3
Técnicas de Implementação de Variabilidades

"O começo de todas as ciências é o espanto das coisas serem o que são."
(Aristóteles)

Neste capítulo são apresentadas as técnicas de implementação de variabilidades utilizadas neste trabalho. Primeiramente, são apresentados os conceitos de *frameworks* na Seção 3.1. A técnica de compilação condicional é apresentada na Seção 3.2. A Seção 3.3 apresenta a técnica de arquivos de configuração com injeção de dependências. Na Seção 3.4 são discutidos alguns conceitos importantes relacionados a programação orientada a aspectos. Por fim, na Seção 3.5 é feito um breve resumo sobre o capítulo apresentado.

3.1
Frameworks

Frameworks (Fayad et al. 1999, Johnson 1997, Greenfield et al. 2004) são largamente utilizados na implementação de LPS. Existem várias definições de *frameworks*, uma delas é a seguinte: "Uma arquitetura desenvolvida com o objetivo de se obter a máxima reutilização, representada como um conjunto de classes abstratas e concretas e com grande potencial de especialização" (Mattsson and Bosch 2000). Os *frameworks* provêem uma solução organizada para uma família de problemas semelhantes contendo os pontos fixos (*frozen spots*) e os pontos flexíveis (*hot spots*). Os pontos fixos representam um conjunto de funcionalidades padrões já implementadas e fornecidas pela arquitetura do *framework*. Os pontos flexíveis são as partes que podem ser customizadas e estendidas de acordo com as particularidades de cada aplicação. Essa customização se dá através da extensão de classes abstratas, implementação de interfaces e configuração de classes existentes. Assim, as aplicações são instanciadas através do reuso da arquitetura definida pelo *framework* e da implementação/extensão dos seus respectivos pontos flexíveis.

Os *frameworks* possuem a característica de permitir o reuso em larga escala, provendo reutilização tanto em nível de código quanto de projeto em um determinado domínio. Sendo assim, é possível aumentar a produtividade através do uso de *frameworks*, pois as aplicações podem ser instanciadas de

forma mais rápida, devido a existência de uma arquitetura semi-pronta já validada e testada.

Os *frameworks* podem ser classificados em verticais e horizontais (Fayad et al. 1999). Os *frameworks* verticais são específicos de um determinado domínio. São também chamados de *frameworks* especialistas. Já os *frameworks* horizontais não são específicos de nenhum domínio, podendo ser utilizados em diversos domínios, por exemplo, o *framework* Hibernate (Middleware 2006), que é usado para persistência de objetos em banco de dados relacionais.

Além das classificações mencionadas acima, os *frameworks* também podem ser classificados de acordo com as técnicas utilizadas na instanciação das aplicações, isto é, da implementação de seus pontos flexíveis, que são: caixa-branca (*white-box*), caixa-preta (*black-box*) e caixa-cinza (*gray-box*). Nos *frameworks* caixa-branca, a instanciação das aplicações ocorre através da utilização de herança e implementação de interfaces. Nos *frameworks* caixa-preta, a instanciação de aplicações ocorre através do mecanismo de composição e configuração de algum tipo de arquivo (eg. XML). Finalmente, o *framework* caixa-cinza é a mistura do *framework* caixa-branca e caixa-preta. Sendo assim, as técnicas de instanciação das aplicações ocorrem através de herança e composição/configuração.

No contexto deste trabalho, dois *frameworks* para o desenvolvimento de SMA são utilizados: JADE (JADE 2001) e Jadex (Braubach and Pokahr 2002).

3.2 Compilação Condicional

A técnica de compilação condicional é baseada no conceito de diretivas pré-processadas. Essas diretivas indicam se um determinado trecho de código irá compilar ou não, dependendo do valor das variáveis ou expressões contidas nas diretivas. Elas podem ser usadas em qualquer lugar do arquivo de uma classe, métodos, trechos de código ou mesmo toda a classe. Esta técnica é muito utilizada no domínio de aplicações para dispositivos móveis. A ferramenta de compilação condicional utilizada neste trabalho foi o Antenna (Web 2004). Esse pré-processador é semelhante ao existente na linguagem C e em outras linguagens como C++ e C#. Diversas diretivas são suportadas pelo pré-processador Antenna, tais como *#ifdef, #endif, #ifndef* e etc.

Abaixo são ilustrados dois exemplos (Códigos 3.1 e 3.2) utilizando a técnica de compilação condicional. Estes trechos de código são referentes a LP-SMA EC (Seção 5.1). No Código 3.1, caso o usuário escolha que seu produto terá revisores adicionais no seu sistema de gerenciamento de conferência, o

trecho de código irá compilar. Para este exemplo, as diretivas foram colocadas dentro de um método. No Código 3.2, as diretivas foram colocadas para os atributos.

