2
Abordagens de Desenvolvimento de Família de Sistemas

Este capítulo apresenta uma visão geral das três diferentes abordagens de desenvolvimento de família de sistemas que são exploradas nesse trabalho. A tecnologia de Frameworks Orientados a Objetos (OO), uma das mais populares para a implementação de arquiteturas de famílias de sistemas, é inicialmente apresentada (Seção 2.1). Problemas de modularização de determinados tipos de características encontrados durante o desenvolvimento de frameworks OO são também descritos. Em seguida, a abordagem de Desenvolvimento de Software Orientado a Aspectos (DSOA), cujo objetivo é a modularização de interesses e características transversais, é apresentada (Seção 2.2). Finalmente, a abordagem de Desenvolvimento Generativo (DG) é descrita (Seção 2.3). DG endereça o estudo de métodos e ferramentas que habilitam a produção automática de membros de uma família de software a partir de especificações de alto nível. O capítulo é concluído com uma análise do potencial de integração entre as abordagens (Seção 2.4).

2.1.
Frameworks Orientados a Objetos

Frameworks Orientados a Objetos (OO) [37, 64] representam uma técnica comum e pragmática para a implementação de arquiteturas de família de sistemas e linhas de produto. Cada framework OO define uma arquitetura flexível na forma de código-fonte em uma linguagem de programação, que pode ser customizada para a implementação de diferentes aplicações para um mesmo domínio. Um framework OO é composto por um conjunto de classes que juntas colaboram para implementar um projeto abstrato para um domínio específico. Uma parte do comportamento definido nas classes do framework é fixo, enquanto uma outra parte do comportamento de suas classes é extensível, sendo passível dessa forma de ser customizado de forma distinta para cada aplicação. Frameworks OO são, em geral, implementados usando linguagens e mecanismos de programação OO.

Os pontos de extensão (*hot-spots*) do framework são especificados através da definição de classes abstratas ou interfaces. Diversos padrões de projeto [45] podem ser usados na implementação de tais pontos de extensão. O desenvolvimento de aplicações baseada em frameworks requer a criação de classes que estendem seus pontos de extensão, em um processo que é denominado instanciação.

Um framework OO mantém as seguintes propriedades: (i) define um conjunto de classes (núcleo do framework) que juntas colaboram para implementar uma arquitetura de família de sistemas; (ii) oferece um conjunto de pontos de extensão na forma de classes abstratas ou interfaces; e (iii) especifica o fluxo de controle principal da aplicação através do comportamento das classes que representam o seu núcleo e que são responsáveis pela invocação das classes que estendem seus pontos de extensão.

A tecnologia de framework OO traz em geral um impacto positivo na produtividade e qualidade de desenvolvimento de aplicações. Frameworks OO possibilitam a reutilização não apenas de código de implementação, mas também reúso de projeto de soluções arquiteturais para um dado domínio. Frameworks OO maduros contribuem para a melhoria da qualidade das aplicações finais gerada, por oferecerem uma implementação estável e bem testada, além de possibilitar a codificação de menos linhas de código.

2.1.1. Problemas de Modularização

Apesar dos benefícios da tecnologia de frameworks na implementação de famílias de sistemas, diversos pesquisadores têm demonstrado a incapacidade dos mecanismos OO em modularizar determinados tipos de características encontrados em frameworks, tais como, características opcionais [14, 73, 118] e de composição tranversal [78, 94]. Basicamente, tem se concluído que o uso de técnicas OO na modularização de várias dessas características pode contribuir para (i) o aumento da complexidade do framework com a inclusão de várias funcionalidades que não são usadas por todas as instâncias do mesmo (esse fenômeno é denominado na literatura de "*overfeatured*" [28]); assim como traz (ii) dificuldades ou até imposibilita o reúso do framework em diferentes contextos

e cenários. As seções seguintes apresentam os principais problemas de modularização identificados pela comunidade e descrevem sintomas de tais problemas no contexto do framework JUnit.

2.1.1.1. Framework JUnit: Um Exemplo

O propósito principal do framework JUnit é possibilitar a implementação e execução de casos e suítes de teste para aplicações Java. Ele é usado principalmente para implementar testes de unidade, mas pode também ser usado para implementar testes de integração entre módulos. A Figura 1 apresenta as classes principais do framework JUnit[4]. Os seguintes componentes principais definem a estrutura do JUnit:

(i) **Framework**: esse componente agrega as classes responsáveis por especificar o comportamento básico de execução de casos e suítes de teste. As principais classes desse módulo são `TestCase` e `TestSuite`. Elas representam os pontos de extensão principais do framework. Usuários do framework estendem essas classes para criar e especificar casos e suítes específicos para o teste de suas aplicações;

(ii) **Runner**: as classes desse componente se responsabilizam pela disponibilização de uma interface gráfica que permita a inicialização e monitoramento da execução de casos e suítes de teste. São oferecidas três implementações alternativas para essa interface: (1) `textui` – baseada em linha de comando; (2) `awtui` – baseado na biblioteca AWT; e (3) `swingui` – baseado na biblioteca Java Swing;

(iii) **Extensions**: define classes que estendem o comportamento básico das classes de teste do JUnit. Exemplos de extensões disponíveis são: execução concorrente de testes em threads distintas; execução repetida de determinados casos e/ou suítes de teste; e configurações *default* durante a inicialização ou finalização de casos e/ou suítes de teste específicos.

As seções seguintes apresentam exemplos no contexto do JUnit dos principais problemas de modularização existentes em frameworks. Embora o framework JUnit tenha uma arquitetura bem organizada e estruturada, a

[4] JUnit Framework. URL: http://www.junit.org. 2007.

identificação desses problemas de modularização refletem dificuldades que poderiam ser encontradas durante a extensão ou evolução do framework. No contexto de frameworks maiores e mais complexos, esses problemas poderiam causar um desgaste gradual de sua arquitetura.

