Uso de Aspectos para Verificar Regras de Instanciação de Frameworks

André Dantas Rocha[1]
Valter Vieira de Camargo[1]
Paulo Cesar Masiero[1]

Resumo: A instanciação de frameworks normalmente é um processo que demanda tempo e é sujeito a erros, pois existem regras específicas que devem ser seguidas com o intuito de produzir uma aplicação compatível com seus requisitos. Algumas dessas regras referem-se a políticas de implementação e não podem ser verificadas em tempo de compilação, como por exemplo regras de nomenclatura e construtores padrão. Um problema relevante é que as falhas geradas pela não obediência a essas regras só podem ser detectadas em um momento posterior à instanciação, durante o uso do sistema. É apresentada neste artigo a implementação de um mecanismo que verifica regras de instanciação de um framework de persistência.
Palavaras-chave: POA, frameworks, regras de instanciação

Abstract: Instantiation of frameworks is usually a time-consuming, error-prone process as there are specific rule types that must be followed to produce an application that meets its requirements. Some of these rules refer to framework specific implementation policies and cannot be verified in compilation time, such as nomenclature rules and default constructors. A relevant problem is that faults generated when these rules are not applied can only be detected later, during the system usage. A mechanism to enforce that certain rules for a persistency framework be strictly followed, is shown.
Keyword: AOP, frameworks, instantiation rules

1 Introdução

Frameworks são softwares que permitem o reuso da análise, projeto, implementação e teste de aplicações de um determinado domínio por meio de uma atividade denominada instanciação, na qual os pontos variáveis (*hot spots*) do framework são definidos por meio de ganchos (*hooks*) para gerar a aplicação final. Esses ganchos geralmente são implementados por meio de classes e métodos abstratos, com base em padrões de projeto [6], e são concretizados na aplicação. Há basicamente dois tipos de frameworks: caixa branca e caixa preta. O primeiro tipo é instanciado por meio da especialização de classes específicas, enquanto que o segundo é geralmente instanciado por meio de parametrização [5].

[1]Instituto de Ciências Matemáticas e de Computação – ICMC/USP, Caixa Postal 668
{[rocha,valter,masiero]@icmc.usp.br}

Durante o processo de instanciação de um framework, há diversas regras que devem ser obedecidas para que a aplicação seja gerada com sucesso. Algumas dessas regras referem-se a políticas de implementação exigidas pelo framework e, geralmente, não podem ser verificadas em tempo de compilação. Nesse contexto, um problema que merece destaque é que as falhas decorrentes do não cumprimento dessas regras só serão detectadas posteriormente, durante a execução do sistema, pois a aplicação instanciada pode funcionar ainda que essas regras não sejam obedecidas.

A Programação Orientada a Aspectos (POA)[11] oferece recursos que podem ser utilizados para verificar se as regras de instanciação de um framework foram seguidas. A captura do contexto de execução dos métodos e o acesso a propriedades estáticas do programa são exemplos de recursos úteis que podem ser utilizados para tal fim. Além disso, a verificação do cumprimento das regras de instanciação pode ser considerado como um interesse (*concern*) e pode ser modularizado sob a forma de aspectos.

É ilustrado neste artigo como a linguagem AspectJ [10] pode ser utilizada para verificar a corretitude da instanciação de frameworks caixa branca. Um processo de instanciação de um Framework Orientado a Aspectos (FOA) é usado para exemplificar a verificação das regras. O artigo encontra-se organizado da seguinte forma: na Seção 2 são abordadas algumas propostas para a verificação de propriedades de programas usando POA. Na Seção 3 descreve-se o framework de persistência [2]. Na Seção 4 é apresentado o módulo de verificação proposto. Por fim, são apresentados o estágio atual do trabalho e perspectivas futuras na Seção 5.