Código 3.1: Exemplo com compilação condicional.

```
1  public void acceptPaper(...) throws RegisterException {
2      Review persistentReview = this.read(review.getId());
3      if (review.getCommiteeMemberAccepted() != null) {
4          if (review.getCommiteeMemberAccepted().booleanValue()) {
5              persistentReview.setCommiteeMemberAccepted(Boolean.
                   TRUE);
6          }
7      }
8      \#ifdef REVIEWER
9       if (review.getReviewerAccepted() != null) {
10         if (review.getReviewerAccepted().booleanValue()) {
11             peristentReview.setReviewerAccepted(Boolean.TRUE);
12         }
13     }
14     \#endif
15 }
```

Código 3.2: Exemplo com compilação condicional.

```
1  public class Review extends PersistentObject {
2      ...
3      private CommitteeMember committeeMember;
4      private Paper paper;
5      \#ifdef REVIEWER
6      private Reviewer reviewer;
7      private Boolean reviewerAccepted;
8      \#endif
9      ...
10 }
```

3.3 Arquivos de Configuração com Injeção de Dependências

A técnica de arquivos de configuração com injeção de dependências utiliza principalmente os conceitos de herança e polimorfismo. Esses conceitos são utilizados para permitir vários tipos de implementações, configuração de diferentes produtos na LPS. A injeção de dependência é um padrão usado para se manter um baixo nível de acoplamento entre os diferentes módulos do sistema. No contexto deste trabalho, o *framework* Spring (Source 2008) foi

utilizado com esta finalidade. O Spring possui uma arquitetura baseada em interfaces e POJOs (*Plain Old Java Objects*). Nele, a injeção de dependência pode ocorrer através de três formas: construtores, métodos *setters* e por interface. No Código 3.3 é apresentado um exemplo de injeção de dependência via método `set` usando o *framework* Spring. De acordo com este exemplo, o método `set` recebe um objeto do tipo `PhoneFinder`.

Código 3.3: Injeção por método *set* usando Spring.

```
1  public class PhoneLister {
2      private PhoneFinder finder;
3
4      public setFinder(PhoneFinder finder) {
5          this.finder = finder;
6      }
7  }
8
9  public class MySqlPhoneFinder implements PhoneFinder {
10     public MySqlPhoneFinder(String dbName) {
11         this.dbName = dbName;
12     }
13     ...
14 }
```

Código 3.4: Configuração do arquivo XML.

```
1  <beans>
2      <bean id="PhoneLister" class="spring.PhoneLister">
3          <property name="finder">
4              <ref local="PhoneFinder"/>
5          </property>
6      </bean>
7      <bean id="PhoneFinder" class="spring.MySqlPhoneFinder">
8          <property name="dbName">
9              <value>myDBName</value>
10         </property>
11     </bean>
12 </beans>
```

Os arquivos XML são utilizados para se definir as dependências entre os módulos do sistema. O Spring então se encarrega de "injetar" em cada componente as dependências declaradas nesses arquivos. No Spring, qualquer objeto da aplicação que está sob seu controle é considerado um *bean*. Na configuração do arquivo XML (Código 3.4), a *tag ref* é utilizada para referenciar outro *bean*. Assim, a referência do *bean* `PhoneFinder` é injetada no método `set` da classe `PhoneLister`.

3.4 Programação Orientada a Aspectos

Com o objetivo de desenvolver software com uma certa complexidade, de qualidade e capaz de se adaptar as mudanças que os paradigmas de programação evoluíram. Essa evolução ocorreu principalmente com o intuito de fornecer atributos específicos na modularização dos requisitos do sistema. O modelo de programação estruturado, por exemplo, utiliza o conceito de procedimentos e funções para fornecer as abstrações necessárias para o desenvolvimento das ações e criar módulos que realizem tarefas específicas. A programação orientada a objetos (POO), por sua vez, possui um grau de modularização superior ao paradigma estruturado, devido ao conceitos de classes, objetos, polimorfismo, herança, dentre outros (Booch 2004).

