4
Diretrizes para o Projeto e Implementação de Frameworks usando Orientação a Aspectos

Nossa abordagem para desenvolvimento de frameworks usando aspectos oferece um conjunto de diretrizes (seção 3.3.1) para a modularização de determinados tipos de características encontradas no projeto e implementação de frameworks. Seguindo nossas diretrizes, tais características são implementadas usando aspectos e estendem o framework em pontos de junção de extensão bem definidos, denominados EJPs. Esse capítulo detalha tais diretrizes, mostrando como EJPs e aspectos de extensão podem ser implementados usando os mecanismos de AspectJ. O framework JUnit é usado para ilustrar as diretrizes de implementação.

4.1.
Implementando EJPs em AspectJ

Um EJP na nossa abordagem declara: (i) um conjunto de pontos de junção do framework que são candidatos a serem estendidos por aspectos; e (ii) um conjunto de contratos que são usados para regular as relações entre o framework, EJPs e aspectos de extensão. Nas subseções seguintes, os mecanismos específicos de AspectJ usados para codificação de EJPs e seus respectivos contratos são detalhados.

4.1.1.
Documentação Visual de EJPs e Aspectos de Extensão

Para facilitar a documentação e visualização dos elementos que compõem a abordagem de desenvolvimento de frameworks, foram definidos alguns estereótipos em diagramas de classes UML [16] que buscam facilitar a identificação e visualização do núcleo do framework, e seus respectivos EJPs e aspectos de extensão e do núcleo. Essa documentação é usada ao longo desse capítulo e na apresentação dos estudos de caso descritos no capítulo 6.

A Figura 11 apresenta a documentação do framework JUnit [73, 75, 84] com seus respectivos aspectos do núcleo, EJP e aspectos de extensão. Diferentes pacotes podem ser definidos para agregar cada um desses elementos. Na Figura 11 foram definidos os seguintes pacotes: *framework core*, *extension join points*, *variability aspects* e *integration aspects*. Aspectos EJPs são representados usando o esteréotipo `<<EJP>>`, e o compartimento de métodos é usado para definir seus diferentes pontos de corte. Aspectos do núcleo e de extensão são representados usando o esteréotipo `<<aspect>>`, a distinção entre eles pode ser feita pelo pacote do qual fazem parte. Dois novos tipos de estereótipos foram também definidos para especificar relações de dependência: (i) `<<crosscuts>>` – indica que determinados aspectos interceptam pontos de junção de classes do núcleo do framework; e (ii) `<<uses>>` – útil para especificar que um dado aspecto de extensão usa a definição de pontos de corte expostos por aspectos ou EJPs ou que utiliza a implementação de uma dada classe.