2 Uso de Aspectos para a Verificação de Propriedades de Software

Em geral, o desenvolvimento de software segue determinadas regras que são refletidas na implementação sob a forma de propriedades. Essas regras são ditadas tanto pelo ambiente de desenvolvimento quanto por políticas organizacionais e a sua verificação, durante o desenvolvimento, pode antecipar potenciais problemas no software. Tendo em vista que essas regras são concretizadas na forma de propriedades, uma maneira de verificar se uma regra específica foi atendida é por meio da verificação da propriedade correspondente no código. Basicamente, três tipos de propriedades podem ser distinguidas na verificação de um programa [4]:

1. *Propriedades técnicas*, que quando violadas levam o software a falhar durante sua execução;
2. *Propriedades de ambiente*, que asseguram que o software execute apropriadamente em um determinado ambiente. Sua violação não leva necessariamente a uma falha no sistema;

3. *Propriedades políticas*, que asseguram consistência no projeto por meio do uso de "boas práticas de programação".

Segundo Eichberg et al. [4], as propriedades de um software podem ser checadas estaticamente, em tempo de compilação, ou dinâmicamente, em tempo de execução. Checagens em tempo de compilação baseiam-se na análise estática do programa e portanto os resultados são válidos para qualquer execução do programa, independentemente dos dados de entrada fornecidos. Algumas checagens, no entanto, envolvem valores só disponíveis durante a execução e portanto não podem ser realizadas dessa forma. As checagens dinâmicas, ainda que exijam a execução do programa, antecipam potenciais problemas que podem ocorrer posteriormente, durante o funcionamento do sistema ou que, sob condições específicas, não poderiam ser detectados.

A implementação de um mecanismo que realize essa verificação de propriedades utilizando apenas orientação a objetos leva ao entrelaçamento e espalhamento do código que implementa esse mecanismo com o código que implementa a funcionalidade do sistema. A implementação de assertivas usando orientação a objetos é um exemplo que ilustra esse problema. Evidencia-se assim que a "verificação de propriedades" representa um interesse transversal e que, portanto, pode ser modularizado sob a forma de aspectos.

A POA tem sido utilizada por alguns autores para verificação de certas propriedades em programas convencionais [1, 9, 12]. Isberg [9], por exemplo, utiliza declarações de erro do AspectJ para garantir que atributos de uma classe só sejam acessados via métodos específicos (*get*, *set*), preservando assim o encapsulamento. Além disso, restringe a criação de objetos a métodos Fábrica (*Factory*), abordagem interessante quando se busca controlar a criação de novas instâncias.

Em relação à análise estática, Gradecki and Lesiecki [7] utilizam o AspectJ para verificar tentativas de mudança de estado em uma instância de uma linha de execução (*Thread*), o que não deve ocorrer em *Servlets* Java sob pena de perda de dados. Eichberg et al. [4] usam a POA em nível de *bytecode* Java para garantir políticas e consistência na elaboração do software. Os autores propõem um framework que auxilia a verificação dos três tipos de propriedades citadas acima, garantindo checagens que não podem ser feitas utilizando somente reflexão computacional.

Outra abordagem que utiliza a POA para verificar certas propriedades do software é proposta por Diotalevi [3]. Nela o autor implementa os conceitos do Projeto por Contrato (*Design by Contract*) utilizando o AspectJ. O objetivo é verificar por meio de invariantes, pré e pós condições se determinadas propriedades de ambiente são satisfeitas em tempo de execução.

Além das abordagens destacadas anteriormente, pode-se citar a ferramenta *PatternTesting* [12], que utiliza verificações estáticas e em tempo de execução para assegurar que as "melhores práticas" de arquitetura e projeto de software foram seguidas.

Embora aspectos sejam utilizados para a verificação de propriedades em programas convencionais, ainda não se encontram abordagens que utilizem essa técnica para a verificação das regras de instanciação de frameworks. Assim como softwares convencionais, a instanciação de frameworks caixa branca, em particular, possui determinadas regras que devem ser seguidas, o que pode ser analisado por meio da verificação das propriedades que a aplicação gerada deve possuir.

