4
O Método de Avaliação

O DSOA tem se tornado um importante tópico de pesquisa nos últimos anos, entretanto, pouca atenção tem sido dada à avaliação de software orientada a aspectos nas fases de projeto e implementação. Geralmente são encontrados na literatura apenas alguns estudos [28] [30] [36] que avaliam a qualidade de implementações AspectJ com foco em características relacionadas à separação de interesses. A principal razão para este problema é o fato de ser difícil avaliar múltiplos fatores sem uma abordagem sistemática de análise. Como resultado, engenheiros de software têm assumido que a propriedade que mais impacta a qualidade no DSOA é a separação de interesses. Por outro lado, como visto no Capítulo 3, alcançar elevada qualidade em software não é trivial e depende, além da separação de interesses, de outros importantes atributos da Engenharia de Software como coesão, acoplamento e tamanho.

Neste capítulo apresentamos um método quantitativo de avaliação de software orientado a aspectos que cobre as fases de projeto detalhado e implementação. Este método está apoiado sobre duas hipóteses principais:

a) avaliação de atributos relevantes é um pré-requisito para se alcançar um software de alta qualidade; e
b) avaliação abrangendo diversos fatores é essencial para que engenheiros de software façam um julgamento mais justo de diferentes soluções alternativas.

O método proposto é progressivo e estruturado em passos bem definidos, como apresentado na Figura 11. Este método é dito progressivo porque a cada iteração é obtido um novo conjunto de artefatos que compõe a solução, permitindo uma melhoria contínua até que se alcance uma solução de qualidade satisfatória. Cada iteração do método é composta de quatro passos, ou atividades, chamados medição, aplicação de regras heurísticas, análise de possíveis problemas e refatoração. Estas atividades devem ser sequencialmente seguidas e se dividem em processo de avaliação e processo de melhoria, como ilustrado na Figura 11.

Além das atividades, o método é suplementado por três conjuntos de recursos orientados a aspectos. São eles: métricas, regras heurísticas e refatorações. Estes recursos e os artefatos de entrada e saída são apresentados na Seção 4.1 e as atividades na Seção 4.2.

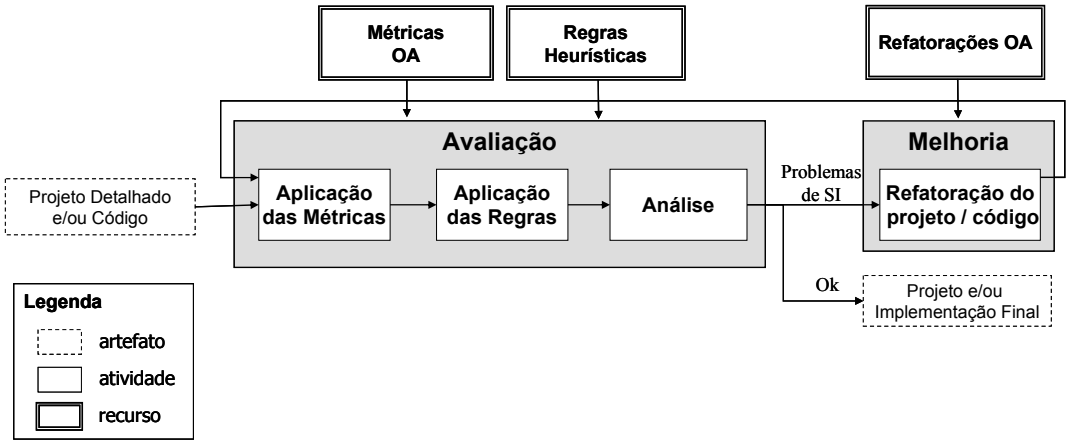

Figura 11 – Método progressivo para avaliação de software

Como mencionado anteriormente, o método proposto permite a avaliação de artefatos de software em duas fases do processo de desenvolvimento: projeto detalhado e implementação. Na fase de projeto detalhado, as informações disponíveis são mais abstratas e, portanto, as métricas, regras heurísticas e refatorações são mais genéricas. Mesmo neste nível, as quatro atividades do método devem ser seguidas. É importante ressaltar que a utilização do método na fase de projeto é independente da linguagem de programação. Mesmo sendo aplicado na fase de projeto detalhado, o método deve ser re-aplicado na fase de implementação, uma vez que os artefatos são refinados nesta fase e alguns atributos de qualidade podem ter sido violados. Na fase de implementação, os recursos do método são dependentes de características da linguagem de programação utilizada. No caso desta dissertação, as métricas e heurísticas para a fase de implementação são definidas em função das características de Java [17] e AspectJ [45].