Entretanto, alguns interesses (*concerns*) não são tão bem modularizados na POO (Kiczales et al. 1997, Tarr et al. 1999). Um interesse é uma parte do problema que queremos tratar como uma unidade conceitual única na solução do software (Dijkstra 1997). Um interesse pode ser um requisito, uma propriedade ou mesmo uma funcionalidade de um sistema. Esses interesses podem ser funcionais ou não funcionais como: *logging*, tratamento de exceções, persistência, dentre outros. Dentre os problemas ocorridos com a implementação de sistemas orientados a objetos (Kiczales et al. 1997, Tarr et al. 1999), tem-se:

- **Entrelaçamento**: uma classe que manipula vários interesses simultaneamente.
- **Espalhamento**: um interesse é implementado em várias classes.

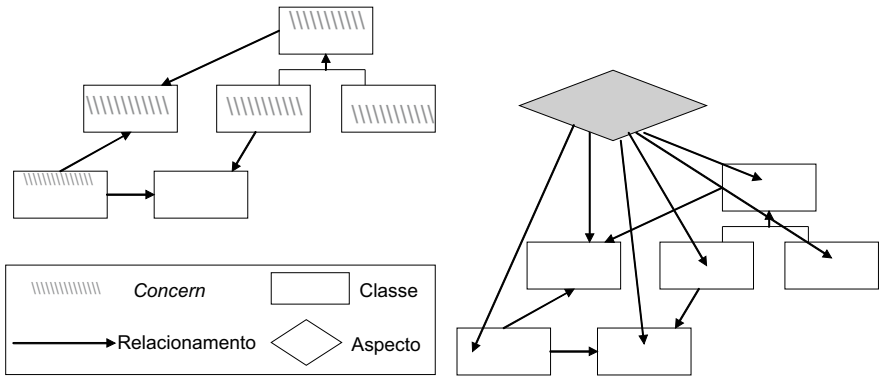
O Código 3.5 apresenta um exemplo clássico encontrado na literatura, que é o registro de operações (*logging*). O *log* está espalhado nos métodos `debitar` e `creditar`. O *logging* está normalmente espalhado e entrelaçado com outros interesses em um sistema OO.

Esses tipos de problemas diminuem consideravelmente a qualidade do software. Essa diminuição da qualidade pode ser observada através dos seguintes fatores: dificuldade de manutenção e evolução do software, dificuldade de compreensão, baixo grau de reuso e baixa extensibilidade. Dado isso, com o intuito de prover uma melhor separação de interesses e resolver os problemas na implementação de programas OO citados acima, surgiu o conceito de programação orientada a aspectos (POA) (Kiczales et al. 1997). A POA é um paradigma complementar ao paradigma OO e usa uma nova abstração chamada aspecto. Além disso, tal paradigma tem o objetivo de aumentar a reusabilidade e a facilidade de evolução. Como pode ser visto na Figura 3.1(a), o interesse

encontra-se espalhado por diversos módulos na POO. A POA é capaz de modularizar este interesse em um aspecto que se associa dinamicamente a esses módulos (Figura 3.1(b)).

Código 3.5: Exemplo de um interesse: Logging

```
1  public class Conta {
2      private String numero;
3      private double saldo;
4      ...
5      public void debitar (double valor) {
6          this.setSaldo(this.getSaldo() - valor);
7          log.info(``ocorreu um débito!'');
8      }
9      public void creditar (double valor) {
10         this.setSaldo(this.getSaldo() + valor);
11         log.info(``ocorreu um crédito'');
12     }
13     ...
14 }
```



3.1(a): POO 3.1(b): POA

Figura 3.1: Implementação de um interesse com POO e POA.

A POA utiliza aspectos como uma nova abstração e provê um novo mecanismo para compor classes e aspectos. Atualmente, existem diversas linguagens de POA: a linguagem AspectJ (Laddad 2003), HyperJ (Tarr and Ossher 2000) e CaesarJ (Mezini and Ostermann 2004a). Um aspecto é definido como uma abstração capaz de modularizar melhor os interesses do sistema. Os aspectos conseguem agrupar fragmentos de código associados a um conjunto de classes. Um aspecto é similar a uma classe e pode também ser especializado por outros aspectos. Um aspecto é composto de *advices*, *inter-type declarations*, *pointcuts*, atributos e métodos. A seguir segue uma breve descrição dos principais conceitos:

- ***Adendos (Advices)***: são utilizados para declarar o código que deve ser executado quando um *join point* especificado em um *pointcut* for atingido. Um *advice* é semelhante a um método e altera dinamicamente o comportamento de classes. Há três tipos de *advices*: (i) *before*: executado sempre que o *join point* é alcançado; (ii) *after*: executado depois que o método tiver sido executado; e (iii) *around*: são executados sempre que um *join point* for alcançado. Essa regra de junção é mais complexa, pois mantém o controle explícito do prosseguimento de execução do *join point*;
- ***Declarações inter-tipos (Inter-type declaration)***: especificam novos membros (atributos e métodos) e relações que afetam a estrutura e a hierarquia de classes do programa. Os *inter-type declarations* alteram o comportamento das classes de forma estática, em tempo de compilação;
- ***Pontos de Junção (Join Points)***: são pontos bem definidos na execução do programa. É a especificação de como as classes e os aspectos se relacionam. Alguns exemplos de *join points* são: chamada de métodos, as execuções de método e instanciação de objetos;
- ***Pontos de Corte (Pointcuts)***: são os elementos usados para especificar os *join points*. Indicam os pontos que queremos interceptar.