3 O Framework de Persistência

Para exemplificar a implementação do interesse de verificação é utilizado um framework de persistência, desenvolvido em AspectJ [10]. Esse framework fornece a infraestrutura básica para persistir objetos em um banco de dados relacional sem que o código de persistência fique espalhado e entrelaçado com o código da aplicação. Suas variabilidades incluem: a política de persistência (controlada ou não pelo framework); o repositório de conexões (*pooling*); e *caching* para as operações de persistência com alto custo computacional. Independentemente da variabilidade escolhida, há um conjunto de regras de instanciação que devem ser seguidas:

1. Implementar classes da aplicação que especializam classes específicas do framework;
2. Criar métodos e construtores para as novas classes da aplicação;
3. Seguir um determinado padrão de nomenclatura de classes, métodos e atributos;
4. (Re)definir métodos herdados, com vistas a incorporar comportamento inexistente ou incompleto nas classes pai;
5. Implementar métodos de acesso aos atributos (*sets* e *gets*) nas classes de aplicação;
6. Criar uma tabela no banco de dados com o nome da classe de aplicação persistente correspondente;
7. Criar colunas na tabela com os mesmos nomes dos atributos da classe de aplicação persistente correspondente;
8. Concretizar pontos de corte referentes à conexão com o banco de dados.

Embora essas regras sejam obrigatórias para a instanciação do framework em questão, os itens de 1 a 5 geralmente são encontradas no processo de instanciação de diversos frameworks e podem ser considerados genéricos. A fim de verificar regras genéricas e específicas de instanciação de frameworks, foi implementado em AspectJ um módulo denominado *Instantiation Checker*.

4 Verificação de Regras de Instanciação

O interesse de verificação de regras de instanciação foi implementado sob a forma de um módulo, denominado *Instantiation Checker*. Esse módulo é composto de alguns aspectos e classes que são responsáveis por realizar diversas checagens, dentre as quais:

- de nomenclatura de atributos, métodos de acesso a atributos e classes;
- de conformidade de nomes entre atributos e colunas do banco;
- da existência de construtores.

Na Figura 1 pode-se observar um modelo de classes que mostra a instanciação do framework de persistência para a aplicação *Workshop* e também as classes do módulo de verificação. Essa aplicação consiste em um sistema de uma oficina de aparelhos eletrônicos que permite gerenciar diversas atividades relativas a esse negócio. Embora apenas duas classes estejam sendo exibidas (**`Customer`** e **`Equipment`**), elas são suficientes para representar o relacionamento com os demais componentes do sistema. A interface **`PersistentRoot`** representa o framework de persistência, que foi instanciado para permitir que os objetos da aplicação possam ser persistidos. Nessa interface são introduzidas as implementações dos métodos de persistência, por meio de declarações entre-tipos (*Inter-type Declaration*) do AspectJ, o que torna essas operações disponíveis nas classes de aplicação.

Os aspectos relativos ao módulo de verificação encontram-se representados no lado direito da figura. Esse módulo foi utilizado a fim de checar se as regras de instanciação do framework de persistência foram corretamente seguidas durante a criação da aplicação *Workshop*. Nesse módulo foram abordadas as regras 5 e 7, descritas na Seção 3, que foram escolhidas porque representam uma regra genérica e uma específica, respectivamente.

O aspecto *Checker*, exibido na Figura 2, possui a estrutura básica para verificação de regras, de programas convencionais ou de frameworks. Ele foi projetado de forma que possa ser reutilizado em outros contextos, pois é independente de arquitetura e também do próprio framework de persistência. Esse aspecto possui métodos utilitários e um *pointcut* abstrato (**`verifier()`**), que deve ser concretizado pelo instanciador nos aspectos especializados a fim de definir pontos da aplicação que devem ser verificados.

O *pointcut* **`mainVerifier()`** implementa o idioma *Composite Pointcut* [8] e restringe a atuação do *pointcut* **`verifier()`**, impedindo interceptações dentro do pacote **`checker`** (declaração **`within`**) ou quando o aspecto estiver desabilitado (declaração **`if`**). Além disso, o aspecto provê uma forma para habilitar ou desabilitar a verificação, recurso útil quando a aplicação já foi instanciada corretamente e portanto não necessita mais de verificação.

O modelo mostrado na Figura 1 foi projetado de forma a contornar algumas limitações do AspectJ. Como essa linguagem não permite herança entre aspectos concretos, uma camada intermediária, formada pelos aspectos **`AttributeChecker`** e **`GetSetChecker`**, foi criada para dar suporte à redefinição incremental de *pointcuts*. Além disso, os aspectos dessa camada possuem implementações padrão para realizar checagens específicas que atuam sobre *pointcuts* que serão definidos posteriormente nos aspectos especializados. Essas verificações serão detalhadas nas próximas seções.