O método emergiu de estudos empíricos [8] [27] [44] realizados no Laboratório de Engenharia de Software (LES) desta instituição de ensino. Estes estudos foram comumente utilizados na avaliação da qualidade de software orientado a aspetos e na comparação entre versões OO e OA de um mesmo sistema. Após a definição da primeira versão do método [21], outros estudos

foram importantes para a sua evolução. Alguns destes estudos experimentais foram realizados no decorrer deste curso de mestrado e estes são abordados no Capítulo 7. Como mencionado no Capitulo 1, alguns dos estudos contaram com a colaboração de outros grupos de pesquisa.

4.1. Artefatos e Recursos do Método

Dependendo da fase em que o método é aplicado (projeto detalhado ou implementação), os artefatos de entrada e saída podem variar. No nível de projeto detalhado os artefatos avaliados são diagramas baseados em UML [5] para representação do sistema. Estes diagramas devem incluir tanto diagramas estruturais (e.g. diagrama de classes e componentes [5]) como comportamentais (e.g. diagramas de seqüência e colaboração [5]). Além disso, é importante uma representação dos elementos que constituem cada interesse para que se possam calcular as métricas de separação de interesses, ou seja, a representação dos diagramas deve suportar a atividade de sombreamento como aparece na Figura 6 (Seção 3.2). Para utilização do método no nível de implementação, pode ser usado como artefato de entrada o próprio código que implementa o sistema, além de uma representação dos interesses neste artefato. Tanto no nível de projeto como no nível de implementação, os artefatos de saída do método são os artefatos de entrada que podem, ou não, ter sido reestruturados durante o processo.

Além dos artefatos de entrada e saída, o método proposto depende de três tipos de recursos orientados a aspectos. No processo de avaliação, são necessários conjuntos de métricas e de regras heurísticas. As métricas introduzem informações quantitativas enquanto as regras complementam com algum raciocínio qualitativo. Para o processo de melhoria é preciso que sejam definidas refatorações orientadas a aspectos para solucionar os problemas detectados na fase anterior. Estes três recursos são discutidos nas subseções a seguir. Na Subseção 4.1.1 são definidas três novas métricas orientadas a aspectos utilizadas em nossos estudos. As regras heurísticas são brevemente motivadas na Subseção 4.1.2, pois uma discussão mais detalhada é postergada para o Capítulo 5. Na Subseção 4.1.3 é motivado o uso de refatorações orientadas a aspectos.

4.1.1. Métricas Orientadas a Aspectos

O método de avaliação depende de um conjunto de métricas orientadas a aspectos para separação de interesses, coesão, acoplamento e tamanho. A maioria das métricas utilizadas em nossos estudos experimentais (Capítulo 7) encontra-se definida na literatura e aparece na Seção 3.2. Entretanto, três novas métricas são propostas neste trabalho para complementarem as métricas utilizadas. Estas três novas métricas apresentadas neste trabalho foram elaboradas a partir de estudos experimentais e têm como objetivo fornecer mais informações ao método de avaliação. Uma vez que várias métricas têm granularidade fina, um grande volume de informação é fornecido para as regras heurísticas (Capítulo 5). Para cada uma das três métricas desta subseção é apresentada uma definição, a relevância de sua utilização e um exemplo de como esta métrica é aplicada.

Número de Atributos do Interesse (NOAconcern)

Definição: Dado um interesse I, um componente C e um conjunto A de n atributos definidos em C, $A = \{ a_1, a_2, ..., a_n \}$. O número de atributos do interesse C, $NOAconcern(C)$, é definido como a cardinalidade do conjunto B, $|B|$, tal que $B \subseteq A$, $a_i \in B$ se e somente se a_i implementa I. Ou seja, NOAconcern conta o número de atributos de um componente cujo propósito principal é a implementação do interesse avaliado.

Relevância: Esta métrica mede o grau de espalhamento de um interesse pelos atributos de um componente. Além disso, ela mede o quanto os atributos deste componente são destinados à implementação do interesse avaliado. Uma intuição por trás disso é que quanto menos atributos são destinados ao interesse, menor é a dedicação do componente àquele interesse e, portanto, este interesse provavelmente deve ser modularizado em outro componente.

