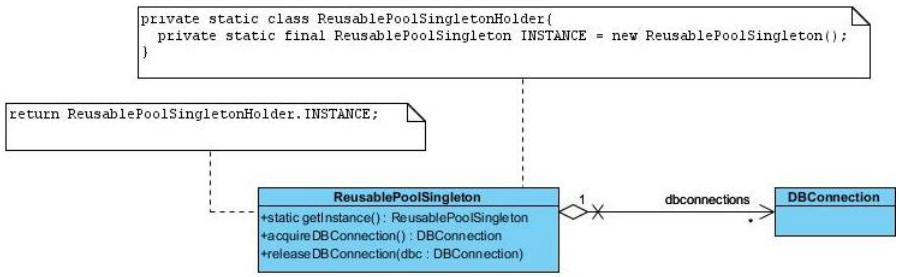

Figura A.8: *Exemplo de Utilização do Singleton (adaptada de [Gamma et al., 1995])*

zado para guardar variáveis globais de uma aplicação, ou, por exemplo, objectos como as fábricas concretas, que entram na estrutura de grande parte deste grupo de padrões de criação [Tokuda and Batory, 1995].

Assim sendo, na Figura A.7, a classe Singleton tem um método público getInstance() que trata de encapsular a inicialização da instância, e que funciona também como ponto global de acesso, e que devolve uma instância desta classe. Os restantes métodos servem para se aceder ao conteúdo da classe Singleton [Gamma et al., 1995].

Como exemplo de utilização do Singleton, pode-se referir a classe que implementa uma *pool* de ligações a uma base de dados. Esta classe deverá então ter apenas uma instância sua presente na aplicação, e providenciar um ponto de acesso global, encapsulando também o código relativo à sua inicialização, ou seja, da criação das várias ligações à base de dados a manter na *pool*. Este exemplo encontra-se representado na Figura A.8.

Através sua observação, é possível verificar que a classe ReusablePoolSingleton contém um conjunto de instâncias do tipo DBConnection, que representam as várias ligações à base de dados. Ao inicializar esta instância única, usando o método getInstance(), são inicializadas também as várias instâncias DBConnection que representam a *pool*. Os restantes métodos servem para dar o comportamento à *pool*.

Anexo B

Padrões Estruturais

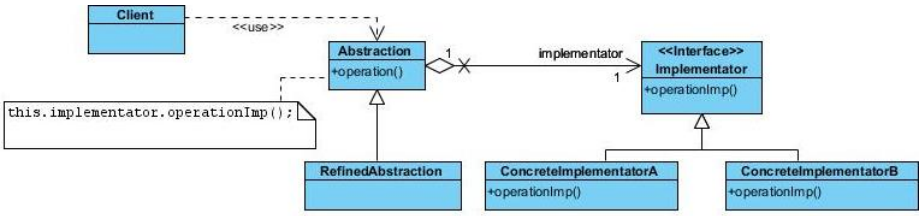

Figura B.1: *Estrutura do Bridge (adaptada de [Gamma et al., 1995])*

Bridge

O padrão estrutural Bridge serve para separar a interface (nível de abstracção) da implementação, para que ambas possam evoluir de forma independente. Deve ser utilizado quando, por exemplo, a implementação só é escolhida em tempo de execução. Como "preço a pagar", tal como referido por [Pohl and Gerlach, 2003], temos a existência de uma indirecção adicional. O Bridge permite também que a evolução da interface e da implementação possa ser feita usando herança [Gamma et al., 1995].

Pela análise da Figura B.1 vemos que o cliente comunica apenas com a classe Abstraction (não é necessariamente um interface), que é o topo da hierarquia ao nível da abstracção. Esta classe apresenta métodos que as suas subclasses herdam (representadas na Figura B.1 pela subclasse RefinedAbstraction). As subclasses podem acrescentar comportamento à classe Abstraction. A classe Abstraction tem ainda uma referência para o interface Implementator, que é o que vai poder utilizar quando for necessário criar um objecto do tipo ConcreteImplementatorA ou ConcreteImplementatorB, como se verifica na Figura B.1 (a já referida indirecção).

Como um exemplo de utilização do padrão Bridge, podemos considerar a seguinte situação: ao nível da abstracção temos uma superclasse Stack, com todas as funcionalidades associadas, e uma referência para o modo como a *stack* deve ser implementada; nessa classe temos duas subclasses (Hanoi e Fifo), que representam dois tipos de *stacks* com diferentes implementações do método push(i : int) e pop() : int, respectivamente, definidos pela superclasse, e temos ainda um novo método na subclasse Hanoi, reportRejected() : int. Quanto à implementação, temos um interface StackImpl, concretizado pelas subclasses SArray e SList, que definem dois modos diferentes para a implementação de uma *stack* de inteiros (usando um *array*, e usando uma lista). Este exemplo pode ser visto na Figura B.2.

Com a utilização do Bridge, vemos que é também relativamente simples acrescentar novas abstracções, bastando para isso, neste caso, adicionar novas subclasses de Stack. De forma semelhante, basta também criar novas implementações do interface StackImpl para termos novas formas de dar comportamento aos métodos da Stack.

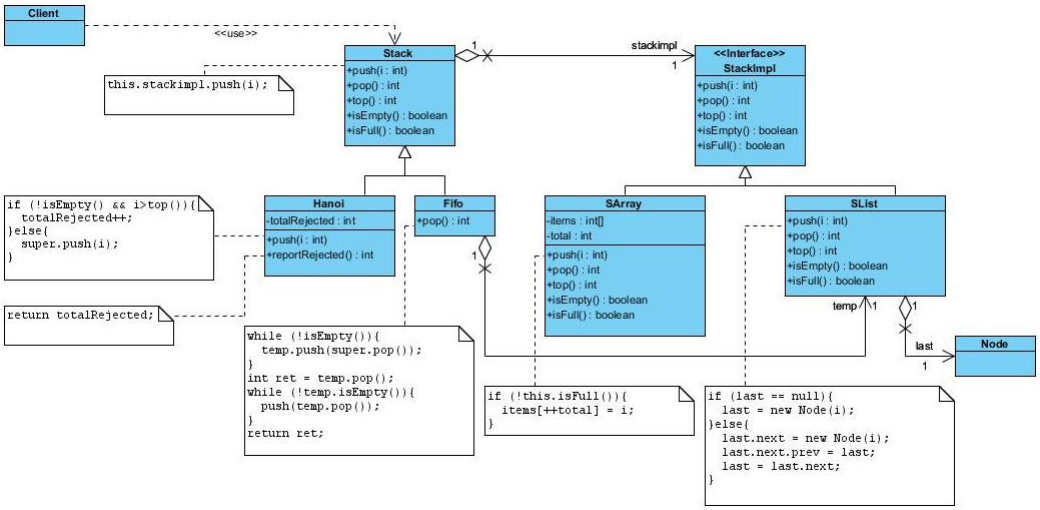

Figura B.2: *Exemplo de Utilização do Bridge (adaptada de [Gamma et al., 1995])*

Decorator

O padrão Decorator permite adicionar responsabilidades a um objecto dinamicamente. Oferece uma alternativa ao mecanismo de herança para subclasses. Assim, devemos utilizar este padrão quando pretendemos adicionar comportamento ou estado a objectos, em tempo de execução [Gamma et al., 1995].

Como se pode verificar pela Figura B.3, existe a classe Component (pode ou não ser um interface) que representa o supertipo dos objectos que pretendem ter funcionalidades acrescentadas. A classe ConcreteComponent implementa esses objectos [Kniesel et al., 2004].

A classe Decorator é uma classe abstracta que herda/implementa os métodos de Component, e contém uma variável do supertipo Component, na qual vai invocar o método operation(). Essa classe tem duas subclasses, ConcreteDecoratorA e ConcreteDecoratorB, que implementam cada uma à sua maneira o método operation() de Decorator, e pretendem acrescentar funcionalidades à classe Component [Kniesel et al., 2004].

O cliente comunica com Component, criando instâncias suas de subtipos ConcreteComponent, ou ConcreteDecoratorA ou ConcreteDecoratorB, e vai compondo à sua medida a instância desejada [Gamma et al., 1995].

Como exemplo de utilização do Decorator, temos o interface Widget com o método draw(), a ser implementado pelas subclasses TextField e Decorator. A classe Decorator tem uma variável do tipo Widget, e é sobre essa variável que invoca o seu método draw(). Em seguida existem duas subclasses que herdam de Decorator, e redefinem o método draw() à sua medida. Este exemplo encontra-se representado na Figura B.4.

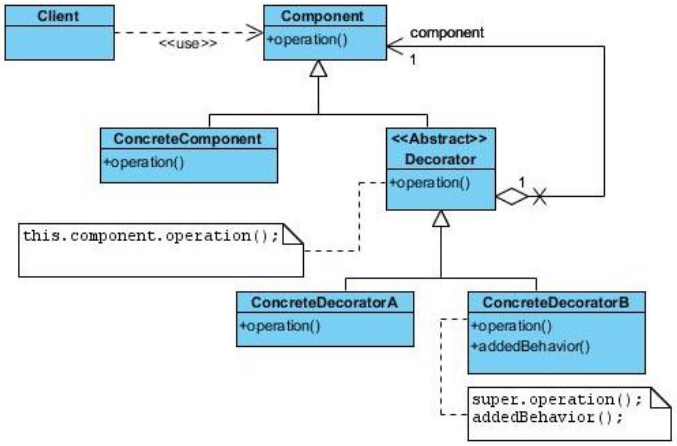

Figura B.3: *Estrutura do Decorator (adaptada de [Gamma et al., 1995])*

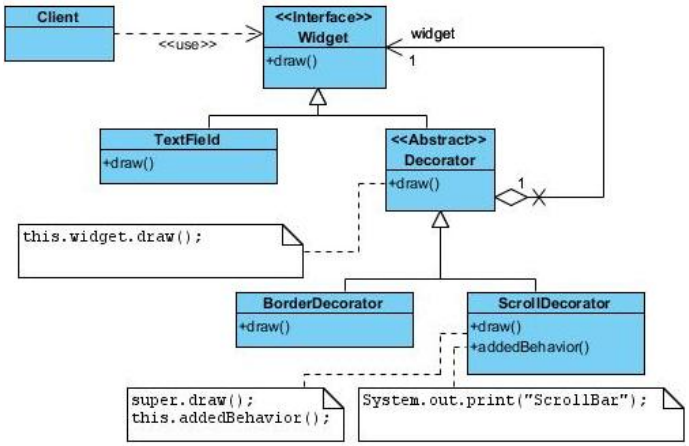

Figura B.4: *Exemplo de Utilização do Decorator (adaptada de [Gamma et al., 1995])*

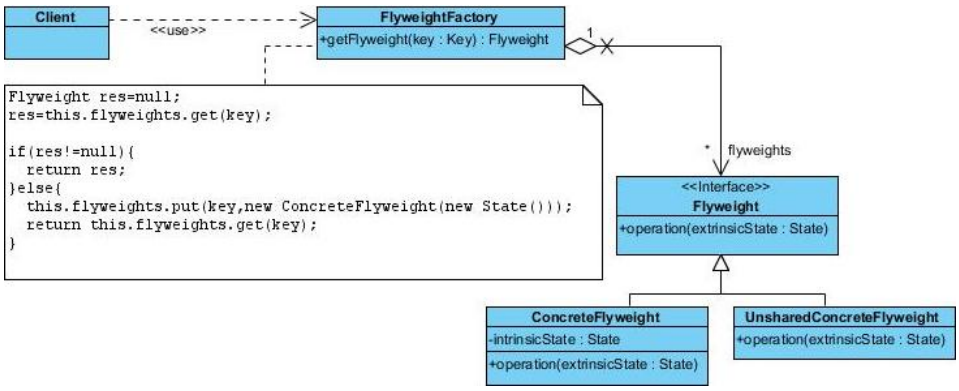

Figura B.5: *Estrutura do Flyweight (adaptada de [Gamma et al., 1995])*

Flyweight

O Flyweight é um padrão estrutural que deve ser utilizado quando é necessário ter muitos objectos a existir simultaneamente, e que partilham alguma informação. Assim, se a informação partilhada por todos esses objectos é intrínseca (independente do estado do objecto), imutável e idêntica para todos eles, então pode ser retirada de cada objecto, passando a ser apenas referenciada [Gamma et al., 1995].