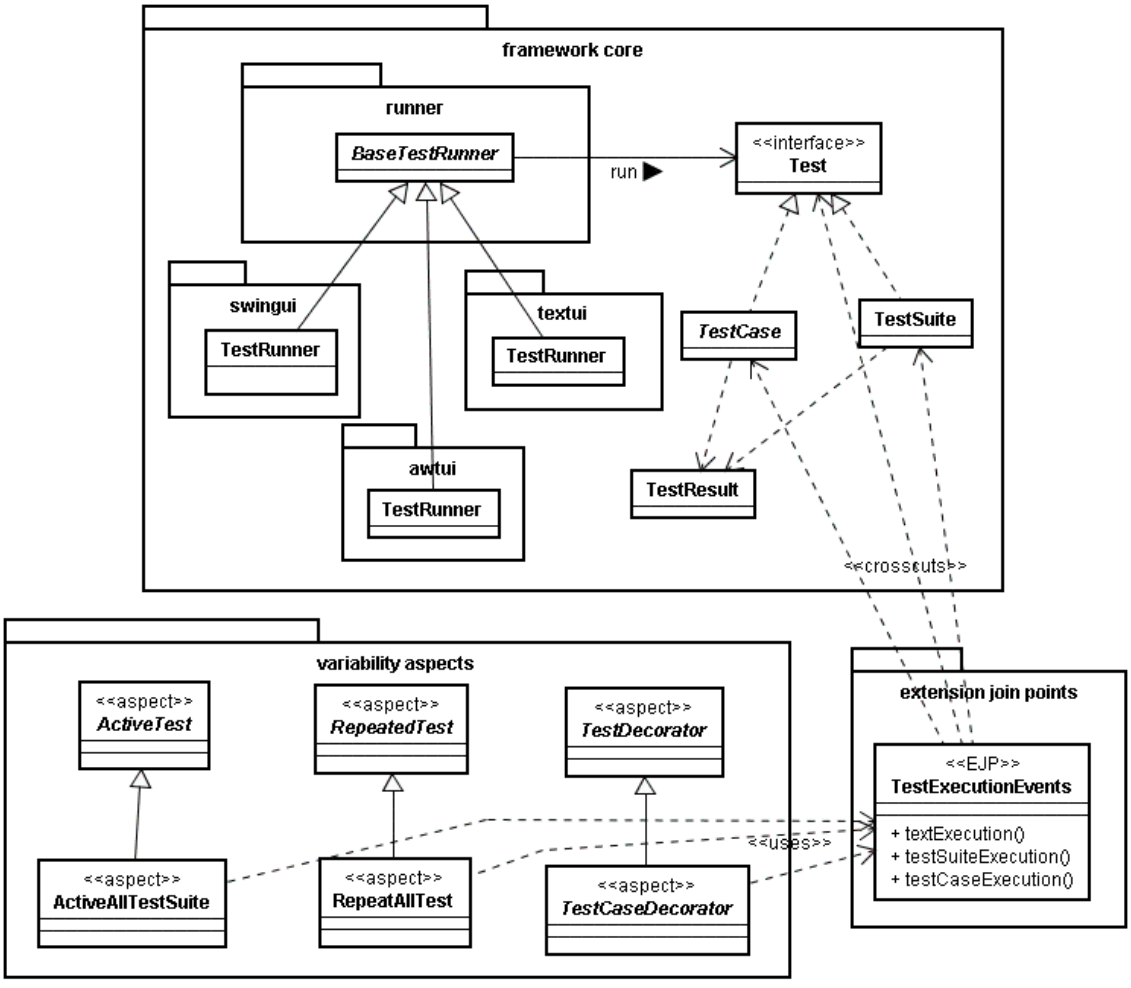

Figura 11. Diagrama de Classes e Aspectos do JUnit

4.1.2. Estrutura de EJPs

EJPs são implementados na nossa abordagem usando a abstração de aspectos de AspectJ. Tais aspectos declaram pontos de junção públicos os quais representam os EJPs do framework. Dependendo da quantidade de pontos de junção a serem expostos pelo framework, vários aspectos podem ser criados, cada um englobando um subconjunto de pontos de junção fortemente relacionados. A Tabela 1 apresenta os elementos que compõem a especificação de cada EJP em AspectJ.

Elemento	Propósito
Nome	• Especifica o nome do aspecto EJP; • É representado pelo nome do aspecto em AspectJ.
Pontos de Extensão Transversais	• Quantificam o conjunto de pontos de junção do framework; • Representam eventos ou estados de transição relevantes que ocorrem durante a execução das funcionalidades do núcleo do framework.
Escopo	• Define o escopo do núcleo do framework, contemplando todos os elementos de implementação (classes, aspectos); • É representado pela especificação de pontos de corte AspectJ, que utilizam a construção `within`, para incluir todos os pacotes dos componentes do framework.

Tabela 1. Estrutura Geral de EJPs

A Figura 12 apresenta o aspecto EJP `TestExecutionEvents`, o qual expõe um conjunto de pontos de junção no framework JUnit, através de vários pontos de corte, entre eles: (i) ponto de corte `testExecution()` – o qual expõe o evento de execução de casos ou suítes de teste (instância do tipo `Test`) (linhas 3-4); (ii) ponto de corte `testSuiteExecution()` – expõe o evento de execução de todos os suítes de teste (linhas 11-14); (iii) ponto de corte `testCaseExecution()` – expõe o evento de execução de todos os casos de teste (linhas 17-20). Esses dois últimos pontos de corte são definidos em função do ponto de corte `testExecutionGeneral()`, o qual também é usado para auxiliar na especialização de EJPs (Seção 4.1.3).