Exemplo: A Figura 12 mostra o diagrama de classes de uma implementação OO do padrão *Mediator* [26] que é parte de um dos estudos experimentais descritos no Capítulo 7. A área destacada da figura mostra os elementos que contribuem para a implementação do papel *Colleague* deste padrão. O papel *Colleague* é o interesse que se deseja avaliar neste sistema. O propósito principal da interface `GUIColleague` é contribuir para implementação deste interesse e a

classe `Button` que implementa `GUIColleague` também contribui parcialmente. A classe `Button` possui o atributo `mediator` que é parte do papel *Colleague*, portanto, na contagem da métrica NOAconcern para o interesse destacado no diagrama, a classe `Button` recebe valor 1 (um). As demais classes do sistema recebem valor 0 (zero), pois elas não possuem nenhum atributo destinado à implementação do papel *Colleague* do padrão *Mediator*.

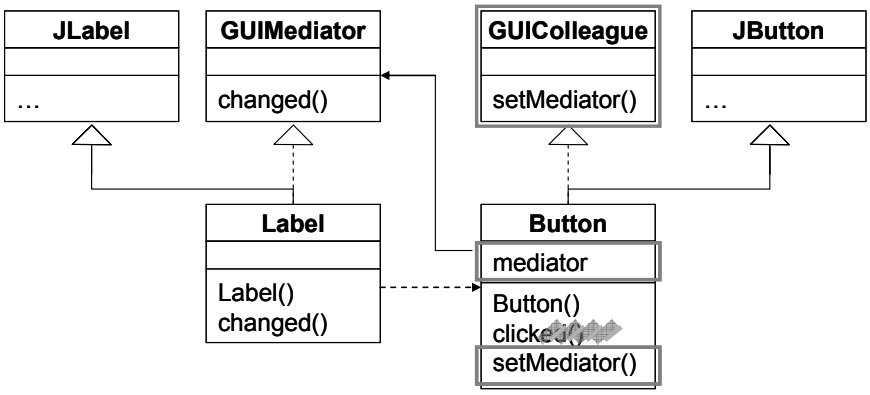

Figura 12 – Diagrama de classes do padrão *Mediator*

Número de Operações do Interesse (NOOconcern)

Definição: Dado um interesse *I*, um componente *C* e um conjunto *O* de *n* operações definidas em *C*, $O = \{ o_1, o_2, ..., o_n \}$, onde cada operação pode ser um método, um construtor ou um adendo. O número de operações do interesse *C*, *NOOconcern(C)*, é definido como a cardinalidade do conjunto *D*, $|D|$, tal que $D \subseteq O$, $o_i \in D$ se e somente se o_i implementa *I*. Ou seja, NOOconcern conta o número de operações de um componente cujo propósito principal é implementar o interesse avaliado.

Relevância: NOOconcern mede o grau de espalhamento de um interesse pelos métodos, construtores e adendos de um componente. Além disso, esta métrica mede o quanto deste interesse é implementado nas operações do componente. Assim como na métrica NOAconcern, a intuição é que quanto menos elementos do componente são destinados ao interesse, menor é a dedicação do componente àquele interesse. Ou seja, se uma classe possui poucas operações cujo propósito é implementar um determinado interesse, provavelmente, este interesse possa ser separado em outro componente, como um aspecto.

Exemplo: Observando o diagrama de classes da Figura 12 notamos que algumas operações são total, ou parcialmente, dedicadas à implementação do papel *Colleague* do padrão *Mediator* [26]. Os elementos que implementam parcialmente este interesse encontram-se rabiscados (). O método `clicked()` da classe `Button` é um exemplo de operação que não é totalmente dedicado à implementação do papel *Colleague*, o código desta classe pode ser verificado na Figura 13. Na contagem da métrica NOOconcern, são contadas apenas operações que têm como propósito implementar exclusivamente o interesse, portanto, o método `clicked()` não é considerado. O resultado desta métrica é 1 (um) para os componentes `Button` e `GUIColleague`, pois estes possuem o método `setMediator()` que implementa exclusivamente o interesse avaliado. Os outros componentes do sistema não possuem elementos dedicados ao interesse e recebem valor 0 (zero) para esta métrica.