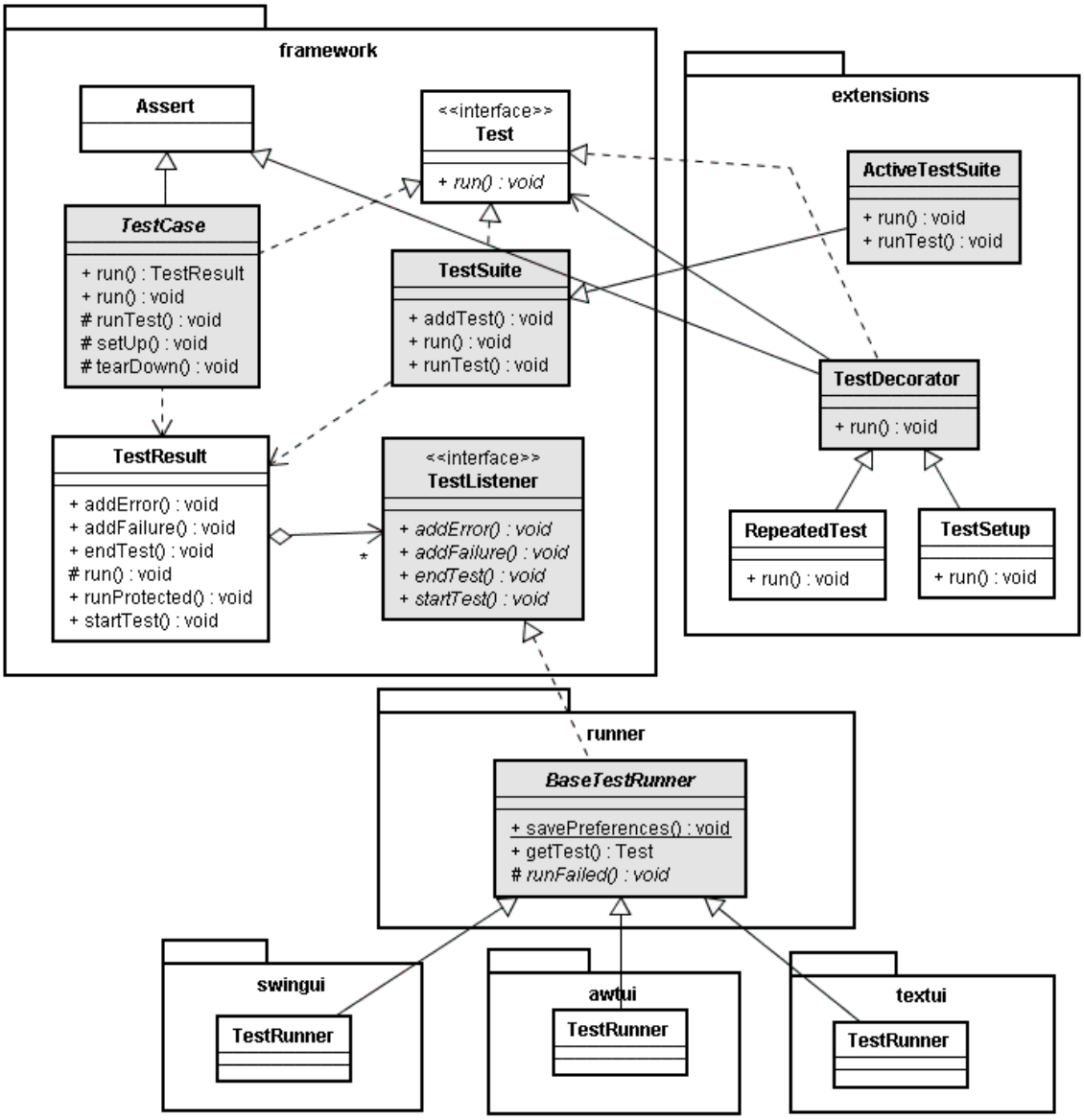

Figura 1. Diagrama de Classes do framework JUnit

2.1.1.2. Complexidade de Colaboração entre Objetos

Um framework OO define um conjunto de classes reusáveis abstratas e concretas que implementam uma arquitetura para uma família de sistemas. Colaborações complexas entre essas classes devem ser implementadas. Essas

colaborações representam funcionalidades comuns compartilhadas por diversas aplicações no domínio do framework. Cada classe do framework, em geral, tem de desempenhar diversos papéis [104], o que significa que elas precisam colaborar com diferentes classes para implementar suas respectivas responsabilidades. Conseqüentemente, o entendimento e manutenção das classes do framework pode se tornar uma tarefa difícil. Riehle et al [104] analisam os problemas de colaborações complexas entre objetos, e seu impacto no projeto e implementação de frameworks. Eles mostram como a complexidade de um framework OO aumenta quando suas classes desempenham diversos papéis.

No framework JUnit, por exemplo, o propósito principal da classe `TestResult` é coletar os resultados da execução de casos de teste. Para endereçar os requisitos do framework, entretanto, essa classe deve assumir um diferente papel adicional. A característica de monitoramento da execução dos testes está sobreposta no código das classes de execução dos testes para possibilitar a notificação para as classes da interface gráfica sobre o estado de execução (execução iniciada, correta, com falha, finalizada) dos testes. Tal característica é implementada através da declaração de um conjunto de objetos do tipo `TestListener` dentro da classe `TestResult`. A interface `TestListener` define os métodos necessários para implementar tal protocolo de notificação. Ela é implementada por classes do componente Runner, para apresentar os resultados do monitoramento da execução dos testes. A implementação dessa característica pode também ser vista como uma instanciação do padrão de projeto *Observer* [45]. A Figura 2 ilustra o espalhamento e entrelaçamento da implementação dessa característica por entre diferentes classes do framework JUnit.

De forma geral, a medida que novas características vão sendo incluídas nas classes pertecentes ao framework, o entendimento e evolução de sua estrutura interna pode se tornar uma atividade bastante complexa. A sobreposição de determinados padrões de projeto na estrutura de classes do framework pode ocasionar o surgimento de código entrelaçado e espalhado, tal como no exemplo ilustrado para o JUnit. Também a inclusão de novos papéis de classes para endereçar características opcionais ou de integração do framework com outros componentes, a serem exploradas nas seções seguintes, também contribui ainda mais para o aumento da complexidade do núcleo do framework.