Desta forma, tal como mencionado em [van Gurp and Bosch, 2002], é possível reduzir a quantidade de memória gasta por esses objectos, pois em vez de estarmos a guardar informação repetida e invariante para cada objecto existente, apenas a guardamos uma única vez (num objecto flyweight), ou seja, apenas teremos o custo de criar uma instância desses objectos, que será reutilizada em todo o programa.

O objecto flyweight pode também ter a si associada informação extrínseca (dependente do estado do objecto), que deve ser *stateless* e determinada pelo contexto. Essa informação é armazenada ou calculada pelo objecto cliente, sendo depois passada ao objecto flyweight quando as suas operações são invocadas[1].

Vemos então na Figura B.5 a estrutura deste padrão, onde existe um interface Flyweight com o método a invocar nas subclasses ConcreteFlyweight e UnsharedConcreteFlyweight. A classe ConcreteFlyweight contém a parte independente do estado, definida pela variável intrinsicState, que vai poder ser então partilhada; já UnsharedConcreteFlyweight contém a parte dependente do estado, sendo agora importante para as instâncias desta subclasse receber o estado actual através do parâmetro do método operation(extrinsicState : State).

O cliente, quando pretender um objecto do tipo Flyweight, comunica com a classe FlyweightFactory (contém um conjunto de instâncias de Flyweight

[1] http://sourcemaking.com/design_patterns

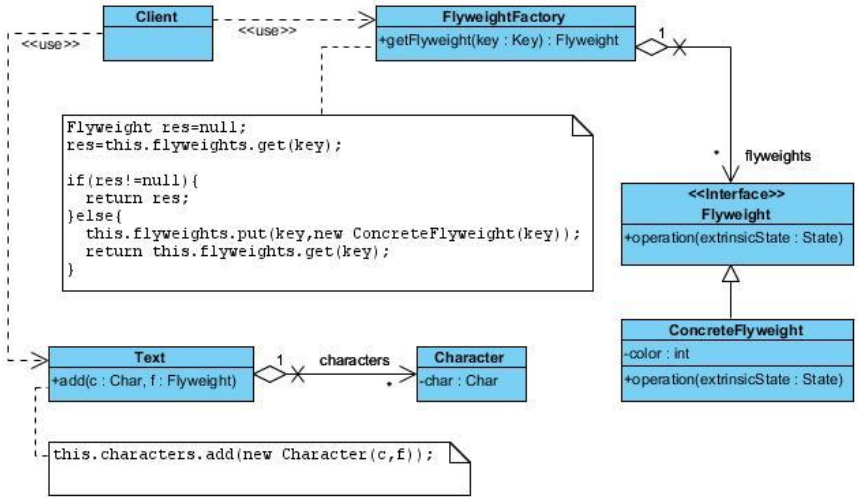

Figura B.6: *Exemplo de Utilização do Flyweight (adaptada de [Gamma et al., 1995])*

que podem ser partilhadas) e pede-lhe um através de uma key, e essa classe devolve-lho (caso esteja no conjunto) ou então cria um novo. Quanto ao estado extrínseco, será mantido pelo cliente e passado no método dos objectos flyweight.

Como exemplo de utilização deste padrão, podemos considerar o descrito na Figura B.6, onde se quer manter o atributo color do objecto Character igual para todos os objectos Character presentes na aplicação.

Assim, o cliente utiliza um objecto Text, que contém um conjunto de objectos do tipo Character. Além disso, o cliente pede à FlyweightFactory um objecto Flyweight, onde vai guardar o atributo color dos vários objectos Character, igual e partilhada por todos eles.

Private Class Data

O padrão Private Class Data permite controlar o acesso para escritas nos atributos de uma classe, fazendo uma separação dos dados relativamente aos métodos que os utilizam. Encapsula a inicialização da classe dos dados[2].

Assim, em vez de termos uma classe principal com variáveis de instância e métodos para interagir com essas variáveis, passamos a ter uma referência na classe principal para uma outra classe, a classe dos dados, e aí sim estão as variáveis de instância e os métodos *get* e *set* respectivos[3]. Esta situação entretanto descrita pode ser vista na Figura B.7.

Vemos que agora passa a haver uma separação entre as variáveis de instância de MainClass, passando a estar encapsuladas por uma outra classe, a

[2] http://sourcemaking.com/design_patterns
[3] http://sourcemaking.com/design_patterns

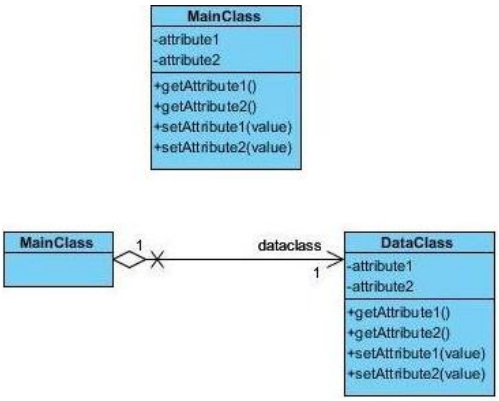

Figura B.7: *Estrutura do Private Class Data*

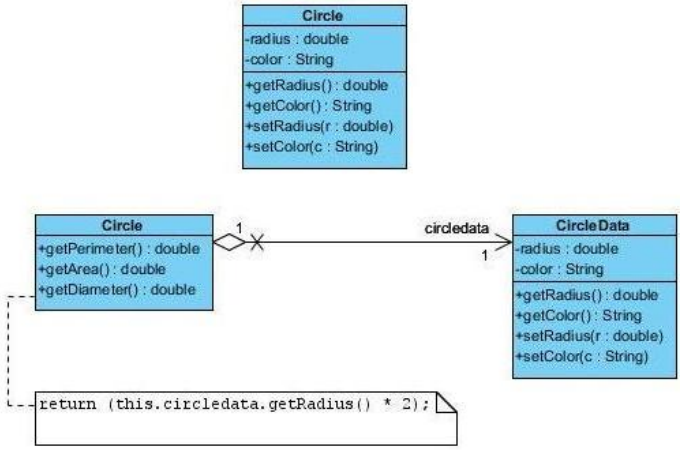

Figura B.8: *Exemplo de Utilização do Private Class Data*

classe DataClass, que fornece os respectivos métodos de acesso a essas variáveis.

Como exemplo de utilização deste padrão, temos uma classe Circle que contém uma referência para a classe CircleData, onde estão guardadas as propriedades dos objectos do tipo Circle, e ainda métodos de acesso a esses atributos[4]. Este exemplo pode ser visto na Figura B.8.

Proxy

O padrão Proxy tem por objectivo oferecer um objecto que sirva de contentor para um outro objecto. Devemos usar esta alternativa quando se precisa de referenciar um objecto de uma forma mais versátil do que um apontador, ou ainda quando pretendemos ter alguma protecção para o objecto a refe-

[4] http://sourcemaking.com/design_patterns

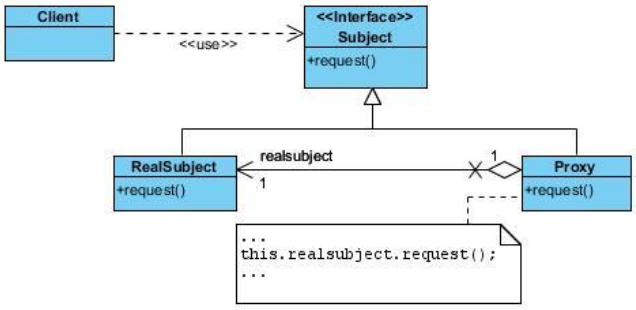

Figura B.9: *Estrutura do Proxy (adaptada de [Gamma et al., 1995])*

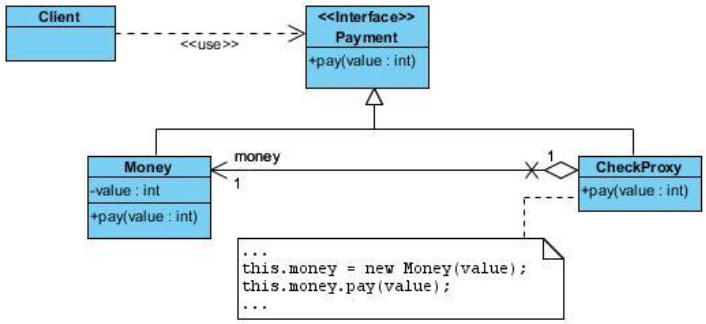

Figura B.10: *Exemplo de Utilização do Proxy (adaptada de [Gamma et al., 1995])*

renciar, e podemos assim criar objectos "pesados" apenas quando for mesmo necessário, assumindo momentaneamente o objecto proxy a identidade desse outro objecto [Gamma et al., 1995].

Como se observa na Figura B.9, existe o interface Subject com métodos a implementar pelas subclasses, RealSubject e Proxy. A classe RealSubject representa o objecto real, e a classe Proxy representa o seu contentor, ou seja, um outro objecto que tem uma referência para o objecto real, e que se faz passar por ele.

Assim, o cliente tem instâncias de Subject, concretizadas por objectos Proxy, e que apenas vão utilizar o objecto real quando necessário, ou seja, quando nos seus métodos existir a indicação para instanciar um objecto real; até lá, vão "fazendo de conta" que são esse objecto real.

Como se verifica através da observação da Figura B.10, temos um interface Payment, e como subclasses temos Money e CheckProxy. O cliente para efectuar o seu pagamento utiliza uma instância de CheckProxy, cujo método pay(value : int) só vai criar o objecto Money algum tempo depois. Até lá, CheckProxy faz o papel de Money.