Cada aspecto EJP pode também definir o escopo das classes que fazem parte do núcleo do framework, através da especificação de pontos de corte com o uso da construção `within` de AspectJ. Tais pontos de corte são usados na especificação dos contratos dos EJPs. Na Figura 12 foi definido o ponto de corte `FWScope()` com tal propósito, ele define que o núcleo do framework JUnit compreeende todas as classes dentro do pacote `junit` e respectivos subpacotes.

```
01 public aspect TestExecutionEvents {
02   // Expose the test execution event
03   public pointcut testExecution(Test test):
04      target(test) && call (void Test.run(TestResult));
05
06   // Expose the method of Test Execution
07   public pointcut testExecutionGeneral(TestResult result):
08           call (void *.run(TestResult)) && args(result);
09
10   // Expose the test suite execution event
11   public pointcut testSuiteExecution (TestSuite testSuite,
12                                       TestResult result):
13        testExecutionGeneral (result) &&
14        target(testSuite) && target(TestSuite);
15
16   // Expose the test case execution event
17   public pointcut testCaseExecution (TestCase testCase,
18                                      TestResult result):
19        testExecutionGeneral (result) &&
20        target(testCase) && target(TestCase);
21   ...
22   // Framework Scope
23   protected pointcut FWScope(): within(junit..*);
24   ...
25 }
```

Figura 12. Aspecto EJP `TestExecutionEvents`

Uma vez exposto o conjunto de EJPs de um framework, aspectos de extensão podem ser codificados para estender a funcionalidade do seu núcleo. A Figura 13 ilustra a definição de um aspecto de variabilidade para o JUnit, que estende o framework para possibilitar a repetição da execução de casos de teste. O aspecto `RepeatedAllTests` define um adendo do tipo `around` (linhas 6-11), o qual intercepta o evento de execução de casos de teste para introduzir a funcionalidade de execução repetida. O ponto de corte `testCaseExecution()` do aspecto EJP `TestExecutionEvents` é usado para introduzir a funcionalidade do adendo dentro do framework (linha 7).

```
01 public aspect RepeatedAllTests {
02    public int getTimesRepeat(){
03      return 10;
04    }
05
06    void around(TestCase testCase, TestResult testResult):
07         TestExecutionEvents.testCaseExecution(testCase, result) {
08       for (int i=0; i < getTimesRepeat(); ++i){
09          proceed(test, testResult);
10       }
11    }
12 }
```

Figura 13. Aspecto de Extensão RepeatAllTests

4.1.3. Especialização de EJPs

Cada EJP expõe um conjunto de pontos de junção do framework nos quais podem ser inseridos novos comportamentos tranversais codificados em aspectos de extensão. Dentre os pontos de junção expostos, alguns podem ser métodos abstratos ou métodos gancho (*hooks*) de classes que representam pontos flexíveis OO (*hot-spots*) do framework. Durante a instanciação do framework, as classes que representam pontos flexíveis são especializadas com a definição de métodos concretos para criar comportamento específico para uma dada aplicação. Assim, uma vez que alguns dos métodos gancho ou abstratos podem ser especializados, também os pontos de junção definidos nos EJPs podem ser especializados para afetar apenas instâncias específicas de pontos flexíveis.

No framework JUnit, por exemplo, alguns dos seus pontos de junção podem ser customizados para afetar apenas instâncias específicas de casos e suítes de teste. Dessa forma, o desenvolvedor que está implementando os aspectos de extensão precisa especializar os EJPs para indicar que ele deseja estender apenas instâncias específicas de casos e suítes de teste.

Em AspectJ, a especialização de EJPs pode ser implementada através do uso de composições de pontos de corte e do designador `target()`. A Figura 14 apresenta o aspecto de extensão `RepeatSpecificTestCases`, o qual afeta apenas subclasses de casos de teste específicas. O novo ponto de corte `testCaseExecutionSpecialization()` é definido como uma especialização do ponto de corte `testCaseExecutionGeneral()`, presente no aspecto EJP `TestExecutionEvents`. Para afetar apenas instâncias específicas, o designador de

ponto de corte `target()` é usado. Ele permite especificar os tipos de subclasses que se deseja afetar (linha 5). No caso específico desse aspecto, ele irá afetar apenas as subclasses `TestCaseA` e `TestCaseB`.

```
01 public aspect RepeatSpecificTestCases {
02    public pointcut testCaseExecutionSpecialization(
03            Test testCase, TestResult result):
04       TestExecutionEvents.testCaseExecutionGeneral(result)
05       && (target(TestCaseA) || target(TestCaseB))
06       && target(testCase)
07       && !adviceexecution();
08
09    public int getTimesRepeat(){
10       return 10;
11    }
12    void around(TestCase testCase, TestResult testResult):
13         TestExecutionEvents.testCaseExecutionSpecialization
14                                        (testCase, result) {
15        for (int i=0; i < getTimesRepeat(); ++i){
16          proceed(testCase, testResult);
17        }
18     }
19 }
```

Figura 14. Exemplo de Especialização de EJP

4.1.4. Contratos de EJPs

Na nossa abordagem, foi também definido um conjunto de contratos [92] que podem existir entre framework, EJPs e aspectos de extensão. Tais contratos são organizados nas seguintes categorias: (i) *contratos internos do framework* – que definem restrições a serem obedecidas pelo framework; e (ii) *contratos de extensão do framework* – definem restrições a serem seguidas pelos aspectos de extensão.