Número de Linhas de Código do Interesse (LOCconcern)

Definição: Dado um interesse *I*, um componente *C* e um conjunto *L* que inclui as *n* linhas de código definidas em *C*, $L = \{ l_1, l_2, ..., l_n \}$. O número de linhas de código do interesse *C*, *LOCconcern(C)*, é definido como a cardinalidade do conjunto *M*, $|M|$, tal que $M \subseteq L$, $l_i \in M$ se e somente se l_i implementa *I*. Ou seja, LOCconcern conta o número de linhas de código de um componente cujo propósito principal é implementar o interesse avaliado.

Relevância: As métricas NOAconcern e NOOconcern medem o grau de espalhamento de um interesse pelos elementos do componente já no nível de projeto. Entretanto, só no nível de implementação que se pode verificar o espalhamento do interesse pelas linhas de código utilizando a métrica LOCconcern. Neste nível, as três métricas são complementares e, em conjunto, informam melhor o quanto de um componente é dedicado ao interesse avaliado. Se todos os elementos (atributos, operações e linhas de código) de um componente implementam o mesmo interesse, pode-se dizer que este componente é modular. Por outro lado, se poucos elementos do componente são destinados ao interesse avaliado, é provável que este interesse seja mais bem separado utilizando técnicas de DSOA.

Exemplo: A Figura 13 mostra o código da classe `Button`, em que a parte sombreada destaca as linhas de código necessárias para implementar o papel *Colleague* do padrão *Mediator* [26]. O número de linhas de código deste interesse (LOCconcern) na classe `Button` é 6 (seis). É importante destacar que esta métrica não conta comentário nem linhas em branco.

```
public class Button extends JButton
                  implements GUIColleague {
   private GUIMediator mediator;

   public Button(String name) {
      super(name);
      this.setActionCommand(name);
      this.addActionListener(new ActionListener() {
         public void actionPerformed(ActionEvent e) {
            clicked();
         }
      });
   }

   public void clicked() {
      mediator.changed(this);
   }

   public void setMediator(GUIMediator mediator) {
      this.mediator = mediator;
   }
}
```

Figura 13 – Sombreamento da classe `Button` do padrão *Mediator*

4.1.2. Regras Heurísticas de Avaliação

Além de um conjunto de métricas, o método de avaliação proposto é baseado em regras heurísticas que são aplicadas sobre o resultado das medições. O objetivo destas regras é apontar potenciais problemas no projeto ou implementação relacionados aos interesses transversais e que não são trivialmente detectáveis. Estes problemas incluem:

a) Interesses que não são facilmente identificados como interesses transversais.
b) Partes de um interesse transversal que não foi modularizado em um projeto OA.

c) Interesses espalhados e entrelaçados pela própria estrutura dos aspectos, como replicação de métodos na hierarquia ou repetição do uso de declaração intertipo.
d) Aspectos com problemas de coesão, acoplamento e/ou concisão devido a um critério de decomposição OA equivocado.

A aplicação das regras adverte o desenvolvedor de possíveis problemas causados por interesses transversais. Entretanto, isso não garante que os problemas existam realmente ou que o software precise ser mudado. Caso as regras levantem alguma advertência, o desenvolvedor deve analisar os artefatos do sistema para encontrar a razão desta advertência. Estas regras avaliam, de forma associada a SI, questões relacionadas a outros atributos de software. Ou seja, elas indicam quando um interesse transversal está afetando negativamente outras características de qualidade do software. Na realidade, a grande contribuição das regras é que elas carregam informações que ajudam o desenvolvedor a se concentrar em certas partes do sistema que são possivelmente problemáticas. O conjunto de regras heurísticas que compõe o método é apresentado no Capítulo 5.

4.1.3. Refatorações Orientadas a Aspectos

Refatorações são alterações feitas a artefatos de software que não alteram o comportamento observável do sistema [25] [52], ou seja, o programa deve produzir as mesmas saídas para um mesmo conjunto de entradas, antes e após a refatoração. O uso deste mecanismo tem como propósito melhorar a estrutura do sistema de tal forma a fazê-lo mais reutilizável, manutenível e fácil de ser entendido. A forma mais comum de se identificar oportunidades de refatoração são os *bad smells* [25]. *Bad smells* são sintomas que sugerem problemas no software e que podem ser removidos pelo uso de refatoração. Entretanto, estes sintomas não fornecem critérios precisos de quando uma refatoração é necessária e o desenvolvedor deve avaliar quando existe necessidade ou não de mudanças no software.