Anexo C

Padrões Comportamentais

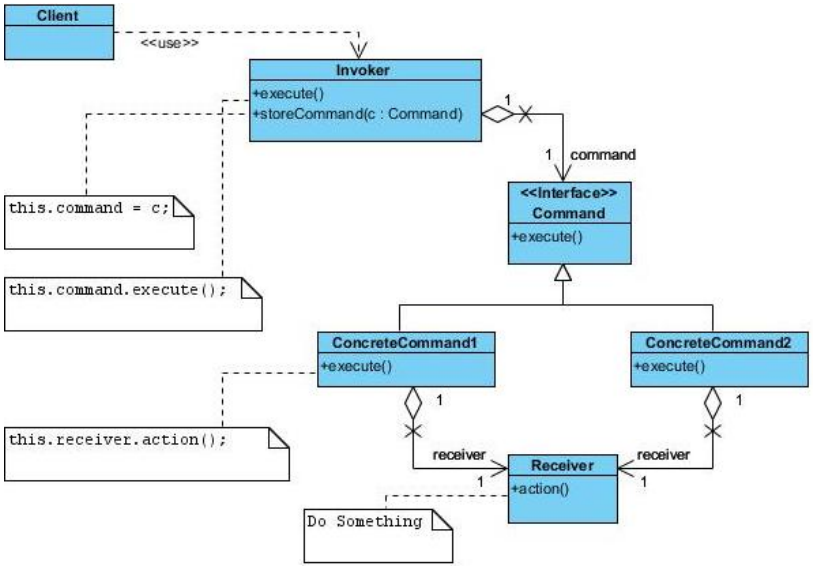

Figura C.1: *Estrutura do Command (adaptada de [Gamma et al., 1995])*

Command

O padrão comportamental Command permite que se encapsule um pedido como sendo um objecto, de modo a tornar possível que os clientes possam ser parametrizados por diferentes pedidos [Gamma et al., 1995].

Devemos recorrer a este padrão se quisermos parametrizar os objectos pela acção a executar, e também se necessitarmos de especificar, serializar e efectuar pedidos em alturas diferentes da execução. Além destas razões, se for necessário ter um mecanismo de *undo* na aplicação, o Command é capaz de o permitir, na medida em que como o comando é encapsulado num objecto, pode não ser executado, dando oportunidade de se recuperar o estado anterior. Por fim, o Command oferece uma noção de transacção, pois um comando pode ser visto, de uma forma simples, como uma transacção [Gamma et al., 1995].

Através da observação da Figura C.1 podemos ver que o interface Command contém o método execute() a ser implementado pelas subclasses, ConcreteCommand1 e ConcreteCommand2. Cada uma delas receberá no construtor uma instância de Receiver, que vai servir para nela se invocar o método action(). A classe Invoker terá uma referência para um Command, no qual vai invocar o método execute(). O Cliente apenas terá de criar um Invoker com o Receiver e o Command que deseja (ConcreteCommand1 ou ConcreteCommand2, neste caso) e chamar o método execute().

Quanto a um exemplo de utilização do padrão Command, pode-se observar a situação representada na Figura C.2.

É um exemplo bastante semelhante ao já descrito, existindo dois métodos

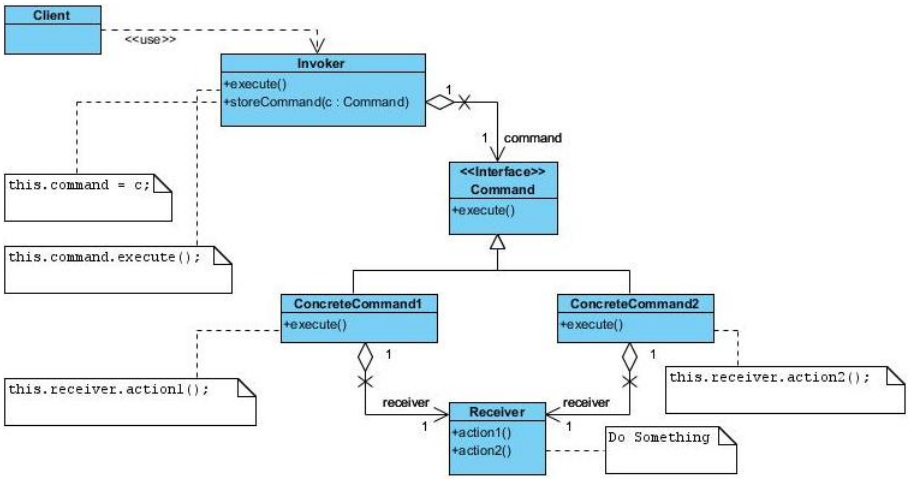

Figura C.2: *Exemplo de Utilização do Command (adaptada de [Gamma et al., 1995])*

na classe Receiver que devolvem resultados diferentes. A chamada a cada um dos métodos fica encapsulada na instância de ConcreteComand (1 ou 2) criada, sendo que aí no caso de se ter instanciado ConcreteCommand1 ao invocar execute() vai ser executado o método action1() de Reveiver; caso a instância seja de ConcreteCommand2 vai ser invocado action2() em Receiver. Isto apenas depende da instância criada pelo Cliente.

Interpreter

O padrão Interpreter tem por objectivo, a partir de uma dada linguagem, definir uma representação para a sua gramática juntamente com um interpretador que utilize essa representação para interpretar frases nessa linguagem. Faz o mapeamento de um domínio para uma linguagem, da linguagem para a gramática, e da gramática para uma estrutura orientada aos objectos [Gamma et al., 1995].

O Interpreter utiliza uma classe para representar cada regra da gramática da linguagem em questão, e tendo muitas vezes as gramáticas uma estrutura hierárquica, a utilização de herança de classes funciona bem. A estrutura deste padrão pode ser vista na Figura C.3.

Como se verifica nesta estrutura, o cliente vai criando expressões do tipo AbstractExpression, que podem ser concretizadas em TerminalExpression ou CompoundExpression, em que estas últimas contêm uma variável do tipo AbstractExpression. O cliente apenas passa uma instância de Context quando invocar solve(c : Context), e a expressão vai sendo resolvida, modificando-se o contexto na invocação sucessiva de solve(c : Context).

Como exemplo de utilização deste padrão, podemos considerar um sistema que simule uma calculadora básica (apenas operação de somar, subtrair

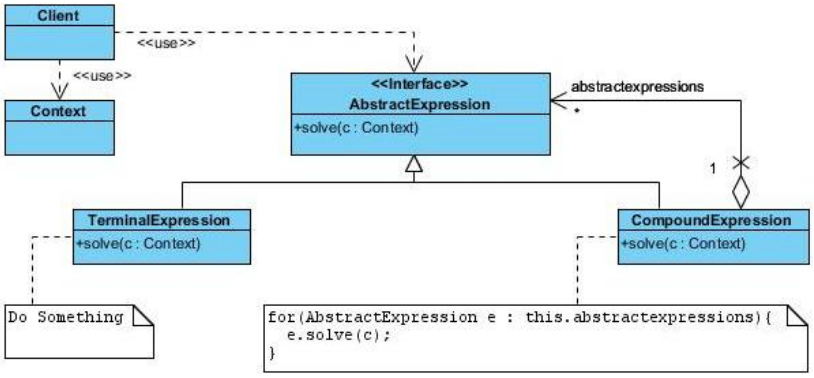

Figura C.3: *Estrutura do Interpreter (adaptada de [Gamma et al., 1995])*

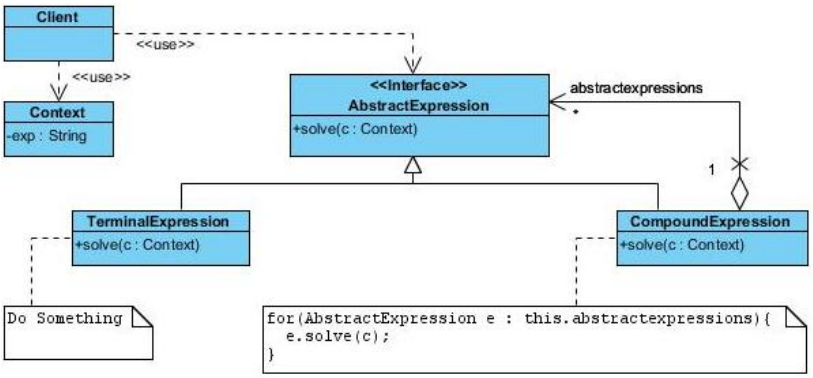

Figura C.4: *Exemplo de Utilização do Interpreter (adaptada de [Gamma et al., 1995])*

e multiplicar, sem símbolos adicionais), onde se passa como parâmetro uma expressão, que é o contexto inicial, que vai sendo modificada à medida que vai sendo resolvida. As expressões terminais serão os dígitos de 0 a 9. Este exemplo pode ser visto na Figura C.4.

Mediator

O padrão Mediator serve para definir um objecto que vai encapsular a forma como um conjunto de objectos vai interagir. Permite que os objectos não se referenciem uns aos outros, mas utilizem um outro objecto como mediador, permitindo que os objectos possam na mesma interagir entre eles [Gamma et al., 1995].

Devemos utilizar este padrão quando tivermos, por exemplo, dois conjuntos de objectos, e onde cada objecto de cada conjunto pode referenciar um ou vários objectos do outro conjunto. Assim, em vez de termos muitas dessas relações, temos apenas a possibilidade de cada objecto comunicar com o mediador, simplificando o processo.

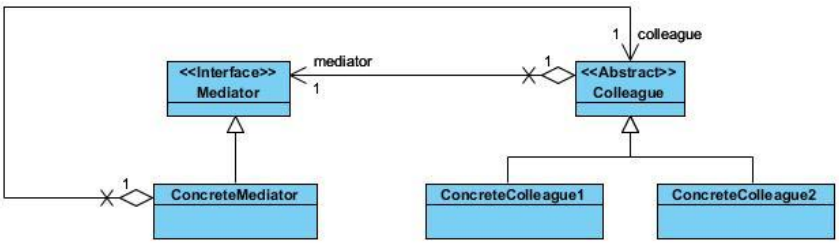

Figura C.5: *Estrutura do Mediator (adaptada de [Gamma et al., 1995])*

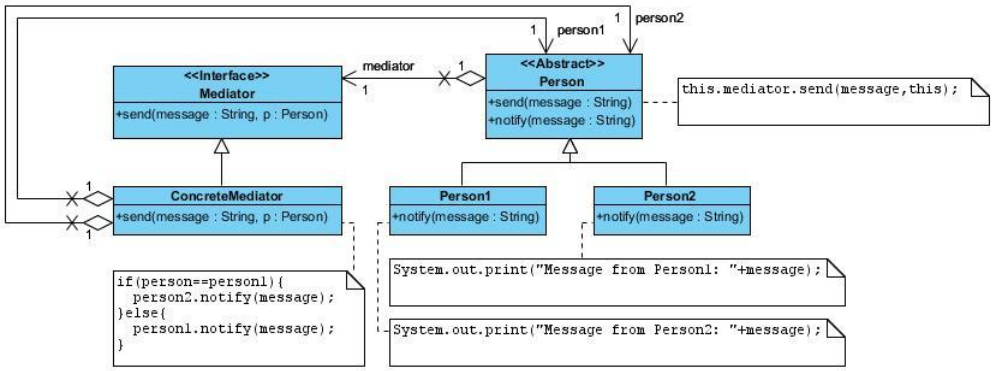

Figura C.6: *Exemplo de Utilização do Mediator (adaptada de [Gamma et al., 1995])*

Através da Figura C.5 podemos ver que existe o interface Mediator, que define os métodos necessários para a comunicação com objectos do tipo Colleague. Este interface é implementado por ConcreteMediator, que contém uma referência para o ConcreteColleague apropriado. A classe Colleague é uma classe abstracta e contém uma referência para o Mediator a utilizar, passado no seu construtor, além de métodos a utilizar na comunicação com o Mediator.