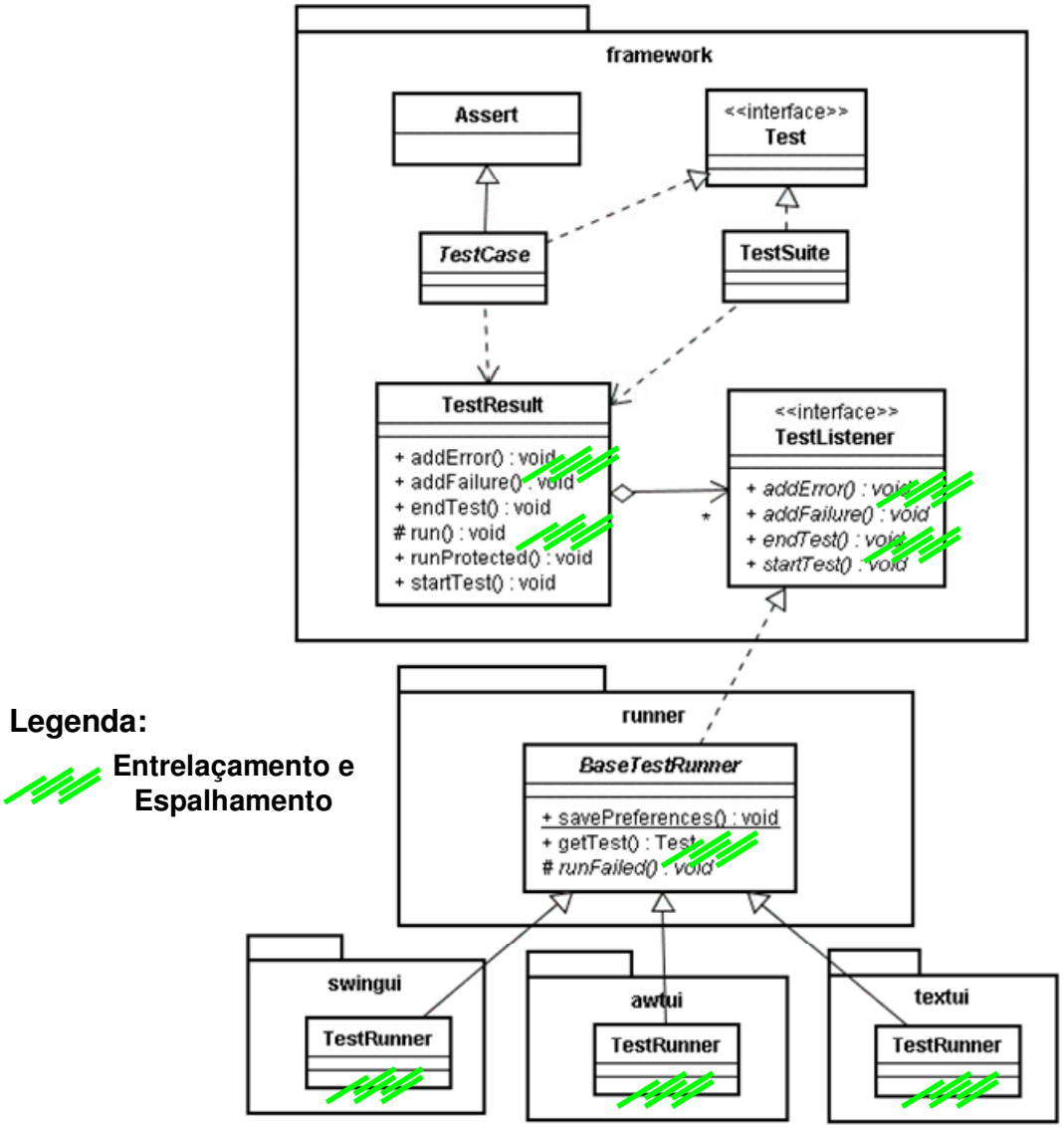

Figura 2. Implementação do monitoramento de testes no JUnit

2.1.1.3. Dificuldade de Modularização de Características Opcionais

Batory et al [14] discutem a dificuldade da técnica de framework em modularizar características opcionais. Uma característica opcional pode ser vista como uma funcionalidade do framework que não é usada em todas as suas instâncias. Esses pesquisadores ilustram [14] alternativas que desenvolvedores usualmente adotam para lidar com tal problema, tais como: (i) implementar a característica opcional no código de classes que implementam os pontos de extensão do framework durante a sua instanciação; e (ii) criar dois diferentes

frameworks, um contendo a característica opcional e outro sem ele, o que conseqüentemente traz problemas para lidar com a evolução dos dois artefatos.

Uma outra prática bastante adotada na implementação de características opcionais em frameworks é a especificação de relações de herança para definir comportamento adicional nas classes já existentes do framework. Essas classes acabam por representar uma característica do framework que foi estendida. No framework JUnit, por exemplo, a implementação da propriedade de execução concorrente de suítes de teste é realizada através do uso de relação de herança. A Figura 3 mostra a classe `ActiveTestSuite` herdando da classe `TestSuite` de forma a possibilitar a execução de suítes de teste em threads distintas.

Vários padrões de projeto podem também ser usados para prover implementações de características opcionais. Eles ilustram o uso sistemático de mecanismos de composição OO (herança e agregação), oferecendo flexibilidade para inclusão de novas implementações para um dado interesse do sistema. Exemplos de tais padrões são [45]: *Decorator*, *Observer* e *Adapter*. No framework JUnit, o padrão de projeto *Decorator* é usado para possibilitar a inclusão de novas propriedades de extensão de casos ou suítes de teste. A Figura 3 ilustra o uso do *Decorator* para implementação de tais propriedades. A classe `TestDecorator` representa o participante *Decorator* do padrão. As propriedades de extensão são definidas como especializações de tal classe, através da especificação de decoradores concretos (`ConcreteDecorator`). Exemplos de tais propriedades no JUnit são: (i) configurações de comandos de inicialização ou finalização de casos ou suítes de teste, implementado pela classe `TestSetup`; e (ii) repetições de testes implementado pela classe `RepeatedTest`. A associação de tais propriedades a casos ou suítes de teste específicos requer a instanciação de tais classes, passando como parâmetro em seus respectivos construtores, instâncias dos casos ou suítes de teste a serem estendidos.