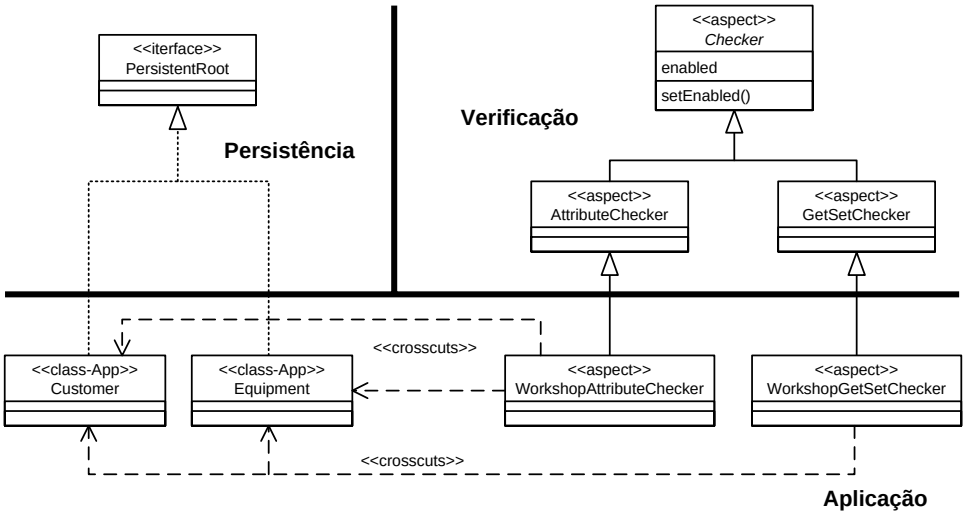

Figura 1. Arquitetura do *Instantiation Checker*

4.1 Verificação de Métodos de Acesso a Atributos

O uso de métodos *get* e *set* nas classes de aplicação é uma prática de programação aconselhada pelo paradigma orientado a objetos pois ajuda a garantir o encapsulamento dos dados. No caso específico do framework de persistência, esses métodos são usados para obter informações sobre o estado de um determinado objeto da aplicação, com a finalidade de executar adequadamente as operações de persistência. A nomenclatura exigida pelo framework

```
1  package checker;
2
3  public abstract aspect Checker {
4    abstract pointcut verifier();
5    pointcut mainVerifier() : verifier() && !within(checker.*) && if(enabled);
6
7    protected static boolean enabled = false;
8    public static void setEnabled(boolean value) { ... }
9    public boolean isEnabled() { ... }
10 }
```

Figura 2. Aspecto *Checker*

para a criação desses métodos consiste no prefixo *get* ou *set*, seguido do nome do atributo com a sua primeira letra em maiúsculo, por exemplo: *getNome()* e *setNome(String)*. O problema que geralmente ocorre é que a não obediência a essa regra de nomenclatura só será identificada quando o objeto for utilizado pelo framework, o que pode ocorrer após muito tempo de operação. Dessa forma, alguma técnica de verificação que auxilie a antecipar a descoberta desse problema é útil para aumentar as chances de corretitude da aplicação instanciada.

No framework em questão, todos os objetos da aplicação são instanciados quando a aplicação é carregada e esse momento mostra-se propício para efetuar essa verificação. Neste sentido, um aspecto foi criado para interceptar a iniciação de objetos e efetuar essa verificação. Na Figura 3 encontra-se implementado o aspecto abstrato **`GetSetChecker`**, que trata do interesse de verificação de métodos de acesso a atributos. O *advice* do tipo *after* atua sobre o *pointcut* composto **`mainVerifier()`**, que foi especificado no aspecto **`Checker`**. Esse *advice* é responsável por obter (via reflexão) os atributos e métodos contidos no objeto instanciado e em seguida efetuar a verificação desejada. Se o método *get* ou *set* correspondente não for encontrado para um determinado atributo, uma exceção será gerada apontando a falha na implementação.