Como exemplo de utilização do Mediator, podemos ter duas instâncias de ConcreteColleague (Person1 e Person2) que pretendem enviar mensagens entre elas. Têm os métodos send(message : String) e notify(message : String), para enviar uma mensagem, e para receber mensagens. Usam um objecto ConcreteMediator que serve de intermediário, e que implementa o método send(message : String, p : Person), verificando quem envia a mensagem e para quem, e vai chamar o método notify(message : String) do receptor desejado para a mensagem. Este exemplo pode ser visto na Figura C.6.

Memento

O padrão de comportamento Memento sugere que, sem que se viole o encapsulamento, se possa capturar o estado interno de um objecto, de modo a

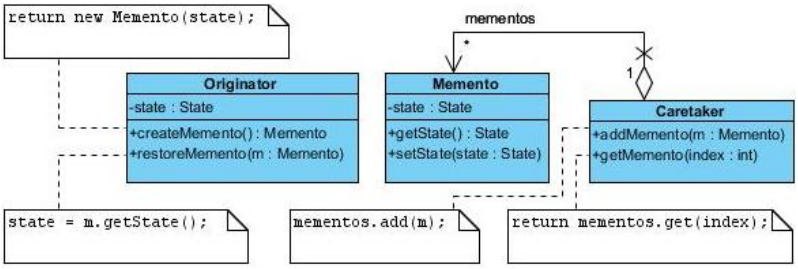

Figura C.7: *Estrutura do Memento (adaptada de [Gamma et al., 1995])*

que esse objecto possa voltar a esse mesmo estado anterior [Gamma et al., 1993]. Consegue-se assim fazer um *checkpoint* do estado do objecto, quando quisermos, e podemos desta forma ter um mecanismo de *undo* associado a esse objecto.

Através da observação da Figura C.7 verificamos que o objecto que vai ter o seu estado guardado é chamado de Originator. Possui um estado (representado por state) que podem ser os valores das suas variáveis de instância, por exemplo, e contém ainda métodos responsáveis por criar um novo Memento, fornecendo-lhe o seu estado actual, e restaurar o estado anterior, a partir de um Memento recebido por parâmetro.

O objecto Memento vai guardar o estado recebido aquando da sua criação, e tem métodos para modificar esse estado por um novo, ou então devolver esse estado. O objecto Caretaker possui uma variável de instância que é um Memento, mas não a pode modificar; apenas vai ser utilizada pela aplicação quando desejar restaurar o estado do objecto Originator. Importa referir que Caretaker poderá conter um conjunto de objectos Memento, correspondendo aos vários estados de Originator, podendo-se assim regressar a qualquer um deles.

Quanto a um exemplo de utilização do Memento, podemos considerar a necessidade de guardar os endereços das páginas *web* visualizadas por um *browser*, sendo que se quer ter implementado o mecanismo de retroceder. Assim, podemos ver cada página acedida como um estado, estado esse que queremos guardar num Memento para que depois se possa lá voltar. Na Figura C.8 podemos ver este exemplo, onde a classe WebPage representa o objecto cujo estado (endereço) importa colocar num Memento, e a classe History mostra os endereços acedidos (conjunto de objectos Memento), e permite à aplicação escolher um deles e recolocá-lo na WebPage.

Null Object

O padrão Null Object tem por missão encapsular a ausência de um determinado objecto, pois vai oferecer uma alternativa que terá um comportamento que é "não fazer nada". Isto permite que, em situações onde é necessário lidar com referências de objectos que podem ser *null*, obrigando a testar esses

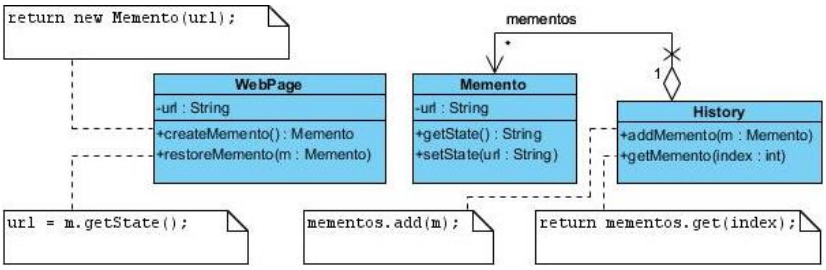

Figura C.8: *Exemplo de Utilização do Memento (adaptada de [Gamma et al., 1995])*

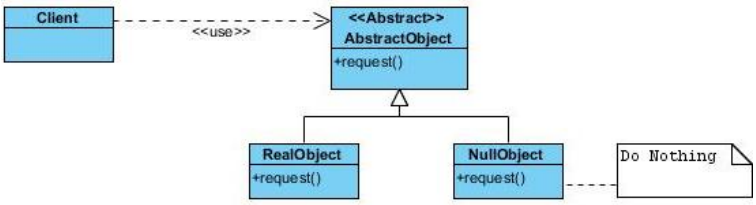

Figura C.9: *Estrutura do Null Object*

casos, e onde em caso de ser mesmo *null* não se vai implementar comportamento algum, importa conseguir fazer o tratamento do objecto que é *null* de uma forma transparente, tratando-o de igual forma como se não fosse *null*[1].

Através da Figura C.9 verifica-se que existe a definição de uma classe AbstractObject, que é uma classe abstracta que funciona como supertipo do objecto que queremos definir. Contém um método request() que vai ser herdado pelas subclasses. Esse método define algum comportamento associado ao objecto a instanciar. A subclasse RealObject representa os objectos AbstractObject que vão implementar um comportamento, ao passo que a classe NullObject corresponde ao objecto AbstractObject que não vai ter comportamento definido. A classe Client cria instâncias de AbstractObject para utilizar como necessitar[2].

Para ilustrar melhor a utilidade do Null Object podemos considerar o exemplo presente na Figura C.10.

Neste exemplo pretende-se implementar uma lista ligada, que vai ser o objecto abstracto. Essa lista terá um objecto à cabeça, e uma cauda, que é a lista restante. Como subclasses temos uma lista não vazia (contém a cabeça e uma cauda) com comportamento associado, como devolver o elemento à cabeça, ou devolver a lista restante (cauda); e temos também uma lista vazia (Null Object) para representar o fim da lista, sem comportamento.

Assim, quando se estiver a iterar sobre a lista ligada, não necessitamos de estar sempre a testar se já se chegou ao fim da lista, pois quando lá se chegar

[1] http://sourcemaking.com/design_patterns
[2] http://sourcemaking.com/design_patterns

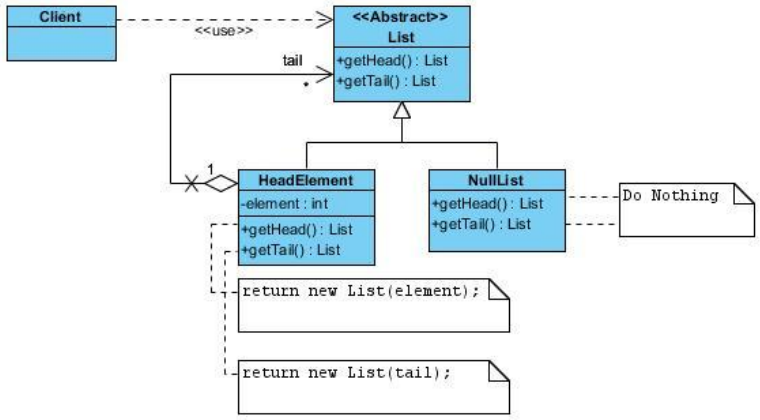

Figura C.10: *Exemplo de Utilização do Null Object*

estaremos a operar sobre uma lista vazia, cujos métodos já estão definidos e não vão provocar nenhuma excepção na aplicação[3].

Observer

O padrão Observer permite que se definam dependências do tipo um para muitos, fazendo com que quando um objecto altere o seu estado, todos os outros objectos que dele dependem sejam avisados e actualizem esse estado automaticamente [Gamma et al., 1995].

Devemos utilizar o Observer quando uma mudança num objecto implica mudar o estado de outros (possivelmente muitos), mas não se sabe à partida quantos. Também devemos recorrer a este padrão de comportamento quando queremos que um determinado objecto avise outros, sem a necessidade de saber que objectos são nem como se comportam [Gamma et al., 1995].

Através da observação da Figura C.11, podemos verificar que a classe Subject funciona como uma superclasse, e a classe ConcreteSubject herda os métodos nela definidos. A classe Subject contém um conjunto de objectos Observer, que são aqueles que estão interessados em ser informados quando houver uma mudança de estado no objecto ConcreteSubject. A classe Subject permite que sejam adionados/removidos novos objectos Observer ao seu conjunto, e avisa-os quando há mudança de estado. A classe Observer é um interface, com um método implementado pelas suas subclasses, que neste caso é a subclasse ConcreteObserver. Esta contém uma referência para o ConcreteSubject que lhe interessa.

Ainda sobre a estrutura do Observer, tal como mencionado em [Mak et al., 2004], não temos necessariamente que ter definidas as classes Observer e ConcreteObserver; no caso de apenas uma classe de objectos actuar como observer, não havendo vários subtipos de observers, podemos ter apenas a classe

[3] http://sourcemaking.com/design_patterns

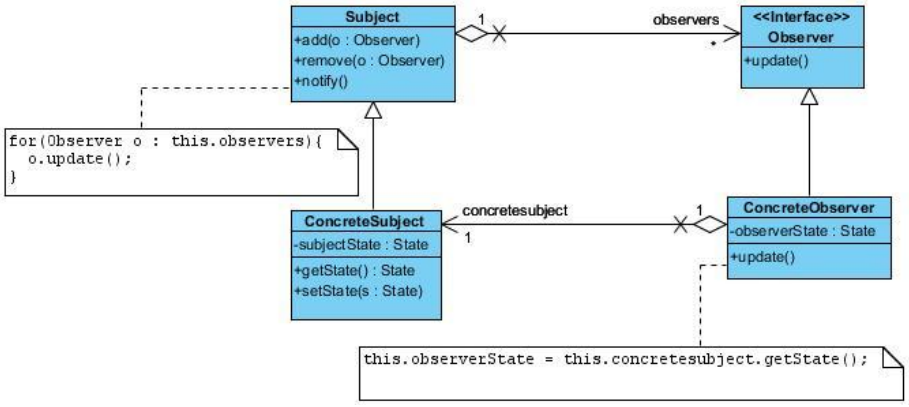

Figura C.11: *Estrutura do Observer (adaptada de [Gamma et al., 1995])*

Observer, deixando ser um interface, e passando a ter um estado e métodos lá definidos.

No entanto, a opção de ter subclasses e um interface será sempre a mais indicada, até porque facilita que, de futuro, se possam adicionar novos observers, bastando que implementem o interface Observer [Piveta and Zancanella, 2003].

Como exemplo de utilização deste padrão podemos considerar um leilão, onde cada pessoa pode fazer uma proposta (se assim o entender), e caso seja maior que a última efectuada, o moderador do leilão avisa todos os participantes que o valor pelo artefacto a leiloar se modificou.

Assim, como se verifica através da observação da Figura C.12, temos a classe Subject com os métodos já descritos, e da mesma forma temos a classe Observer. A classe Auction será subclasse de Subject, e a classe Person é subclasse de Observer.