Os contratos internos do framework definem restrições (*constraints*) cujo propósito é garantir que refatorações e evoluções do framework não irão afetar a funcionalidade de seus aspectos de extensão. Eles são classificados em: (i) *estruturais* – cujo objetivo é garantir que o framework implementa interfaces específicas definidas pelos EJPs; e (ii) *comportamentais* – os quais buscam garantir que os EJPs expõem todos (e apenas) os eventos ou estados desejados.

Os contratos de extensão do framework são usados para garantir que cada aspecto de extensão respeita restrições e invariantes do framework. As seguintes

categorias foram definidas: (i) estruturais – estes contratos objetivam garantir que aspectos apenas estendem os pontos de junção do framework que foram expostos pelos EJPs; (ii) comportamentais – definem os métodos de classes do framework que podem ser invocados pelos aspectos de extensão; e (iii) invariantes – definem um conjunto de pré e pós-condições que devem ser preservadas antes e depois da execução dos adendos dos aspectos de extensão.

As Tabelas 2 e 3 apresentam a categorização de contratos, assim como os respectivos mecanismos de AspectJ que podem ser usados para implementá-los. AspectJ oferece diversos mecanismos úteis para implementar cada tipo de contrato. Durante a escolha de qual mecanismo usar para cada tipo de contrato, foram priorizados mecanismos estáticos ao invés de dinâmicos, uma vez que os primeiros podem ser verificados em tempo de compilação. Esse é o caso, por exemplo, dos mecanismos de AspectJ: `declare parents`, `declare error` e `declare warning`. Alguns tipos de contrato, entretanto, dependem de informações dinâmicas para serem implementados. Para esses casos específicos, tal como a verificação de invariantes do framework, a construção `adviceexecution` de AspectJ que permite interceptar a execução de adendos de aspectos pode ser utilizada. Alguns tipos de contratos não podem ser codificados usando os mecanismos disponíveis em AspectJ, tais como, os contratos de extensão estrutural (Tabela 3).

Tipo do Contrato	Descrição e Mecanismos de Implementação em AspectJ
Estrutural	• Especificação de interfaces que devem ser implementadas pelas classes do framework; • A obrigação de implementar essas interfaces é atribuída pelos contratos estruturais dos EJPs usando a construção `declare parents` de AspectJ. As interfaces são também declaradas dentro dos aspectos que representam os contratos dos EJPs.
Comportamental	• Implementação de políticas de reforço que garantem que os EJPs são chamados em todos e apenas lugares apropriados dentro do código das classes do framework. • Esses contratos podem ser especificados usando os comandos `declare warning` e `declare error` de AspectJ.

Tabela 2. Contratos Internos do Framework.

A Figura 15 apresenta um aspecto contendo a implementação de dois contratos para o framework JUnit. O primeiro é um contrato interno

comportamental definido usando a construção `declare error` (linhas 7-11). Ele restringe que o método `run(TestResult)` da interface `Test` deve ser invocado apenas por determinados métodos das classes `TestCase` e `TestSuite`. Isso garante que apenas tais métodos podem efetivamente demandar a execução dos testes dentro do framework JUnit. O método `run()` da interface `Test` é o responsável direto pela execução dos testes no JUnit. Durante a refatoração ou evolução do framework JUnit, caso novas classes passem a invocar tal método, o contrato acusará um erro que deverá ser analisado pelo desenvolvedor para garantir que aquele contrato interno continua sendo preservado, se necessário o código do contrato pode evoluir para incluir novos métodos que fazem chamadas ao método `run()` da interface `Test`.