O uso de padrões de projeto e relações de herança para implementar características opcionais pode trazer dificuldades para o entendimento e posterior manutenção do projeto de frameworks. Em particular, as seguintes dificuldades podem ser enfrentadas: (i) a inclusão de características opcionais através de relações de herança, implica no entendimento de novas classes que representam novos pontos de extensão do framework (tais como, as classes `TestDecorator`, `TestSetup`, `RepeatedTest` e `ActiveTestSuite` no caso do JUnit,); (ii) a

modularização de muitos padrões de projeto usando os mecanismos de OO incorre em problemas [51, 58] de entrelaçamento com outras funcionalidades e/ou espalhamento por entre várias classes, e isso como conseqüência também traz dificuldades para gerenciar as variabilidades implementadas no framework; e (iii) as formas de combinação e composição de várias dessas características opcionais precisa ser explicitamente codificada no código de instâncias do framework (classes `TestScenarioComposition1` e `TestScenarioComposition2` da Figura 3) e pode sofrer limitações em função dos mecanismos e soluções de projeto OO usados.

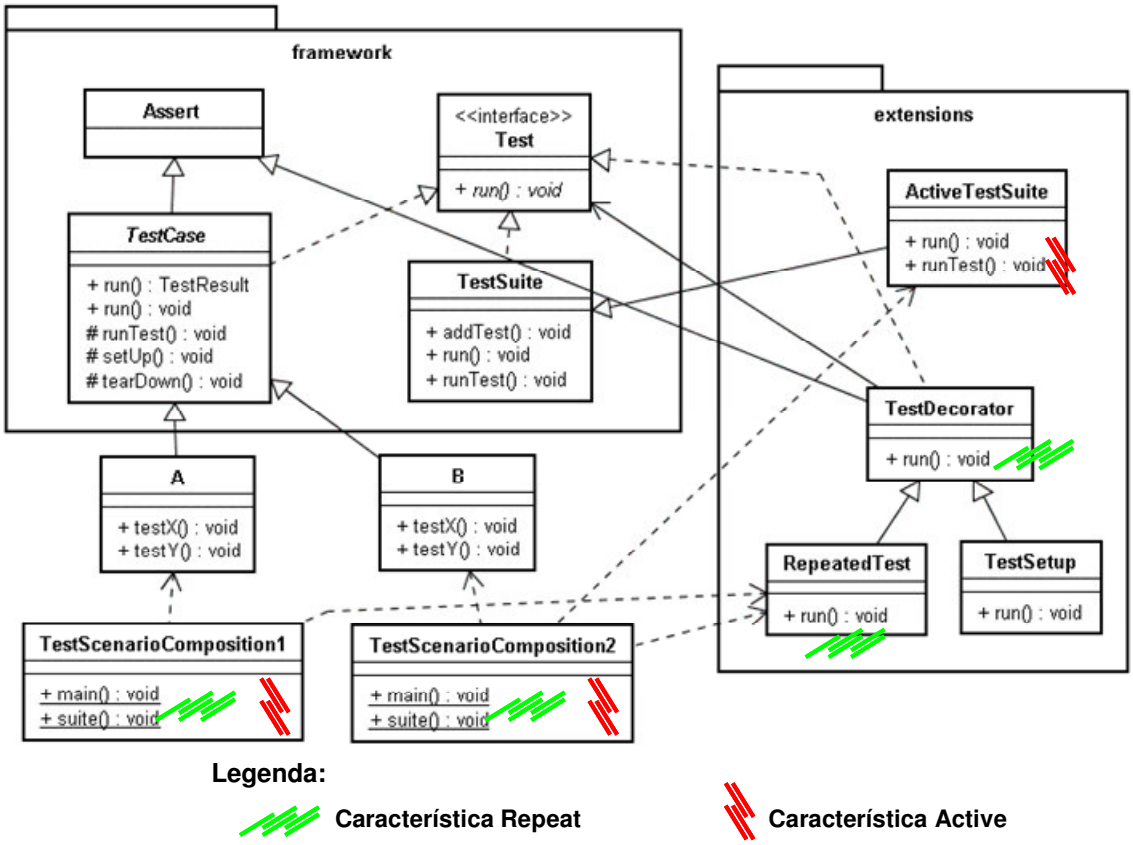

Figura 3. Implementação de propriedades de extensão no JUnit

A combinação e composição de características de extensão de casos e suítes de testes no caso do framework JUnit é explicitada diretamente no código de classes das suas instâncias. A Figura 4 mostra o código de 2 cenários possíveis de combinação das características de execução concorrente (`ActiveTestSuite`) e

execução repetitiva (`RepeatedTest`), aplicados a duas subclasses[5] (`A` e `B`) da classe `TestCase` do JUnit. Na classe `TestScenarioComposition`, os dois casos de testes são adicionados a um `ActiveTestSuite`, e em seguida, essa suíte de teste é executada 5 vezes. Na classe `TestScenarioComposition2`, os dois casos de teste são inicialmente caracterizados como repetitivos, e cada um dos casos de teste (total=2) é executado numa diferente *thread*. A diferença central entre esses os dois cenários implementados por essas classes é que: (i) o primeiro demanda a criação de 10 *threads*, pois o `ActiveTestSuite` é repetido 5 vezes, e cria uma thread para cada um dos 2 métodos de teste das classes A e B; e (ii) o segundo cenário ocasiona a criação de apenas 2 *threads*, um para cada `RepeatedTest`. Diferentes cenários de composição das características existentes do JUnit demandam a criação de diferentes classes na instância sendo criada pelo framework.


```
01 public class TestScenarioComposition {
02     public static void main(String[] args) {
03       junit.textui.TestRunner.run(AllTestsScenario2.suite());
04     }
05     public static Test suite() {
06        ActiveTestSuite suite= new ActiveTestSuite();
07
08        suite.addTestSuite(A.class);
09        suite.addTestSuite(B.class);
10
11        // Execute each suite 5 times
12        Test test= new RepeatedTest(suite, 5);
13        return test;
14     }
15 }
```

```
01 public class TestScenarioComposition2 {
02   public static void main(String[] args) {
03     junit.textui.TestRunner.run(AllTestsScenario4.suite());
04   }
05   public static Test suite() {
06     // Execute each suite 5 times
07     RepeatedTest test= new RepeatedTest(new TestSuite(A.class), 5);
08     RepeatedTest test2=new RepeatedTest(new TestSuite(B.class), 5);
09      ActiveTestSuite suite= new ActiveTestSuite();
10
11     suite.addTest(test);
12     suite.addTest(test2);
13     return suite;
14   }
15}
```