Como se pode ver na Figura C.12, quando é feita uma nova oferta por parte de uma pessoa, caso o valor sugerido seja superior ao actual valor, então todos os interessados no leilão devem ser notificados. Para isso, é invocado o método notify() presente na superclasse Subject, que por sua vez chama o método update() de cada observer [Lauder and Kent, 1998].

State

O padrão comportamental State deixa que um objecto possa alterar o seu comportamento quando o seu estado interno mudar. Desta forma parecerá que o objecto mudou a sua classe. Consegue-se assim ter uma representação de uma máquina de estados orientada aos objectos [Gamma et al., 1995].

Devemos usar este padrão quando o comportamento associado a um determinado objecto depende do seu estado, e não é viável termos esses comportamentos definidos com o recurso a estruturas condicionais. Isto acontece porque cada uma das hipotéticas partes condicionais vai ter de redefinir

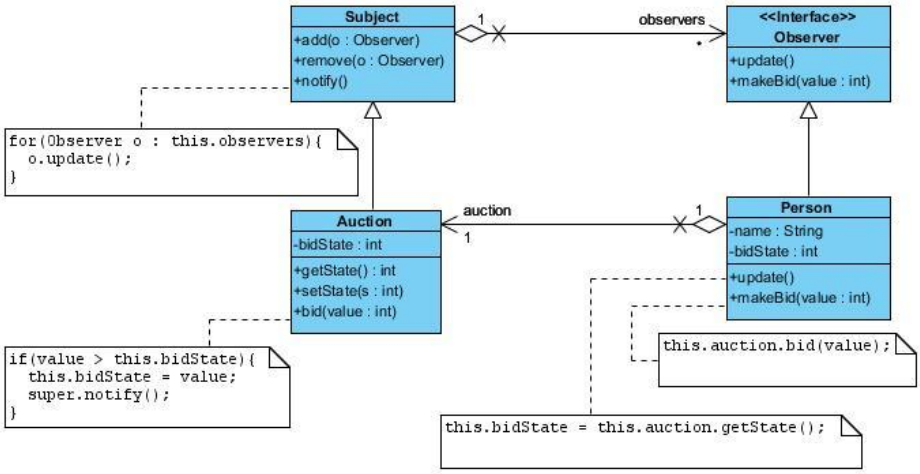

Figura C.12: *Exemplo de Utilização do Observer (adaptada de [Gamma et al., 1995])*

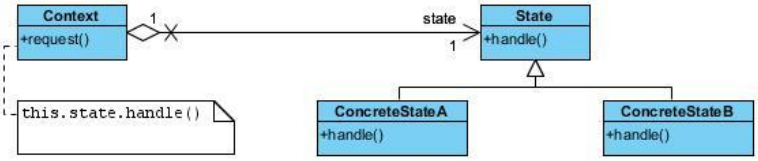

Figura C.13: *Estrutura do State (adaptada de [Gamma et al., 1995])*

muitas coisas, tornando impraticável a sua programação usando estruturas condicionais [Gamma et al., 1995].

Assim sendo, tal como é referido em [van Gurp and Bosch, 2002] com a utilização do padrão de comportamento State, os estados são implementados como subclasses de uma classe State, como se verifica pela observação da Figura C.13.

Através da sua análise verifica-se que estão representados dois estados possíveis (subclasses de State), herdando o método handle() responsável por definir o comportamento a implementar em cada estado.

A classe Context contém uma instância de State, que representa o estado actual onde o objecto se encontra, e os métodos que o cliente pode invocar para gerar um evento no objecto, e provocar uma mudança no seu estado, caso seja necessário.

Como exemplo de utilização deste padrão podemos considerar uma simulação do comportamento associado a uma ventoinha. Este exemplo pode ser encontrado na Figura C.14.

Temos então a classe Fan, que representa a ventoinha, e que pode ter os seguintes estados: Off, Normal e Fast. A classe Context guarda o estado actual da ventoinha, e disponibiliza os métodos que o cliente pode invocar

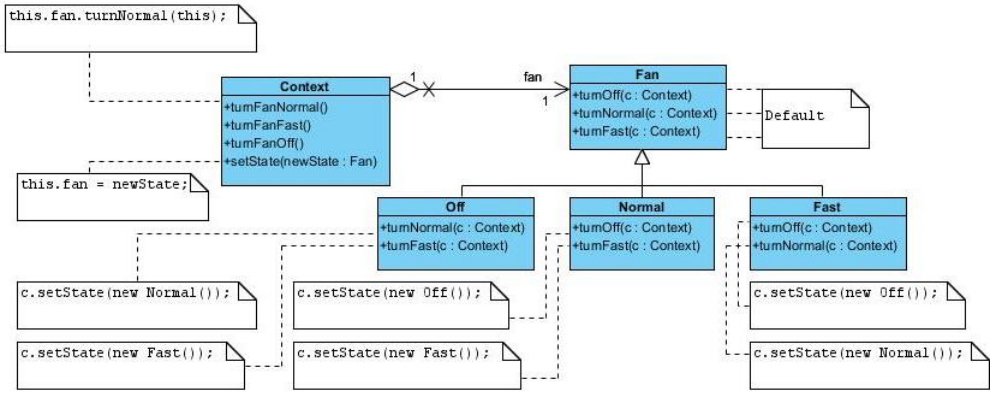

Figura C.14: *Exemplo de Utilização do State (adaptada de [Gamma et al., 1995])*

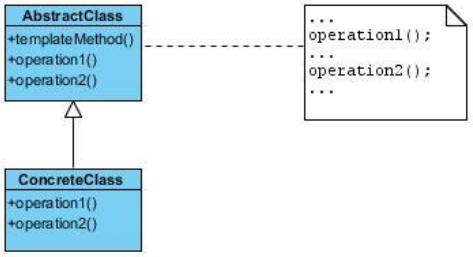

Figura C.15: *Estrutura do Template Method (adaptada de [Gamma et al., 1995])*

para provocar as mudanças de estado.

Template Method

O padrão Template Method define o esqueleto de um algoritmo para uma dada operação, mas deixa que as subclasses implementem alguns passos do algoritmo, sem que alterem a estrutura do mesmo. Caberá a quem estiver a desenvolver um algoritmo identificar quais os seus passos que são invariantes, e quais os que podem ser variantes [Gamma et al., 1995].

Assim sendo, os passos invariantes serão definidos numa classe abstracta, deixando para as subclasses a definição dos passos variantes. Esta estrutura pode ser vista na Figura C.15.

Assim, vemos que a classe AbstractClass tem a definição do método templateMethod() que apenas chama, pela ordem desejada, os métodos operation1() e operation2().

No caso de estes métodos serem invariantes, então estão definidos na própria classe; no caso de serem variantes, serão definidos nas várias subclasses ConcreteClass que for necessário criar (uma para cada variação em cada passo do algoritmo). O cliente apenas vai interagir com a classe AbstractClass, usando o método templateMethod().

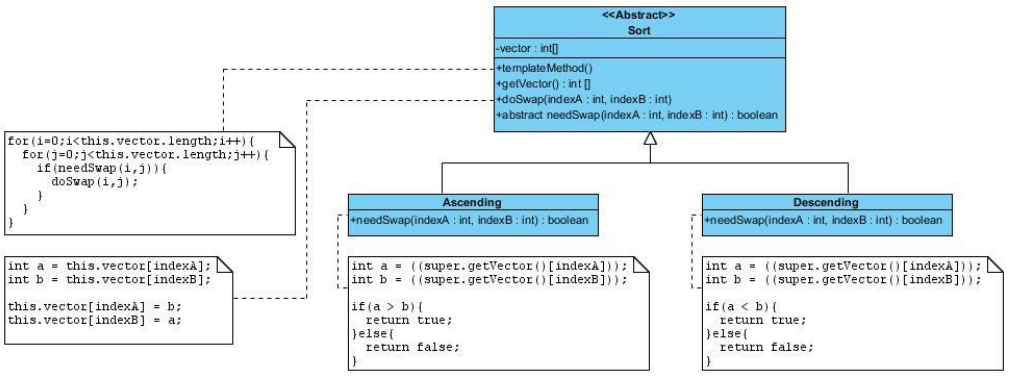

Figura C.16: *Exemplo de Utilização do Template Method (adaptada de [Gamma et al., 1995])*

Como um exemplo de utilização deste padrão, podemos considerar que queremos ordenar os elementos (números inteiros) de um *array*, mas que essa ordenação pode variar, sendo crescente ou decrescente.

Assim, a comparação entre os elementos (dois a dois) varia de acordo com o algoritmo, pois só queremos que troquem as suas posições no *array* caso se verifique uma determinada condição (por exemplo, se quisermos ordenar por ordem crescente, e formos percorrendo o *array* do início para o fim, trocaremos de posição um par de elementos caso o primeiro elemento seja maior que o segundo). Já o método para efectivar a troca de posição de dois elementos será sempre igual e invariante[4]. Este exemplo encontra-se representado na Figura C.16.

Visitor

O padrão comportamental Visitor possibilita que se represente uma operação a efectuar nos elementos de uma estrutura de objectos, podendo assim ser definidas novas operações sem que seja necessário mudar a classe dos elementos onde opera [Gamma et al., 1995].

Devemos então utilizar o Visitor quando queremos efectuar operações distintas e não relacionadas nos elementos de uma estrutura de objectos, sem que tenhamos de implementar essas mesmas operações nos objectos desse conjunto. Para melhor se compreender este padrão podemos ver a sua estrutura na Figura C.17.

Através da análise da Figura C.17, podemos encontrar a classe ObjectStructure que contém um conjunto de objectos do tipo Element. A classe Element é um interface que define um método que recebe um Visitor como parâmetro.

Como subclasses de Element temos dois elementos em concreto, e ambos

[4] http://sourcemaking.com/design_patterns

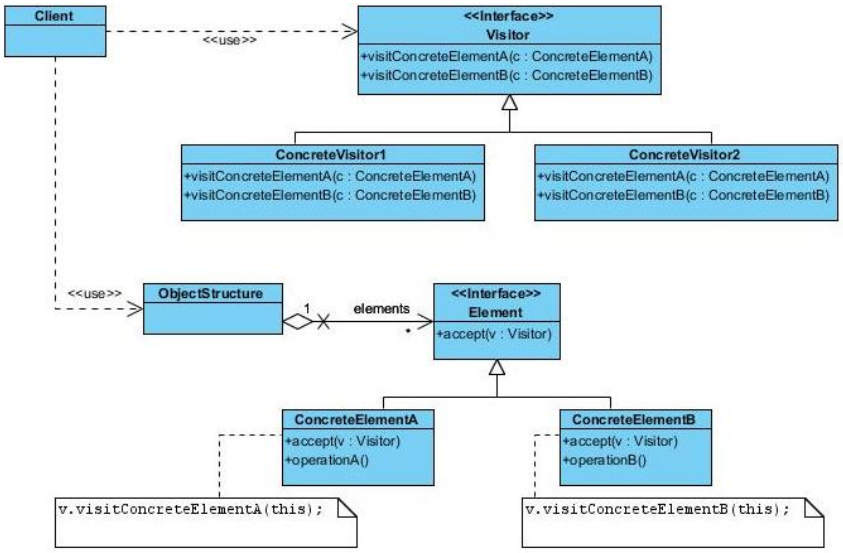

Figura C.17: *Estrutura do Visitor (adaptada de [Gamma et al., 1995])*

implementam o comportamento associado ao método do interface. Nesse método accept(v : Visitor), tal como referido em [Palsberg and Jay, 1998], cada uma das subclasses vai passar o controlo para o Visitor recebido como parâmetro, bastando para isso invocar o método adequado no Visitor recebido.