Tipo do Contrato	Descrição e Mecanismos de Implementação em AspectJ
Estrutural	• Esse contrato garante que aspectos de extensão apenas estendem os pontos de junção do framework expostos pelos EJPs; • Esse contrato não pode ser implementado em AspectJ, devido a limitação corrente da linguagem a qual não permite restringir pontos de junção específicos para serem estendidos; • Uma forma de garantir esses contratos é estabelecer que desenvolvedores sigam a prática de programação de estender o framework apenas nos pontos de junção especificados nas EJPs.
Comportamental	• Determina quais métodos de classes do framework podem ser invocados pelos aspectos de extensão; • Tais contratos podem ser especificados em AspectJ, usando os comandos `declare warning` e `declare error` de AspectJ, o qual permite a verificação estática das políticas de acesso.
Invariantes	• Definem pré e pós-condições que devem ser garantidas antes e depois da execução dos adendos; • O comando `adviceexecution()` o qual permite interceptar os adendos de aspectos pode ser usado com a finalidade de implementar tais pré e pós condições nos aspectos de extensão.

Tabela 3. Contratos de Extensão do Framework.

A Figura 15 também apresenta um contrato de extensão comportamental que visa garantir que apenas os aspectos de extensão do JUnit podem invocar métodos internos das classes do framework JUnit (linhas 14-22). Esse contrato foi

definido usando dois mecanismos sofisticados de AspectJ, são eles: (i) `adviceexecution` - que permite interceptar a invocação de adendos de aspectos; e (ii) `cflow` - que permite interceptar todos pontos de junção que ocorrem no contexto do fluxo de controle de um dado ponto de junção. Assim, o adendo `before` de tal contrato lança uma `RuntimeException`, sempre que houver chamadas para classes do framework, ocorrendo dentro de adendos de aspectos (`adviceexecution`) ou originadas a partir de tais adendos (`cflow`) e que não estejam dentro do pacote de aspectos de extensão (`!extensionAspectsScope()`).

```
01 public aspect JUnitEJPContracts {
02   //Behavioral Internal Contract
03   public pointcut EJPMethodsCallScope():
04     withincode (public TestResult TestCase.run())  ||
05     withincode (public void TestSuite.runTest(
06                                   Test, TestResult));
07   declare error:
08      (!EJPMethodsCallScope()
09        && call(void Test.run(TestResult))):
10      "Contract violation: Test execution should occur" +
11      "through one of the methods specified above";
12
13   //Behavioral Extension Contract
14   public pointcut extensionAspectsScope():
15             within(junit.aop.extensions..*);
16   before() :
17     cflow ( adviceexecution() && !extensionAspectsScope())
18     && ( call(* *(..)) && TestExecutionEvents.FWScope() ){
19      throw new RuntimeException("Contract Violation: no " +
20        "aspects, except variability aspects, can access " +
21        "the elements of JUnit framework.");
22    }
23 }
```

Figura 15. Exemplo de Contratos de EJPs no Contexto do JUnit

Esta tese apenas propõe uma categorização de contratos de EJPs. Tais contratos foram utilizados na implementação de dois de nossos estudos de caso [29, 75]. Uma abordagem para verificação e teste de características transversais

[30], a qual detalha a utilização de tais contratos em conjunção com a definição de testes automatizados de unidade e integração vem sendo desenvolvida em um outro trabalho de pesquisa.

4.2. Implementando Aspectos de Extensão em AspectJ

Na nossa abordagem, todas as extensões tranversais do framework são introduzidas no seu núcleo usando aspectos de variabilidade e integração. Cada aspecto introduz comportamentos transversais no conjunto de pontos de junção expostos pelos EJPs. Os aspectos desempenham um papel específico que está diretamente relacionado com a funcionalidade transversal que eles introduzem no framework. Eles podem, por exemplo, desempenhar o papel de observadores de eventos internos do framework para notificar módulos externos interessados em tais eventos. Eles podem também mediar a comunicação entre classes específicas do framework e outras extensões OO dentro de particulares pontos de junção expostos pelos EJPs. Ou eles podem ainda decorar EJPs com implementações de características transversais opcionais ao núcleo do framework. Assim, muitos aspectos desempenham o papel de padrões de projeto tradicionais (tais como, *Observer*, *Mediator* ou *Decorator*) [45] com o objetivo de estender o núcleo do framework.