Figura 4. Composição de propriedades de extensão no JUnit

[5] Cada uma dessas subclasses definem 2 métodos de teste.

O problema de combinação e composição de várias características tem sido também abordado por vários outros trabalhos de pesquisa [20, 59, 118]. Zhang & Jacobsen [118] mostram, por exemplo, que no contexto do desenvolvimento de middlewares para sistemas distribuídos, várias características precisam ser compostas e coordenadas de diferentes formas e que o uso de mecanismos OO para a sua modularização remete ao problema de convolução de implementação [118]. Tal problema diz respeito à dificuldade de modularização de determinadas características e ao seu natural entrelaçamento com outras características existentes no middleware. Conseqüências diretas da perda de modularidade de tais características são: (i) perda de configurabilidade devido à impossibilidade de plugar/desplugar a característica do núcleo do middleware; e (ii) penalidades no desempenho do sistema devido à execução de código desnecessário quando a característica não deve estar presente.

2.1.1.4. Composições Transversais na Integração de Frameworks

Mattsson et al [90, 91] analisam os problemas e causas relacionados com a composição de frameworks. Para cada problema apresentado, eles propõem um conjunto de soluções OO. Uma composição de dois frameworks, como descrita pelos autores, pode ser vista como uma composição de um novo conjunto de características (representado por um framework) na estrutura de um outro. A seguir é apresentado um exemplo de composição entre frameworks.

Considere, por exemplo, um framework de medições e análise de qualidade de produtos para sistemas de manufatura. O framework *Measurement* apresentado em [17] endereça tal propósito. A Figura 5 apresenta as classes principais desse framework. Itens de um dado produto são analisados pelo sistema de medição (uma instância do framework *Measurement*), através de um ciclo composto pelos seguintes passos: (i) a classe `Trigger` indica que um item de produto será analisado pelo sistema de medição; (ii) sensores coletam informações (classes `PhysicalSensor`, `Sensor` e `UpdateStrategy`) sobre o produto. Um exemplo seria tirar fotografias do mesmo; (iii) em seguida, as informações coletadas do produto são comparadas com valores esperados (classe `MeasurementValue`) de forma a classificar o produto de acordo com critérios de qualidade; e (iv)

finalmente, ações são tomadas por acionadores baseadas na classificação do produto (classes `Actuator`, `PhysicalActuator` e `ActuationStrategy`). Um exemplo de possível ação seria emitir um rótulo no produto de acordo com sua categoria de qualidade.

Suponha agora que se deseja integrar o framework *Measurement* com um framework de interface gráfica (GUI) disponível, para endereçar a funcionalidade de visualização de informações do processo de medição, tais como, itens já processados por sensores e acionadores, tempo de processamento e rótulo atribuído a cada item de produto. A Figura 5 também apresenta um framework GUI baseado na biblioteca Java Swing, que permite apresentar dados em tabelas visuais. As classes abstratas `DataTableModel` e `ObjectTableModel` são usadas como fontes de dados das tabelas visuais. Três subclasses (`SensorTableModel`, `ActuatorTableModel` e `ProcessedItemTableModel`) da classe `ObjectTableModel` foram definidas para permitir a apresentação das informações do processo de medição. A composição entre os frameworks também requer, entretanto, que sejam feitas chamadas internas de determinadas classes (`Trigger`, `PhysicalSensor`, `PhysicalActuator` e `ActuationStrategy`) do framework *Measurement* para repassar informações para as subclasses de `ObjectTableModel`. Dessa forma, a composição entre os frameworks pode ser vista como transversal, uma vez que ela requer mudanças invasivas em várias classes e métodos do framework *Measurement* de forma a notificar classes do framework GUI sobre o andamento do processo de medição. A Figura 5 ilustra as modificações invasivas e transversais que devem ser feitas no framework Measurement para endereçar a composição.

Um estudo de análise [78] das soluções de composição de frameworks propostas por Mattsson et al [90, 91] foi conduzido por nosso grupo de pesquisa. O estudo concluiu que várias das soluções OO propostas pelos autores demandam modificações invasivas e trazem dificuldades para a implementação, entendimento e manutenção do código de composição do framework. Nossa análise envolveu a realização de um estudo de caso com a composição de características de quatro diferentes frameworks endereçando interesses de domínios verticais e horizontais [33]. A análise também mostrou que de nove soluções OO descritas por tais autores, seis delas possuem problemas de modularização e uma natureza transversal, e demandaram mudanças internas

invasivas no código do framework. Detalhes adicionais desse estudo serão apresentados no capítulo 7.

De forma geral, o estudo concluiu que o uso de mecanismos OO para implementar código de composição transversal entre frameworks traz dificuldades: (i) para entender e evoluir o código de composição dos frameworks que pode se entrelaçar e se espalhar por entre várias classes; e (ii) para desacoplar os dois frameworks e tratá-los como entidades individuais. O capítulo 7 apresenta o estudo de caso de composição do framework Measurement com outros diferentes frameworks de GUI, Persistência e Estatística, o qual foi usado para avaliar o uso de aspectos no projeto de diferentes composições transversais existentes entre frameworks.