A classe Visitor é também um interface que define uma operação para cada subclasse de Element existente. Essas operações serão implementadas nas subclasses ConcreteVisitor1 e ConcreteVisitor2. A classe ObjectStructure poderá também ter um método que receba um Visitor e trate de, para todos os elementos do conjunto, invocar o método accept(v : Visitor) com o Visitor recebido. Quanto ao cliente, apenas tem de utilizar o Visitor desejado na estrutura de dados que possui.

Como exemplo de utilização do Visitor podemos considerar a situação descrita na Figura C.18.

Temos então uma classe Set para representar um conjunto de elementos do tipo Element. Esses elementos são do tipo VInt ou VDec. Os elementos do tipo VDec possuem uma operação particular de arredondamento para o inteiro mais próximo, e os VInt de arredondar para a dezena mais próxima. O Visitor vai ter de, para cada elemento do conjunto, fazer a operação correspondente e imprimir o novo resultado.

Relativamente a limitações na utilização do Visitor, em [Büttner et al., 2004] são referidos aspectos importantes, entre os quais importa salientar que pode ser bastante complexo introduzir este padrão numa aplicação já desenvolvida, pois obriga a que as classes envolvidas tenham de definir o método accept(v : Visitor).

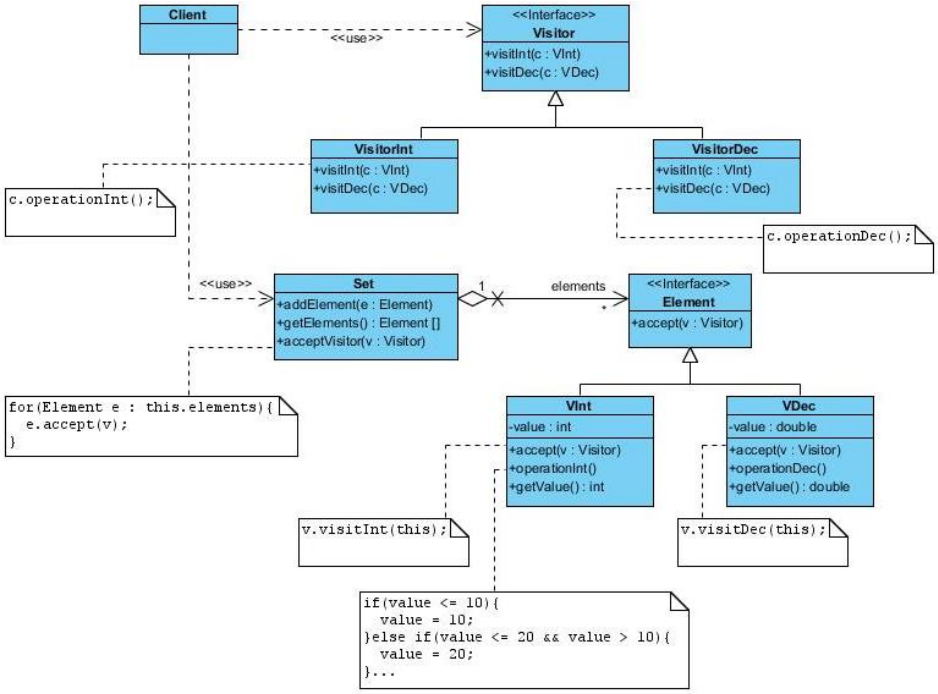

Figura C.18: *Exemplo de Utilização do Visitor (adaptada de [Gamma et al., 1995])*