Refatorações têm sido propostas e são frequentemente utilizadas em sistemas orientados a objetos [25]. No contexto de DSOA, também existe a necessidade de refatorações que permitam a manipulação de artefatos na presença de aspectos. As refatorações orientadas a aspectos devem tratar dois novos tipos

de problemas provenientes deste paradigma: (i) extração de interesses transversais para aspectos e (ii) reestruturação interna dos aspectos. Felizmente, muitas refatorações também têm sido propostas para possibilitar a remoção de *bad smells* em sistemas orientados a aspectos [35] [38] [52]. O método proposto nesta dissertação utiliza refatorações orientadas a aspetos para retirada de problemas (ou *bad smells*) apontados pela regras heurísticas. Entretanto, a definição de um conjunto de refatorações vai além do escopo deste trabalho. Em nossos estudos experimentais (Capítulo 7), quando necessárias, as refatorações para evolução do software foram feitas manualmente pelos próprios desenvolvedores da aplicação ou por uma equipe experiente no domínio da aplicação.

4.2. Atividades do Método

A partir da identificação dos interesses, o método apresentado propõe uma seqüência de atividades para avaliar o quão bem modularizados estão estes interesses no sistema. As atividades do método se classificam em duas categorias de processos: avaliação e melhoria. No processo de avaliação as atividades que devem ser seguidas são: medição, aplicação das regras heurísticas e análise. Caso seja encontrado algum problema, deve ser feita a atividade de refatoração no processo de melhoria do software.

4.2.1. Atividade de Medição

A primeira atividade do método é a aplicação de métricas nos artefatos de entrada. Esta atividade fornece informação sobre os atributos internos do software, tais como separação de interesse, acoplamento, coesão e tamanho. O principal objetivo do método de avaliação é ajudar o projetista a determinar se o uso de aspectos pode contribuir para a melhoraria da separação de interesses. Sendo assim, são aplicadas métricas de separação de interesses para se obter informações de quais interesses podem ser classificados como transversais. Antes da aplicação destas métricas, é necessário que os interesses estejam identificados e anotados nos artefatos do projeto detalhado ou código. Além das métricas de separação de interesses, são aplicadas medições auxiliares que fornecem

características mais gerais do sistema. A Tabela 1 mostra o conjunto atual de métricas usadas em nossos estudos experimentais.

Tabela 1 – Conjunto de métricas orientadas a aspectos do método

Atributos	Métricas	Ref.	Definições
Separação de Interesses	Difusão de Interesse em Componente (CDC)	[31] [57]	Conta o número de classes, interfaces e aspectos que contribuem para implementar o interesse avaliado.
	Difusão de Interesse em Linhas de Código (CDLOC)	[31] [57]	Conta o número de pontos em que existe uma transição entre o interesse avaliado e os outros interesses do sistema.
	Número de Atributos do Interesse (NOAconcern)	Seção 4.1.1	Para cada componente, conta o número de atributos que implementam o interesse avaliado.
	Número de Operações do Interesse (NOOconcern)	Seção 4.1.1	Para cada componente, conta o número de operações cujo propósito principal é implementar o interesse avaliado.
	Número de Linhas de Código do Interesse (LOCconcern)	Seção 4.1.1	Para cada componente, conta o número de linhas de código que implementam o interesse avaliado.
Tamanho	Tamanho do Vocabulário (VS)	[31] [57]	Conta o número de classes, interfaces e aspectos do sistema.
	Número de Atributos (NOA)	[31] [57]	Conta o número de atributos de cada classe, interface e aspecto.
	Número de Operações (NOO)	[22] [40]	Conta o número de operações de cada classe, interface e aspecto.
	Número de Linhas de Código (LOC)	[40] [57]	Conta o número de linhas de código de cada classe, interface e aspecto.
Acoplamento	Acoplamento entre Componentes (CBC)	[14] [57]	Conta o número de outras classes, interfaces e aspectos com os quais um componente se relaciona.
	Profundidade da Árvore de Herança (DIT)	[14] [57]	Mede o comprimento máximo da hierarquia de herança de um componente para a raiz da árvore.
	Número de Filhos (NOC)	[14] [22]	Conta o número de classes, interfaces ou aspectos que herdar diretamente do componente.
Coesão	Perda de Coesão em Operações (LCOO)	[14] [57]	Conta os pares de operações que não acessam atributos em comum, menos os pares de operações que acessam pelo menos um atributo em comum.