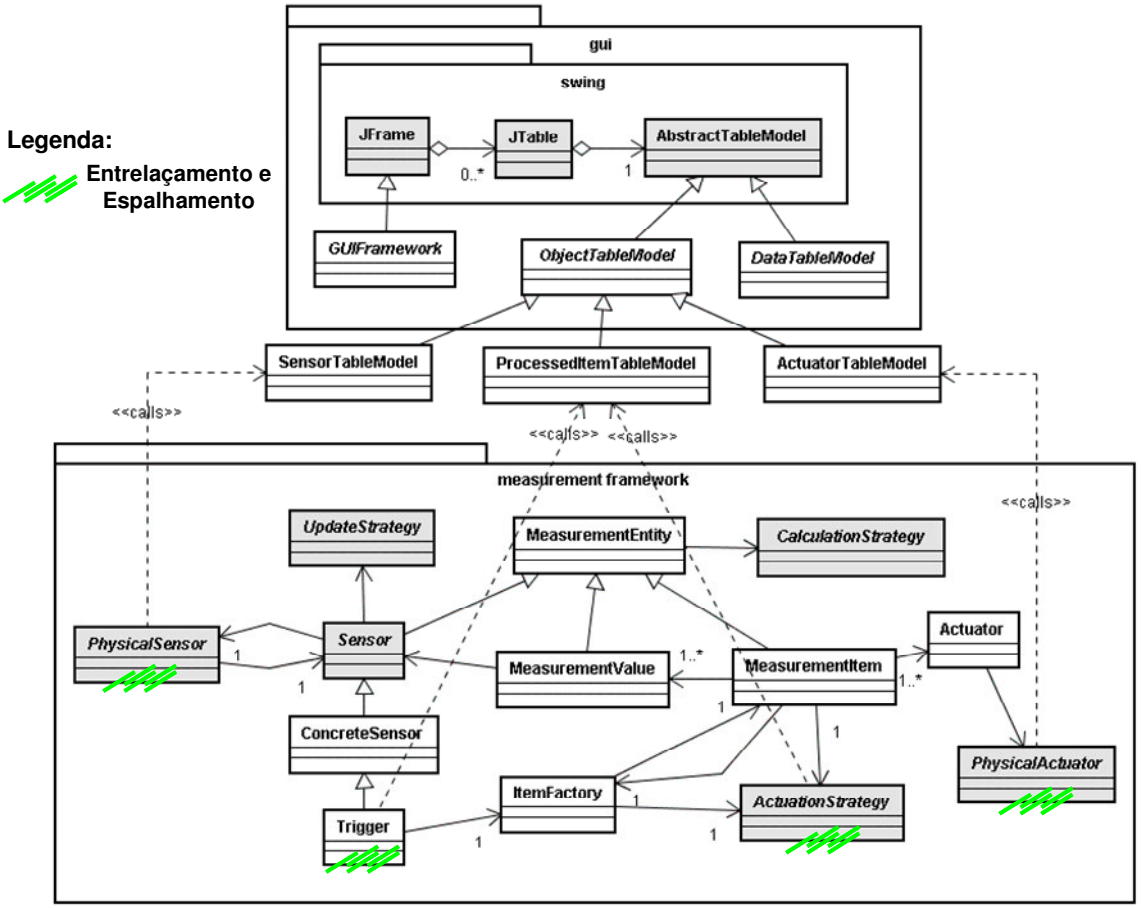

Figura 5. Exemplo de Composição entre Frameworks

2.2. Desenvolvimento de Software Orientado a Aspectos

Desenvolvimento de Software Orientado a Aspectos (DSOA) [42, 68] é uma abordagem de engenharia de software que objetiva a modularização dos chamados interesses transversais. Interesses traversais são aqueles que entrecortam diversos módulos dentro de um sistema de software. DSOA encoraja a descrição modular de sistemas de software complexo oferecendo mecanismos para separar claramente a funcionalidade básica do sistema, a qual pode ser endereçada por abordagens já propostas (tais como OO), dos interesses traversais. DSOA permite a modularização dos interesses tranversais propondo uma nova abstração, denominada aspecto, e novos mecanismos que permitem compor aspectos e abstracões dos paradigmas vigentes (exs: classes, interfaces, métodos, construtores). A composição entre aspectos e abstrações OO são realizadas dentro de pontos específicos, denominados pontos de junção. DSOA como a maioria dos paradigmas de desenvolvimento de software surgiu inicialmente no nível de implementação, sendo caracterizada pelo termo Programação Orientada a Aspectos (POA) [68].

Diversos trabalhos de pesquisa apresentados pela comunidade demonstram os benefícios que programação orientada a aspectos pode trazer para a modularização de propriedades sistêmicas ou requisitos não funcionais, tais como, rastreamento [31], auditoria [31], delimitação de transações [113], persistência [102, 113], segurança [85], tratamento de exceções [40], monitoramento e distribuição [113] . Trabalhos recentes [5, 7, 8, 73, 89, 94, 118] têm explorado o uso de técnicas orientadas a aspectos (OA) no projeto e implementação de arquiteturas de famílias de softwares, tais como, frameworks e linhas de produto de software. Nesses trabalhos, a abstração de aspectos é usada para melhorar a modularização de características transversais encontradas no domínio endereçado.

Esta tese propõe uma abordagem para o desenvolvimento de frameworks baseado em técnicas OA. São apresentadas diretrizes para a modularização de características transversais opcionais, alternativas e de integração existentes em frameworks usando a abstração de aspectos. As subseções seguintes apresentam: (i) uma visão geral da linguagem orientada a aspectos AspectJ utilizada nos

estudos de caso da tese; e (ii) o conceito de interface transversal, o qual está diretamente relacionado com a proposta de nossa abordagem.