É importante notar que o *pointcut* **`mainVerifier()`** do aspecto **`Checker`** utiliza o *pointcut* componente **`verifier()`**, o qual deve ser concretizado em um aspecto especializado para definir os pontos específicos da aplicação instanciada que devem ser verificados.

O aspecto **`WorkshopGetSetChecker`** é mostrado na Figura 4. O *pointcut* **`verifier()`** é concretizado com o intuito de interceptar a iniciação dos objetos da aplicação *Workshop*, os quais implementam a interface **`PersistentRoot`**. Esse *pointcut* é um dos componentes do *pointcut* composto **`mainVerifier()`**, que foi definido no aspecto abstrato **`Checker`**.

```
1  package checker;
2
3  public abstract aspect GetSetChecker extends Checker {
4
5    after() : mainVerifier() {
6      Object o = thisJoinPoint.getTarget();
7      ...
8      if (!getMethodFound) {
9        throw new ImplementationException(
10         "ATENÇÃO! o framework exige que o método " +  getMethodName +
11         " seja implementado na classe " + o.getClass().getName());
12     }
13     if (!setMethodFound) {
14       throw new ImplementationException(
15          "ATENÇÃO! o framework exige que o método " + setMethodName +
16          "(" + attributeType + ") seja implementado na classe " +
17          o.getClass().getName());
18     }
19   }
20 }
```

Figura 3. Aspecto *GetSetChecker*

4.2 Verificação de Conformidade entre Atributos de uma Classe e Colunas do Banco de Dados

Cada classe de aplicação que utiliza o framework de persistência é mapeada para uma tabela no banco de dados, e cada atributo para uma coluna dessa tabela. Para simplificar o processo de persistência, o framework implementa correspondência direta entre o nome da classe e o nome da tabela e entre os nomes dos atributos da classe e os nomes das colunas correspondentes da tabela. Assim, por exemplo, se uma classe da aplicação possui nome **`Cliente`** e um atributo **`nome`**, o banco deverá conter uma tabela denominada **`Cliente`** com uma coluna denominada **`nome`**. De forma semelhante ao problema descrito no exemplo anterior, uma inconsistência na implementação dessa regra só será apontada quando o framework executar a persistência de um objeto da classe em questão (e algumas vezes após muito tempo de operação), uma vez que o compilador Java não efetua esse tipo de verificação. Aspectos também podem auxiliar nesse tipo de verificação, indicando o problema antes que ele ocorra.

O aspecto **`AttributeChecker`**, mostrado na Figura 5, efetua a verificação da correspondência entre nomes de atributos e nomes de colunas de uma tabela. Um *advice* do tipo *after* captura as informações dos atributos da classe e colunas do banco e procede a veri-

```
1 public aspect WorkshopGetSetChecker extends GetSetChecker {
2
3   pointcut verifier(): initialization(abacoV6.PersistentRoot+.new(..));
4
5 }
```

Figura 4. Aspecto *WorkshopGetSetChecker*

ficação. Assim como no exemplo anterior, qualquer não conformidade à regra resulta no lançamento de uma exceção que alerta sobre esse problema.

```
1  package checker;
2
3  public abstract aspect AttributeChecker extends Checker {
4
5    after(): mainVerifier() {
6      ...
7      Object o = thisJoinPoint.getTarget();
8      ...
9      if (!colFound) {
10       throw new ImplementationException("A coluna " + currentColumn +
11                                         " não foi encontrada na tabela " +
12                                         o.getClass().getName());
13     }
14     ...
15   }
16 }
```

Figura 5. Aspecto *AttributeChecker*

As implementações apresentadas anteriormente poderiam ser elaboradas utilizando apenas recursos providos pela orientação a objetos, e modularizadas em um pacote separado. Uma possível solução seria incluir chamadas ao módulo de verificação dentro dos construtores de cada classe de aplicação (neste caso a execução de cada construtor é considerada um *join point*). Porém, em sistemas complexos o número de construtores afetados pelo código de verificação pode ser muito grande, o que torna essa atividade repetitiva e sujeita a erros. Além disso, as chamadas ao módulo de verificação ficam entrelaçadas com o código funcional e espalhadas pelos diversos módulos do sistema.