Como discutido no Capítulo 7, houve algumas variações no conjunto de métricas utilizadas em cada um dos cinco estudos experimentais. Estas variações ocorreram de forma a avaliar as vantagens e desvantagens das possíveis combinações de métricas. Por exemplo, no estudo *Telestrada* (Subseção 7.1.5) realizado em parceria com pesquisadores da Universidade Estadual de Campinas,

a métrica **Peso das Operações por Componentes (WOC)** foi utilizada em alternativa a **Número de Operações (NOO)**.

4.2.2. Atividade de Aplicação das Regras

Normalmente, a aplicação de métricas gera uma grande quantidade de números cuja interpretação não é trivial e consome tempo. Desta forma, terminada esta atividade de medição, regras heurísticas são aplicadas para ajudar na interpretação dos números, apontando os valores que possam indicar oportunidade de melhoria no software. A Tabela 2 apresenta um conjunto de regras heurísticas que são diretamente derivadas de algumas das métricas listadas na Tabela 1.

Tabela 2 – Conjunto de regras heurísticas baseadas em uma única métrica

	Regras	Métricas
01	**Se** um INTERESSE afeta vários componentes do sistema **então** este é um INTERESSE ESPALHADO	CDC
02	**Se** o código um INTERESSE se mistura ao código de outros interesses **então** este é um INTERESSE ENTRELAÇADO	CDLOC
03	**Se** os atributos e operações de um COMPONENTE não são fortemente relacionados **então** este é um COMPONENTE POUCO COESO	LCOO
04	**Se** um COMPONENTE depende de muitos outros componentes **então** este é um COMPONENTE ALTAMENTE ACOPLADO	CBC
05	**Se** um COMPOENTE está muito distante da raiz na hierarquia de herança **então** este é um COMPONENTE DE HERANÇA PROFUNDA	DIT

As duas primeiras regras da Tabela 2 podem ser usadas para alertar o desenvolvedor da existência de um interesse, possivelmente transversal, que se encontra espalhado e entrelaçado no código. Estas duas regras são suportadas, respectivamente, pelas métricas **Difusão de Interesse em Componente (CDC)** e **Difusão de Interesse em Linhas de Código (CDLOC)**. Esta tabela também apresenta outras três regras que são usadas para identificar componentes com baixa coesão (Regra 3) ou alto acoplamento (Regras 4 e 5). As três últimas regras são suportadas pelas métricas **Perda de Coesão em Operações (LCOO)**, **Acoplamento entre Componentes (CBC)** e **Profundidade da Árvore de Herança (DIT)**. A avaliação de outros atributos do sistema, como coesão e acoplamento, é importante porque a separação de interesses pode afetar tais

atributos [21] [27]. Ou seja, mesmo que um sistema possua boa separação de interesses, se muitos de seus componentes apresentarem baixa coesão e alto acoplamento, o projetista pode decidir fazer uma refatoração.

4.2.3. Atividade de Análise e Identificação de Problemas

As regras heurísticas propostas neste trabalho apóiam a interpretação do resultado de medições. Elas apontam possíveis problemas no projeto ou implementação do sistema e alertam o engenheiro de software. Porém, o uso destas regras não dispensa a análise feita por parte de um especialista humano. As regras indicam os pontos potencialmente problemáticos, mas os desenvolvedores têm que se certificar de que estes pontos são realmente problemas, e caso sejam, avaliar se uma refatoração de software é benéfica. Esta tarefa do método não pode ser completamente automatizada e é representada pela terceira atividade da Figura 11.

4.2.4. Atividade de Refatoração

Como definido na Subseção 4.1.3, refatorações são alterações aplicadas aos artefatos de software e muitas refatorações orientadas a aspectos têm sido propostas na literatura [25] [35] [38] [52]. A última atividade definida no método de avaliação é aplicação destas refatorações no software com o objetivo de melhorar sua estrutura. Esta atividade é utilizada para eliminação de problemas detectados no processo de avaliação, em especial, apontados pelas regras heurísticas. Para exemplificar o uso de refatorações orientadas a aspectos no processo de melhoria do software é apresentada uma implementação do padrão *Prototype* [14]. Esta implementação foi feita por Hannemann e Kiczales [36] e é parte de um dos nossos estudos experimentais.