No estudo de caso do framework JUnit, por exemplo, foram definidos vários aspectos de variabilidade, os quais "decoram" o comportamento oferecido por classes do núcleo do framework, são eles: (i) `RepeatedTests` – aspecto abstrato que é concretizado para permitir a execução de casos de teste repetidamente; (ii) `ActiveTest` – aspecto abstrato que é especializado para permitir a execução de suítes de teste em *threads* distintas; e (iii) `TestCaseDecorator` – aspecto abstrato que é usado para introduzir comportamentos adicionais antes ou depois de casos e suítes de teste. Esses aspectos atuam como decoradores de suítes e casos de teste.

As Figuras 16 e 17 apresentam, respectivamente, o código dos aspectos de extensão `ActiveTest` e `ActiveAllTestSuite`. O aspecto abstrato `ActiveTest` define o comportamento comum de execução de casos ou suítes de teste em threads distintas, através de um conjunto de adendos e pontos de corte abstratos. O aspecto `ActiveAllTestSuite` especializa o aspecto `ActiveTest`, concretizando

os pontos de junção do sistema onde se deseja aplicar tal funcionalidade. Observe que na definição dos pontos de corte do aspecto `ActiveAllTestSuite`, pontos de corte expostos pelo aspecto EJP `TestExecutionEvents` são reusados.

```
01 public abstract aspect ActiveTest {
02   private volatile int fActiveTestDeathCount;
03   private int testCaseQuantity;
05   public abstract pointcut activeTestsPoints(
06              TestSuite testSuite, TestResult testResult);
08   before (TestSuite testSuite, TestResult result):
09                activeTestsPoints(testSuite, result){
10      fActiveTestDeathCount= 0;
11   }
12   after (TestSuite testSuite, TestResult result):
13                activeTestsPoints(testSuite, result){
14      waitUntilFinished();
15   }
16   public abstract pointcut activeInternalTestsPoints(
17     TestSuite testSuite, Test test, TestResult testResult);
19    void around(TestSuite testSuite,
20              Test test, TestResult result):
21      activeInternalTestsPoints(testSuite, test, result){
22        testCaseQuantity = testSuite.countTestCases();
23        runTest(test, result);
24    }
25    public void runTest(final Test test,
26                        final TestResult result){
27       Thread t= new Thread() {
28          public void run() {
29            try {  test.run(result);
30            } finally {
31              ActiveTest.this.runFinished(test);
32            }
33          }
34        };
35        t.start();
36    }
37    synchronized void waitUntilFinished() {  ... }
38 }
```

Figura 16. Aspecto de Variabilidade `ActiveTest`

```
01 public aspect ActiveAllTestSuite extends ActiveTest {
02    public pointcut activeTestsPoints(TestSuite testSuite,
03                                      TestResult result):
04      TestExecutionEvents.testSuiteExecution(testSuite,
05                                             result);
06
07    public pointcut activeInternalTestsPoints(
08        TestSuite testSuite, Test test, TestResult result):
09      TestExecutionEvents.testMethodExecutionInsideSuite(
10                            testSuite, test, result);
11
12 }
```

Figura 17. Aspecto de Variabilidade `ActiveAllTestSuite`

4.2.1. Variabilidades em Aspectos de Extensão

A implementação dos aspectos de extensão pode também requerer a separação entre código comum e variável presente nos mesmos. Isso significa que aspectos de extensão podem também conter variabilidades. Os aspectos `ActiveTest` e `ActiveAllTestSuite`, apresentados na seção anterior, representam a separação entre código comum e variável para a característica de execução de suíte de teste em *threads* distintas.

O aspecto `RepeatedAllTests` (apresentado na Seção 4.1.2) também pode ser modularizado de tal forma a postergar para tempo de instanciação da aplicação, a definição da quantidade de vezes de repetição de suítes ou casos de teste e a quais casos de teste específicos se deseja aplicar tal funcionalidade de extensão. Diferentes tipos de variabilidades podem existir nos aspectos de extensão, tais como:

(i) **comportamento variável do aspecto** – nesse caso parte da funcionalidade do aspecto depende de informação disponível na aplicação que será gerada a partir da instanciação do framework. Os mecanismos de definição de aspectos e métodos abstratos oferecidos por AspectJ permitem a implementação desse comportamento variável;

(ii) **ponto de junção varíavel do aspecto** – esse tipo de variabilidade ocorre quando o comportamento geral do aspecto já está definido, mas os pontos de junção onde ele irá atuar só são conhecidos em tempo de instanciação do framework. Em AspectJ podem ser usados os mecanismos de aspectos e pontos de corte abstratos para implementar tal tipo de variabilidade;

(iii) **introdução de um conjunto de atributos e métodos em interfaces e classes** – um conjunto de atributos e métodos pode ser introduzido numa interface (usando os mecanismos de declarações inter-tipos de AspectJ), e posteriormente, serem introduzidos em uma dada classe do sistema (através da construção `declare parents` de AspectJ).