A abordagem apresentada neste artigo não resulta nos problemas citados acima, pois a chamada ao código de verificação é eliminada dos construtores e implementada em um aspecto separado. Essa inversão de dependência, propiciada pela orientação a aspectos [13], torna possível especificar esses *join points* em um único local, eliminando o espalhamento desse interesse pelo sistema e portanto diminuindo as dificuldades de instanciação. Além disso a modularização do interesse de verificação facilita a evolução do próprio módulo quando novas características e regras forem incorporadas ao framework.

5 Considerações Finais

O processo de instanciação de um framework exige que diversas regras sejam obedecidas durante a sua instanciação, o que torna muitas vezes o processo de instanciação uma atividade custosa e sujeita a erros. É possível efetuar correções com um menor custo quando o não cumprimento dessas regras pode ser detectado em tempo de compilação ou em fases iniciais de operação. No entanto, existem algumas regras que quando não obedecidas não levam necessariamente a um erro de compilação ou a um erro de fácil detecção, e a aplicação gerada pode funcionar durante algum tempo.

Em razão de dificuldades encontradas no processo de instanciação de frameworks, torna-se necessário um mecanismo para verificação das regras de instanciação. Neste artigo foi apresentada uma iniciativa que utiliza o AspectJ para auxiliar a verificação de regras de instanciação de frameworks. Como essa verificação pode ser considerada um interesse transversal à aplicação, foi implementada em um módulo, denominado *InstantiationChecker*, que contém aspectos concretos e abstratos.

O projeto do *Instantiation Checker* permite sua utilização em diferentes contextos e provê facilidades para a sua evolução e adaptação pois seus *pointcuts* podem ser concretizados nos aspectos especializados de forma a descrever pontos específicos de verificação. Nesse sentido, utilizou-se um framework de persistência para exemplificar o uso desse módulo e foram apresentadas e tratadas duas regras específicas de instanciação.

Atualmente estão sendo implementados aspectos de verificação genéricos, buscando abordar a verificação das regras mais comuns existentes nos frameworks. Serão testados frameworks mais complexos e será executada uma comparação do *Instantiation Checker* com implementações totalmente baseadas em orientação a objetos.

Referências

[1] AspectJ Team (2003). The AspectJ Programming Guide.

[2] Camargo, V. V., Ramos, R. A., Penteado, R. A. D., and Masiero, P. C. (2003). Projeto Orientado a Aspectos do Padrão Camada de Persistência. In *Proceedings of Brazilian Symposium of Software Engineering (SBES)*, Manaus-Amazonas.

[3] Diotalevi, F. (2004). Contract enforcement with AOP: Apply Design by Contract to Java software development with AspectJ.

[4] Eichberg, M., Mezini, M., Schäfer, T., Beringer, C., and Hamel, K. M. (2004). Enforcing System-Wide Properties. In *Proceedings of Australian Software Engineering Conference 2004 (ASWEC'04)*, pages 158–167, Melbourne – Australia. IEEE Computer Society Press.

[5] Fayad, M. E., Schimidt, D. C., and Johnson, R. (1999). *Building application frameworks: Object-oriented foundations of framework design*. John Wiley & Sons.

[6] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[7] Gradecki, J. D. and Lesiecki, N. (2003). *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley Publishing, Indianapolis – Indiana, 1 edition.

[8] Hanenberg, S., Unland, R., and Schmidmeier, A. (2003). Aspectj idioms for aspect-oriented software construction. In *Proceedings of 8th European Conference on Pattern Languages of Programs (EuroPLoP)*, Irsee – Germany.

[9] Isberg, W. (2002). Get Test-Inoculated! Software Development Article.

[10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An Overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355.

[11] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. In Aksit, M. and Matsuoka, S., editors, *Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York.

[12] Massol, V., Cheng, S., Bernard, J., Donagh, S., and Joshi, J. (2002). PatternTesting.

[13] Norderberg, M. E. (2001). Aspect-oriented dependency inversion. In *Proceedings of Workshop on Advanced Separation of Concerns in Object Oriented Systems (OOPSLA)*, Tampa – Florida.