Anexo D

Esboços da interface do PatGer

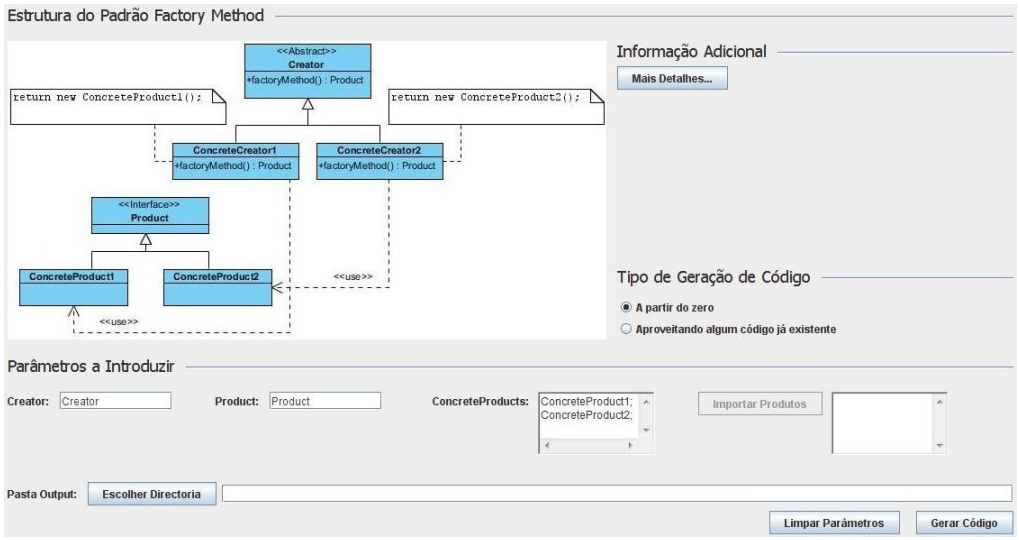

Figura D.1: *Esboço da interface do Factory Method*

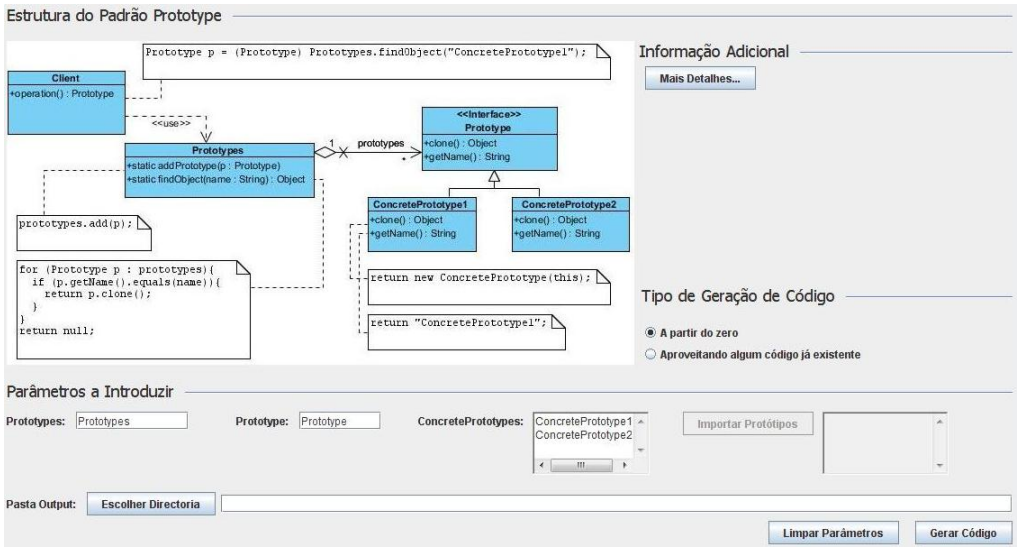

Figura D.2: *Esboço da interface do Prototype*

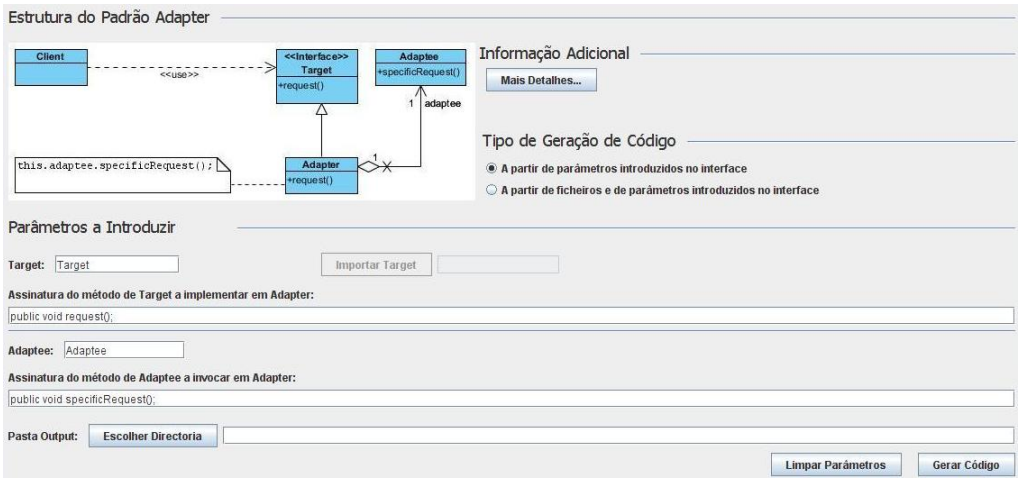

Figura D.3: *Esboço da interface do Adapter*

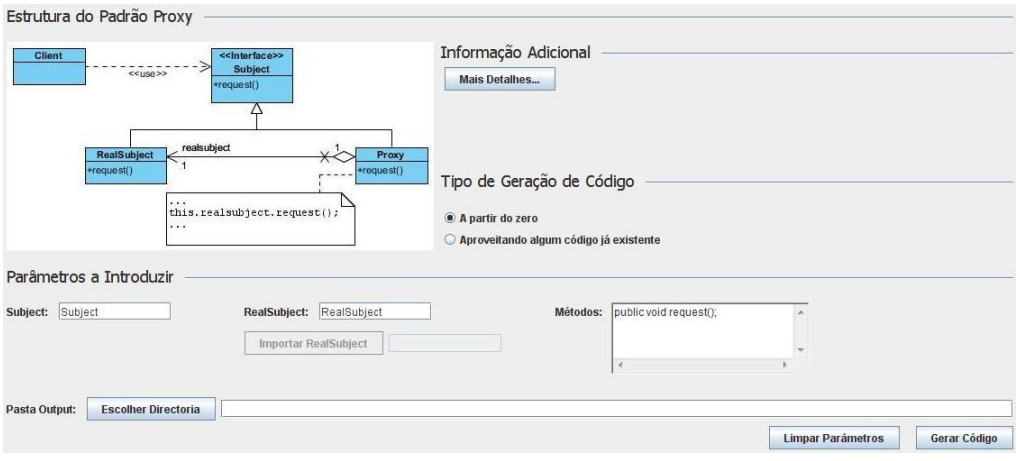

Figura D.4: *Esboço da interface do Proxy*

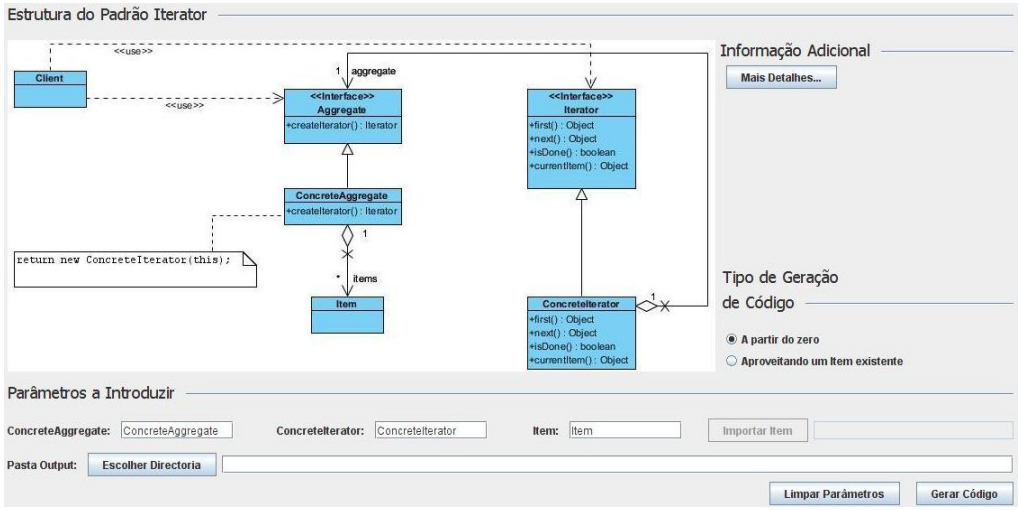

Figura D.5: *Esboço da interface do Iterator*

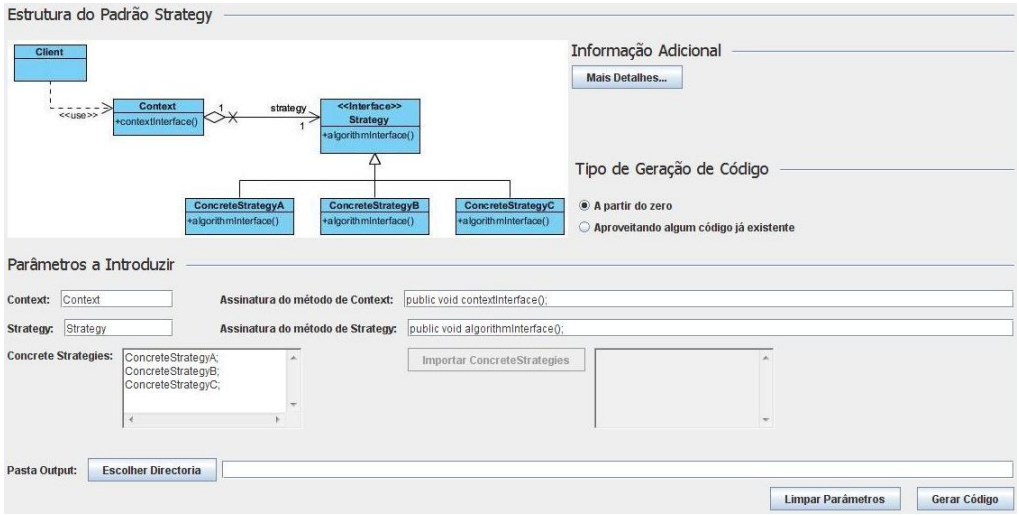

Figura D.6: *Esboço da interface do Strategy*

Anexo E

Diagramas de Sequência do PatGer

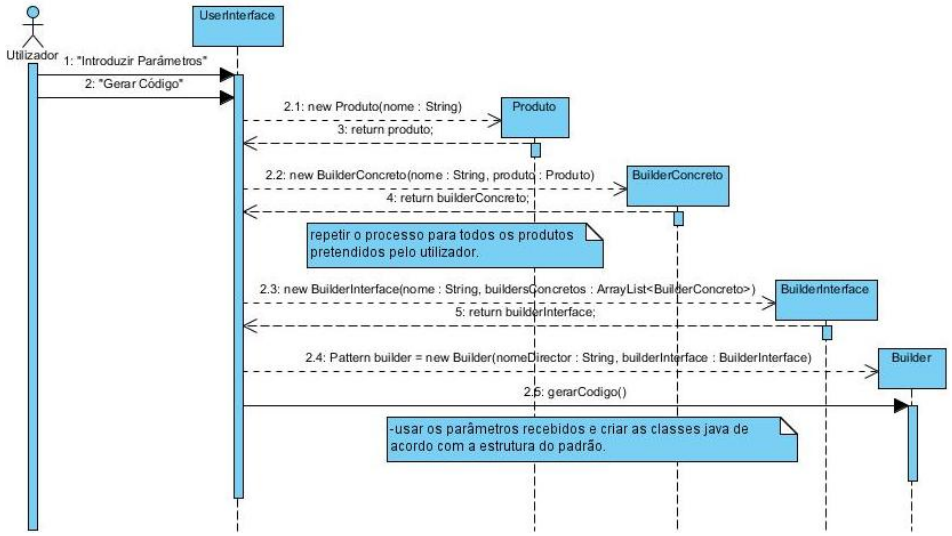

Figura E.1: *Diagrama de Sequência do Builder*

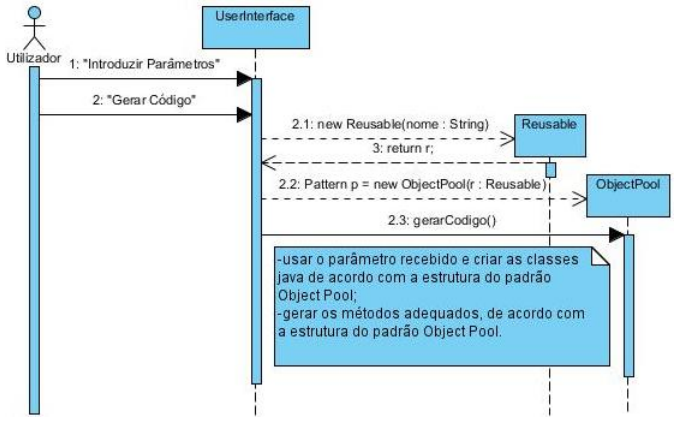

Figura E.2: *Diagrama de Sequência do Object Pool*

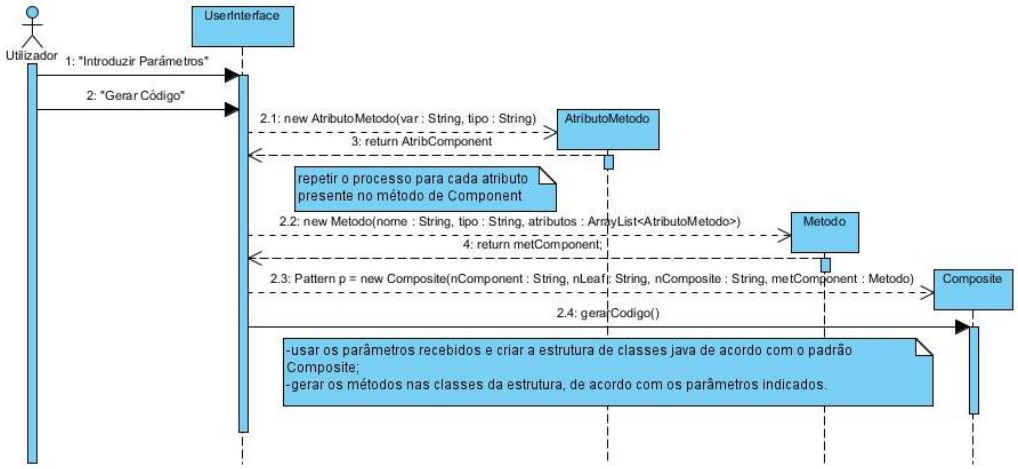