2.2.1. AspectJ

AspectJ [69, 70] é uma extensão orientada a aspectos para a linguagem de programação Java. Ela é atualmente a abordagem mais amadurecida e popular de orientação a aspectos. AspectJ propõe uma nova abstração para modularização do software, denominada aspecto. Um aspecto em AspectJ é composto por pontos de corte (*pointcuts*), adendos (*advices*) e declarações inter-tipos (*inter-type declarations*). Os pontos de corte especificam os pontos de execução, denominados pontos de junção (*join points*) em AspectJ, de um programa Java que o aspecto deseja atuar/interceptar. Em AspectJ, diversos pontos de execução de classes Java são expostos para serem interceptados por aspectos, entre eles: execução de métodos e construtores, chamada de métodos ou construtores, execução de manipuladores de exceções, consulta ou modificação de valores de atributos, etc. Adendos definem o comportamento que será invocado durante a ocorrência daqueles pontos de junção. Três tipos de adendos podem ser definidos em AspectJ: (i) adendos `before` – são invocados sempre que um dado ponto de junção ocorre e antes do prosseguimento normal da sua execução; (ii) adendos `after` – são invocados depois da computação que define o ponto de junção; e (iii) adendos `around` – são executados sempre que um dado ponto de junção é alcançado, e têm o controle para decidir se a computação que caracteriza o ponto de junção deve ou não ser executada.

Um aspecto em AspectJ pode também definir declarações inter-tipos. Elas permitem modificar a estrutura de classes definidas em Java. Os seguintes tipos de declarações inter-tipos podem ser definidas: (i) introdução de atributos ou métodos em classes; (ii) introdução de relações de herança entre classes; e (iii) introdução de relações de implementação entre classes e interfaces.

A Figura 6 mostra o código de um aspecto exemplo, denominado `FaultHandler`. Esse aspecto é apresentado no guia de programação de AspectJ. O propósito desse aspecto é manipular eventuais faltas que ocorram em um servidor. O aspecto `FaultHandler` define: (i) a introdução do atributo `disabled` na classe

`Server` (linha 3); (ii) dois métodos internos do aspecto, o método privado `reportFault()` usado para indicação de faltas ocorridas (linhas 5-7) e o método público estático `fixServer()` que determina que o servidor voltou a operar normalmente (linhas 9-11); (iii) o ponto de corte `services()`, que intercepta todas as chamadas para métodos de escopo público da classe `Server` (linha 13); e (iv) dois adendos, associados ao ponto de corte `services()`, um do tipo `before` - que determina o lançamento de uma exceção quando o servidor não está operando normalmente (linhas 15-17), e outro do tipo `after throwing` - que modifica o estado do servidor para desabilitado sempre que uma exceção do tipo `FaultException` é lançada nos métodos públicos da classe `Server` (linhas 19-22).

```
01 public aspect FaultHandler {
02
03   private boolean Server.disabled = false;
04
05   private void reportFault() {
06     System.out.println("Failure! Please fix it.");
07   }
08
09   public static void fixServer(Server s) {
10     s.disabled = false;
11   }
12
13   pointcut services(Server s): target(s) && call(public * *(..));
14
15   before(Server s): services(s) {
16     if (s.disabled) throw new DisabledException();
17   }
18
19   after(Server s) throwing (FaultException e): services(s) {
20     s.disabled = true;
21     reportFault();
22   }
23 }
```

Figura 6. Aspecto FaultHandler

2.2.2. Interfaces Transversais

Duas propriedades, denominadas Inconsciência e Quantificação (*Obliviousness* e *Quantification*), têm sido identificadas como fundamentais para Programação Orientada a Aspectos por alguns pesquisadores [43]. A propriedade Quantificação diz respeito à capacidade dos programadores em escrever trechos de código com a seguinte forma: "*No programa P, quando quer que a condição C*

ocorra, execute a ação A". A linguagem AspectJ, por exemplo, oferece suporte para tal propriedade por meio das construções de ponto de corte, pontos de junção e adendos, citadas anteriormente. A propriedade de Inconsciência estabelece que desenvolvedores do código base - as classes do sistema que são potencialmente afetadas pelos aspectos - não precisam estar conscientes dos aspectos que as afetarão. Isso significa que os programadores não precisam preparar o código para ser afetado por aspectos. A seguinte sentença dos autores sintetiza ambas as propriedades [43]: "*POA pode ser entendida como o desejo de especificar comandos com o uso da quantificação sobre o comportamento de programas, e garantir que tais comandos são aplicados sobre programas escritos por programadores inconscientes da sua existência*".

Em um estudo recente, Sullivan et al [115] compara a metodologia baseada na propriedade de *Inconsciência* com uma nova abordagem para desenvolvimento OA baseado em *design rules*[6] [9]. Nessa abordagem, os autores propõem a especificação de interfaces entre o código base e os aspectos. Dessa forma, é necessária a especificação antecipada de pontos de junção do código base antes da sua própria codificação. Tais pontos de junção são posteriormente usados na codificação dos aspectos. Tal abordagem baseada em regras de projeto endereça o desacoplamento do código base e dos aspectos através de uma especificação explícita das interações e contratos entre tais elementos permitindo dessa forma o seu desenvolvimento paralelo. No estudo conduzido pelos autores [115] foram observados os benefícios que a abordagem traz para reduzir ou eliminar diversas desvantagens da abordagem *Inconsciente*, tais como: (i) a codificação de pontos de corte frágeis e complexos; e (ii) o forte acoplamento dos aspectos para detalhes de implementação do código base.

Griswold et al [56] mostram como as interfaces entre o código base e os aspectos, chamadas de interfaces transversais (XPIs - *crosscutting interfaces*) e propostas anteriomente pela abordagem baseada em *design rules*, podem ser implementadas em AspectJ. As XPIs são usadas para abstrair um comportamento existente no código base. A implementação das XPIs em AspectJ é composta de: (i) uma parte sintática – que permite expor pontos de junção específicos através da especificação de pontos de corte em AspectJ; e (ii) uma parte semântica – a qual

detalha o significado dos pontos de junção especificados e pode definir restrições (tais como, pré e pós-condições) que devem ser satisfeitas quando estendendo tais pontos de junção. A parte semântica pode ser parcialmente implementada com aspectos de *enforcement* [31] (implementados por meio das construções `declare error` e `declare warning` de AspectJ) ou definindo aspectos com contratos que garantem restrições específicas a serem satisfeitas antes e depois da execução de adendos.