A linguagem AspectJ é uma extensão para a linguagem Java (Laddad 2003) e foi criada na empresa XEROX PARC (*Palo Alto Research Center*). O Código 3.6 ilustra o aspecto para modularização do interesse *logging* (Código 3.5). O aspecto `LoggingAspect` (Código 3.5) contém dois *pointcuts*: (i) `logCredito`: responsável por interceptar as chamadas feitas ao método `creditar`; e (ii) `logDebito`: responsável por interceptar as chamadas feitas ao método *debitar*. Nesse aspecto existem dois *after advices* associados aos *pointcuts* `logCredito` e `logDebito`. Assim, o interesse *logging* encontra-se melhor modularizado no aspecto `LoggingAspect`.

Recentemente, a programação orientada a aspectos (POA) tem sido investigada como uma técnica de programação importante na modularização e gerenciamento de *features* em LPS (Anastasopoulos and Muthig 2004, Alves et al. 2006, Alves et al. 2005, Griss 2000, Mezini and Ostermann 2004b), *frameworks* (Kulesza et al. 2006a) e em SMA (Garcia et al. 2003a). A implementação das variabilidades em LPS usando POA permite a possibilidade da ativação/desativação das *features* da arquitetura base da LPS, simplificando assim, a complexidade do desenvolvimento de arquiteturas de LPS.

Código 3.6: Exemplo do aspecto LoggingAspect.

```
1  public aspect LoggingAspect {
2      pointcut logCredito(): call (* Conta.creditar(double));
3      pointcut logDebito(): call (* Conta.debitar(double));
4
5      after(): logCredito() {
6          log.info(``ocorreu um crédito'');
7      }
8      after(): logDebito(){
9          log.info(``ocorreu um débito'');
10     }
11 }
```

A combinação de artefatos e variabilidades de produtos é uma tarefa bastante desafiadora (Anastasopoulos and Muthig 2004). Alguns desafios centrais em relação às variabilidades são (Voelter and Groher 2007): identificação, tipo de variabilidade, descrição, gerenciamento e implementação. Alguns trabalhos vêm utilizando a POA na derivação automática de produtos de LPS (Anastasopoulos and Muthig 2004, Cirilo et al. 2007, Voelter and Groher 2007) devido ao fato de permitir uma melhor modularização no nível de código, modelo e templates.

No desenvolvimento de aplicações baseadas em agentes, principal escopo deste trabalho, o aumento da complexidade motiva o uso de POA, pois permite evitar código replicado, espalhado e entrelaçado. Esses tipos de problemas podem causar dificuldades para gerenciamento, manutenção e reuso de *features* em LPS. Dentre os benefícios do uso de POA no desenvolvimento de aplicações complexas, podemos citar: menor replicação de código, melhor separação de interesses, menor acoplamento e maior coesão entre os módulos.

3.5 Resumo

Ao longo do capítulo apresentamos algumas técnicas de implementação de variabilidades utilizadas no contexto de LPS. Essas técnicas são muito úteis para permitir uma melhor gerência de variabilidades em LPS. *Frameworks* são caracterizados por possuírem uma arquitetura flexível semi-pronta que pode ser customizada pelas aplicações de acordo com as suas necessidades. Essa customização ocorre através da implementação/extensão dos pontos flexíveis especificados. A técnica de compilação condicional é baseada em diretivas pré-processadas que indicam se um determinado trecho de código deverá compilar ou não. Arquivos de configuração do *framework* Spring são usados para permitir a injeção de dependências dentro dos pontos de variação da

arquitetura de LPS. O paradigma orientado a aspectos provê mecanismos e abstrações para permitir uma melhor separação de interesses. Este capítulo também apresentou alguns conceitos básicos relacionados a POA, utilizando principalmente a linguagem AspectJ como exemplo. Os aspectos basicamente são compostos por métodos *advices* (*before*, *after* ou *around*), *inter-type declarations*, *pointcuts*, atributos e métodos.