Esses mecanismos podem ser combinados quando um dado aspecto de extensão precisa implementar mais do que um tipo de variabilidade. Hannenberg et al [57] apresentam um conjunto de idiomas AspectJ que pode ser utilizado para

implementar e compor esses diferentes mecanismos. Idiomas [19] são padrões de implementação específicos de uma dada linguagem de programação. Os idiomas AspectJ propostos endereçam o projeto e implementação de aspectos que podem ser reutilizados em diferentes aplicações.

A Figura 18 mostra a refatoração do aspecto `RepeatedAllTests` (Seção 4.1.2) em um novo aspecto abstrato, denominado `RepeatedTests`. Ele define dois elementos abstratos: (i) o ponto de corte `repeatedTestsPoints()` – que deve ser especializado para indicar os casos de teste que terão sua execução repetida (linhas 2-3); e (ii) o método `getTimesRepeat()` – que retorna a quantidade de vezes que determinados casos de teste serão repetidos (linha 4). A Figura 19 apresenta dois subaspectos de `RepeatedTests`. O subaspecto `RepeatAllTests` concretiza o ponto de corte `repeatedTestsPoints()` determinando a execução repetida de todos os testes, através do reuso do ponto de corte específico exposto pelo aspecto EJP `TestExecutionEvents`. O subaspecto `RepeatSpecificTestCases` especializa o ponto de corte `testCaseExecutionGeneral()` do aspecto EJP `TestExecutionEvents` para introduzir a funcionalidade de repetição apenas nos casos de teste `TestCaseA` e `TestCaseB`.

Durante a descrição dos estudos de caso (Capítulo 6), serão apresentados diferentes exemplos de ocorrência de variabilidades em aspectos de extensão.


```
01 public abstract aspect RepeatedTests {
02    public abstract pointcut repeatedTestsPoints(
03               Test test, TestResult testResult);
04    public abstract int getTimesRepeat();
05
06    void around(Test test, TestResult testResult):
07            repeatedTestsPoints(test, testResult) {
08       for (int i=0; i < getTimesRepeat(); ++i){
09         proceed(test, testResult);
10       }
11    }
12 }
```

Figura 18. Aspecto de Variabilidade `RepeatedTests`

```
01 public aspect RepeatAllTests extends RepeatedTests {
02   public pointcut repeatedTestsPoints(
03        Test testCase, TestResult result):
04    TestExecutionEvents.testCaseExecution(testCase, result);
05
06    public int getTimesRepeat(){  return 10; }
07 }

01 public aspect RepeatSpecificTestCases extends RepeatedTests {
03    public pointcut repeatedTestsPoints(
04       Test testCase, TestResult result):
05            testCaseExecutionGeneral(result)
06            (target(TestCaseA) || target(TestCaseB));
07            && target(testCase)
08            && !adviceexecution();
09    public int getTimesRepeat(){  return 10; }
10 }
```

Figura 19. Exemplos de Especialização de Aspectos de Extensão

4.3. Sumário

A codificação dos elementos (EJPs e aspectos de extensão) de nossa abordagem em AspectJ traz sistematização para a sua adoção, oferecendo um conjunto concreto de diretrizes de implementação. Os seguintes benefícios podem ser observados como fruto dessas diretrizes: (i) habilita o desenvolvedor a expor um conjunto de pontos de junção que estão espalhados no framework em um conjunto restrito de aspectos, que por sua vez podem ser usados para estender a funcionalidade do framework, através da codificação de aspectos de variabilidade e integração; (ii) permite a especificação de vários contratos a serem satisfeitos quando estendendo pontos de junção do framework. A especificação de tais contratos permite que os mesmos sejam não apenas declarados, mas também garantidos em tempo de compilação e execução; (iii) permite a especificação de variabilidades existentes nos EJPs, através do uso do mecanismo de composição e especialização de pontos de junção; e (iv) permite implementar variabilidades encontradas nos aspectos de extensão, através do uso de herança entre aspectos, métodos e pontos de corte abstratos, e declarações intertipo.