A intenção do padrão *Prototype* é especificar a criação de objetos pela clonagem de uma instância da classe [14]. Este padrão define um único papel ou interesse, chamado *Prototype*, que define o comportamento para a clonagem de objetos. A instância do padrão, ilustrada na Figura 14, é usada para definir objetos `String` que podem ser clonados [36]. O diagrama de classes desta figura mostra a implementação OO em que as classes `StringPrototypeA` e `StringPrototypeB`

implementam a interface `Cloneable` e definem o método `clone()` para permitir a clonagem.

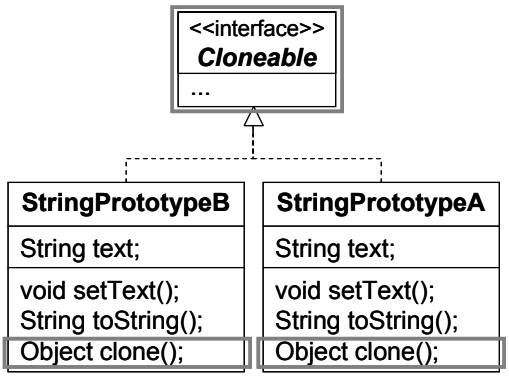

Figura 14 – Diagrama de classes OO do padrão *Prototype*

O resultado das atividades de avaliação na versão OO do padrão *Prototype* sugere a existência de um problema de separação de interesses. A Figura 14 destaca os elementos do padrão espalhados e entrelaçados pelo diagrama, o que confirma este resultado. Para solucionar o problema, desenvolvedores do sistema podem decidir usar refatorações OA e separar o interesse do padrão. Uma possível solução após a refatoração do sistema é apresentada na Figura 15. Nesta versão OA, o código que implementa o padrão *Prototype* está localizado nos dos aspectos `StringPrototypes` e `PrototypeProtocol`.

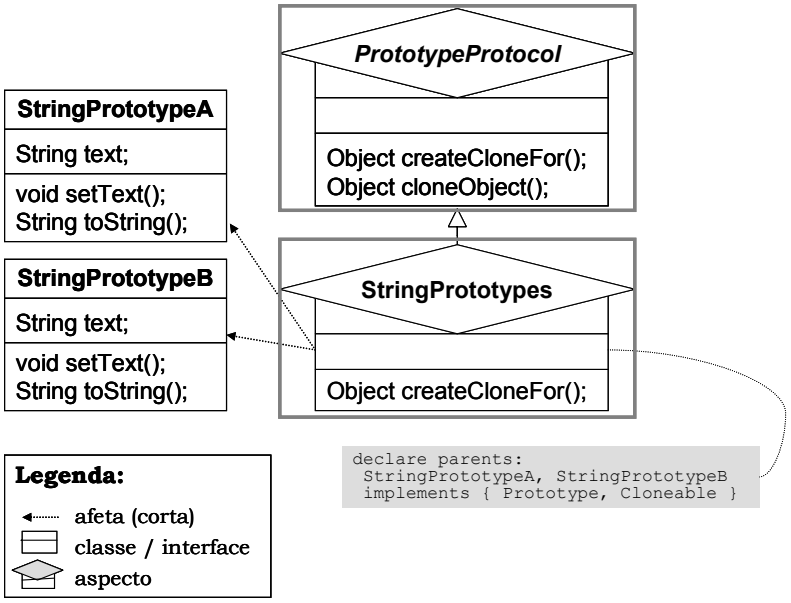

Figura 15 – Diagrama de classes OA do padrão *Prototype*

Um possível conjunto de refatorações aplicadas à versão OO (Figura 14) do padrão *Prototype* para transformá-la em uma solução OA (Figura 15) é sumarizado da seguinte forma:

1. Criar um aspecto vazio e nomeá-lo `StringPrototypes`;
2. Mover os métodos `clone()` das classes `StringPrototypeA` e `StringPrototypeB` para o aspecto `StringPrototypes` (*Move Method* [25]);
3. Criar um super-aspecto de `StringPrototypes` e nomeá-lo `PrototypeProtocol`;
4. Mover o método `clone()` do aspecto `StringPrototypes` para o aspecto pai `PrototypeProtocol` (*Pull up Method* [25]);
5. Criar uma interface `Prototype` interna ao aspecto `PrototypeProtocol` e usar *container introduction* [34] para adicionar o método `clone()` às classes `StringPrototypeA` e `StringPrototypeB`;
6. Substituir a implementação explicita da interface `Cloneable` feita pelas classes `StringPrototypeA` e `StringPrototypeB` por declaração intertipo (*Replace Implements with Declare Parents* [52]).