Figura E.3: *Diagrama de Sequência do Composite*

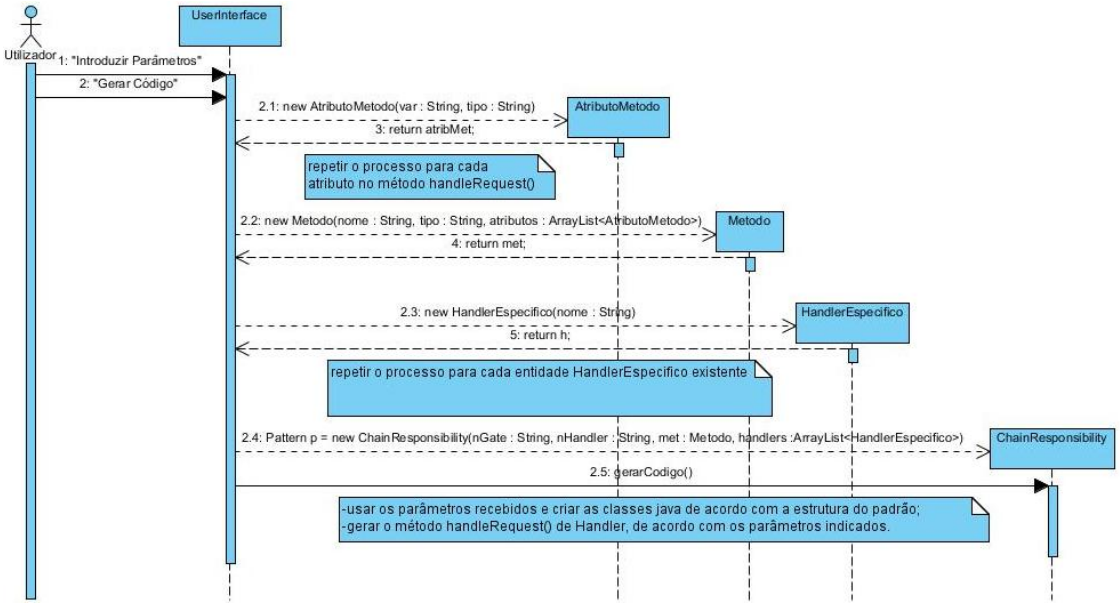

Figura E.4: *Diagrama de Sequência do Chain of Responsibility*

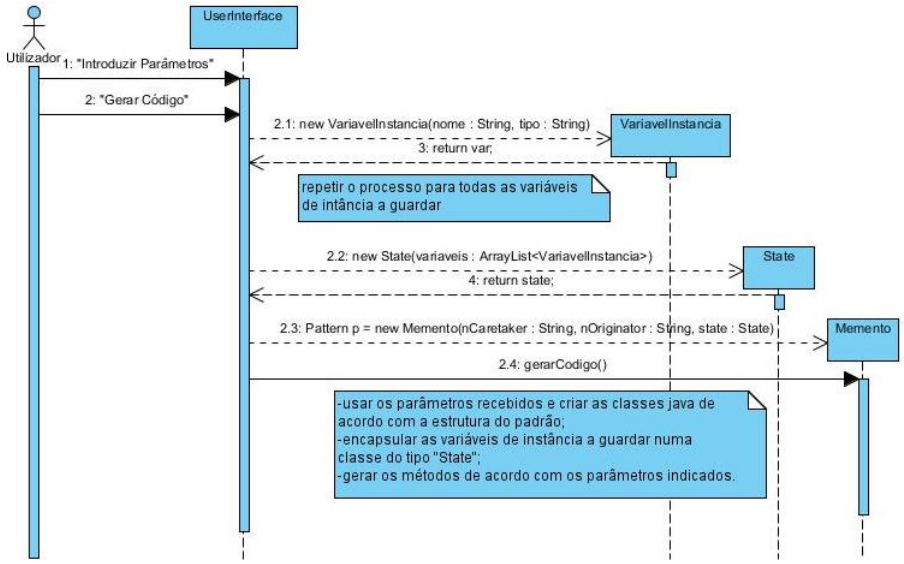

Figura E.5: *Diagrama de Sequência do Memento*

Anexo F

Diagramas de Classes do PatGer

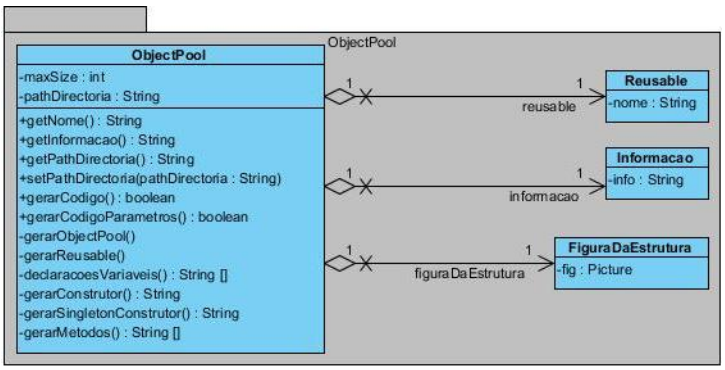

Figura F.1: *Package Object Pool*

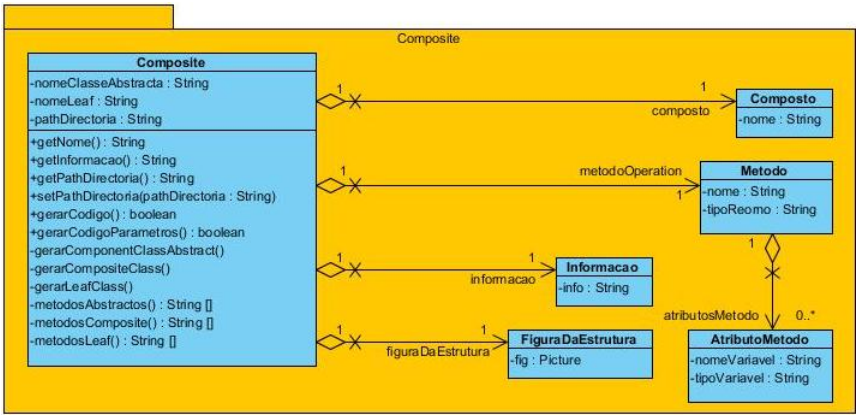

Figura F.2: *Package Composite*

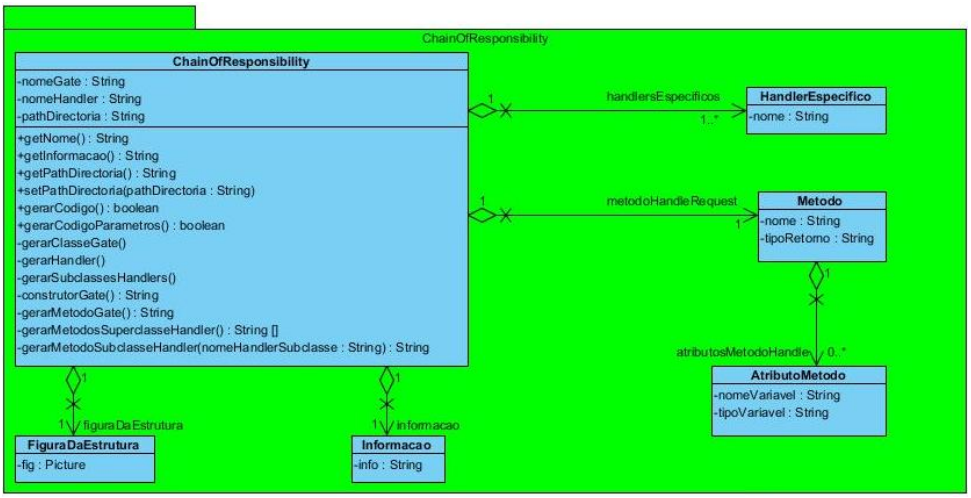

Figura F.3: *Package Chain of Responsibility*

Anexo G

Imagens do PatGer

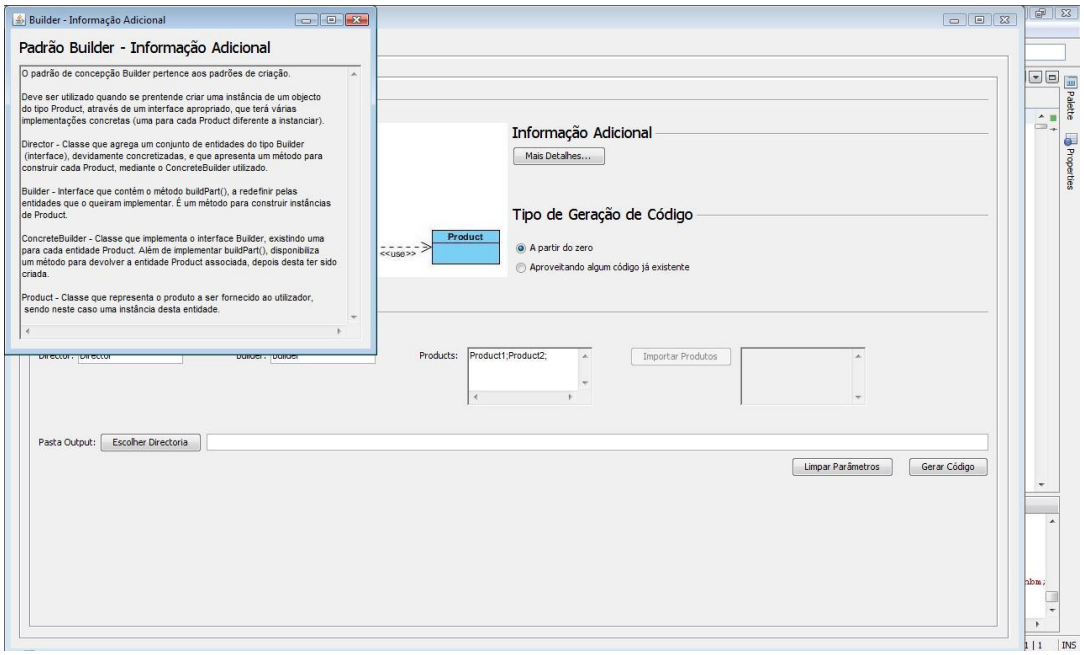

Figura G.1: *Informação adicional do Builder no PatGer*

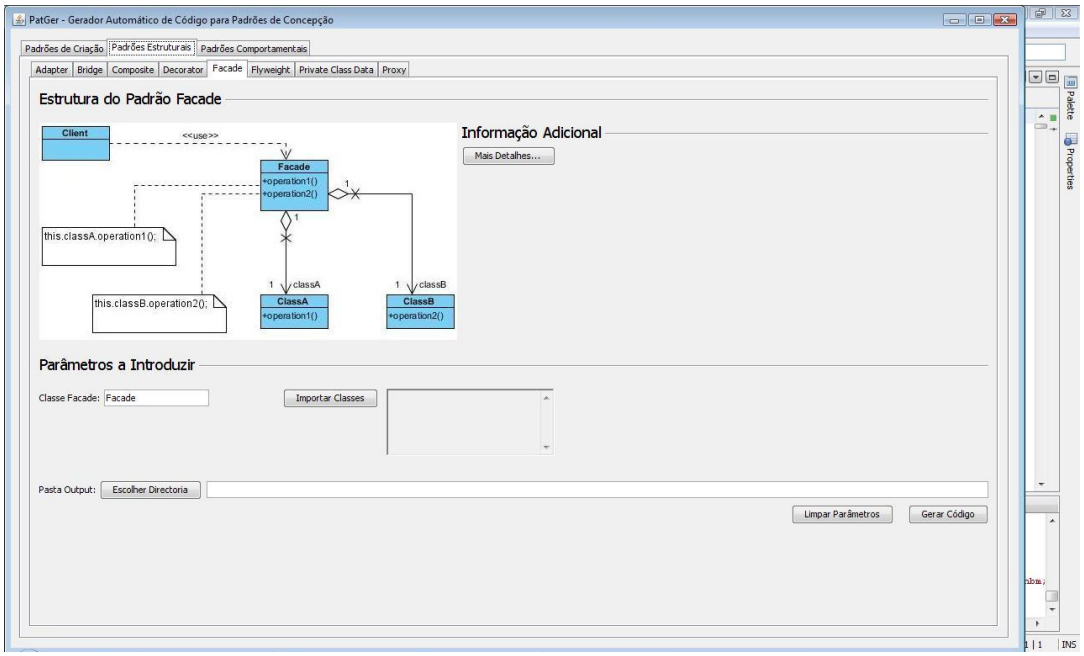

Figura G.2: *Escolha do Facade no PatGer*

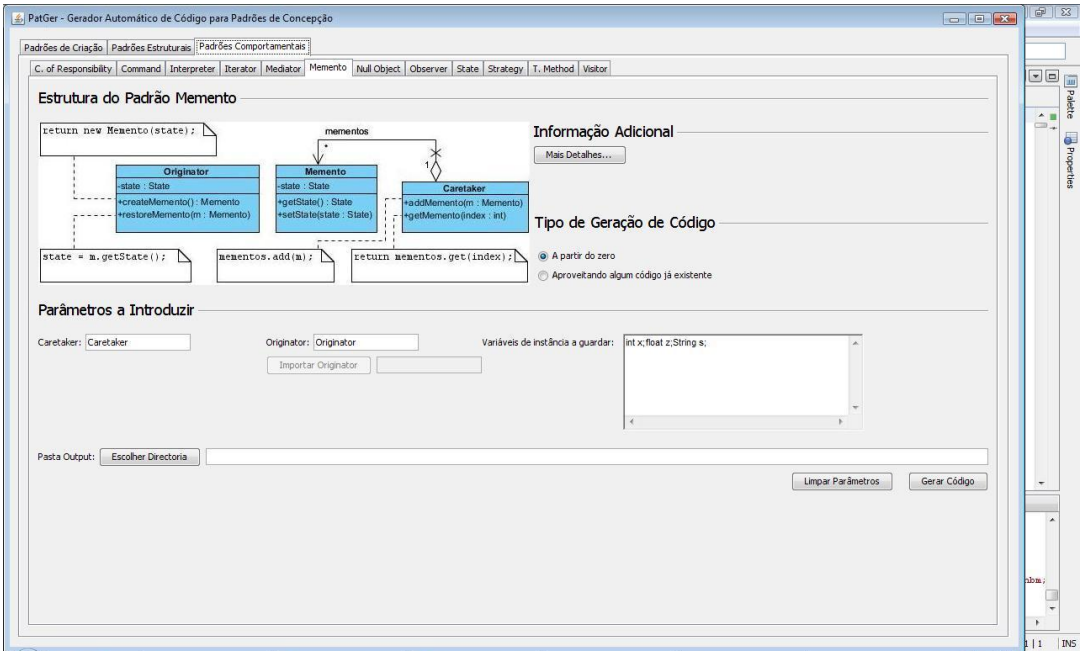

Figura G.3: *Escolha do Memento no PatGer*