Essa tese define o conceito de pontos de junção de extensão (EJPs), os quais são usados para estender o comportamento básico de um framework orientado a objeto. Os EJPs são inspirados na proposta de XPIs e podem ser vistos como o uso especializado das mesmas para auxiliar a modularização e composição de frameworks OO.

2.3. Desenvolvimento Generativo

Desenvolvimento Generativo (DG) [32, 33] endereça o estudo de métodos e ferramentas que habilitam a produção automática de membros de uma família de software a partir de especificações de alto nível. Ele promove a separação dos espaços de problema e solução dando flexibilidade de evoluir ambos de forma independente. Para prover tal separação, Czarnecki & Eisenecker [33] propõem o conceito de modelo de domínio generativo. Este modelo é composto de: (i) espaço de problema – que representa os conceitos e características existentes em um domínio específico; (ii) espaço de solução – que consiste na arquitetura de software e componentes usados para construir membros de uma família de sistemas; e (iii) conhecimento de configuração – que define como combinações específicas de características no espaço de problema são mapeadas para um conjunto de componentes de software no espaço de solução. Geradores de código representam o conhecimento de configuração dentro de um modelo generativo. A Figura 7 apresenta os elementos principais do modelo generativo.

DG foi proposto como uma abordagem baseada em engenharia de domínio [101]. Dessa forma, métodos de engenharia de domínio [33] podem ser usados

[6] *Design Rule* é definido como um parâmetro de projeto que pode ser usado como uma interface entre módulos com o objetivo de reduzir o acoplamento entre os mesmos.

para apoiar a definição de um modelo de domínio generativo. Atividades comuns encontradas em tais métodos são: (i) análise de domínio – a qual está interessada na definição de um domínio para uma familia de software específica e na identificação de características comuns e variáveis dentro desse domínio; (ii) projeto de domínio – a qual se concentra na definição de uma arquitetura e componentes comuns para este domínio; e (iii) implementação de domínio – que envolve a implementação de uma arquitetura e componentes anteriormente especificados no projeto de domínio. Duas novas atividades [33] precisam ser introduzidas em tais métodos de engenharia de domínio de forma a endereçar os objetivos de DG, são elas: (i) desenvolvimento de mecanismos apropriados para especificar membros da família de sistemas sendo desenvolvida. Linguagens específicas de domínio (DSLs - *Domain Specific Languages*) devem ser desenvolvidas para lidar com tal requisito; e (ii) modelagem detalhada do conhecimento de configuração de forma a automatizá-lo através do uso de geradores de código.

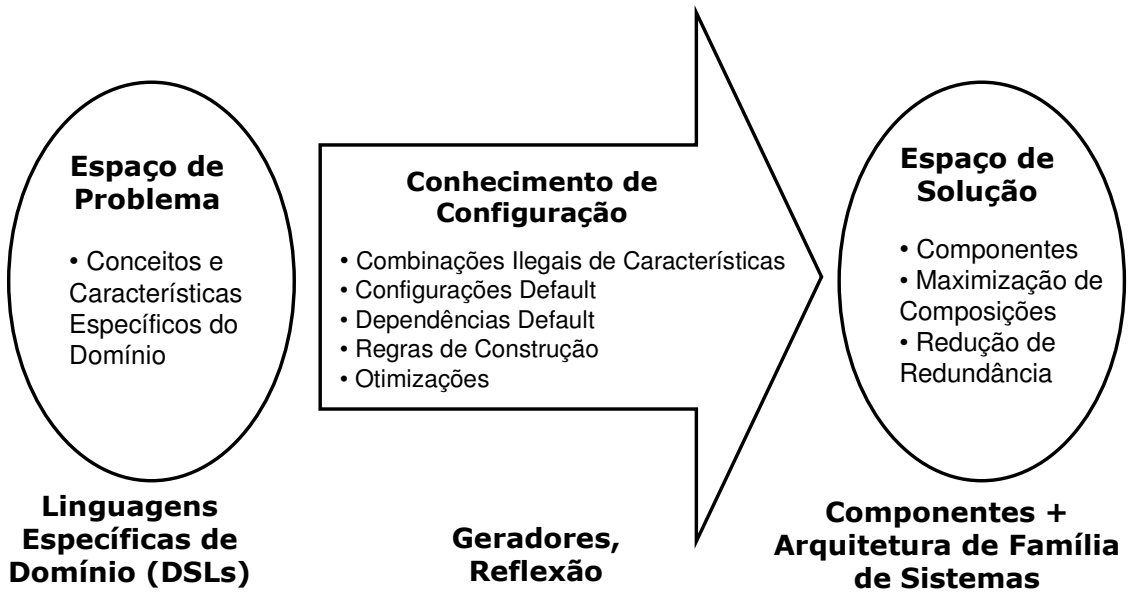

Figura 7. Modelo Generativo

De acordo com Czarnecki [32], o mapeamento entre o espaço de problema e o espaço de solução pode ser visto sobre duas perspectivas: (i) *visão de configuração* – nessa perspectiva o espaço de problema é visto como um conjunto de características a serem selecionadas e o espaço de solução consiste em um conjunto de componentes que podem ser combinados e customizados para derivar

uma instância de uma família de sistemas ou linha de produto. O mapeamento entre tais espaços é definido por regras de construção que indicam como certa combinação de características deve ser traduzida para determinadas configurações de componentes; e (ii) *visão transformacional* – na qual o espaço de problema é representado por uma linguagem específica de domínio (DSL) e o espaço de solução por uma linguagem de programação. O mapeamento entre tais espaço é definido através de transformações que geram código-fonte em uma linguagem de programação a partir de um programa especificado na DSL.

Essa tese propõe uma abordagem para desenvolvimento de frameworks usando programação orientada a aspectos. A abordagem define um modelo generativo orientado a aspectos cujo objetivo é facilitar a instanciação automática de variabilidades existentes em frameworks implementados com o uso da tecnologia de aspectos. O modelo generativo proposto contempla a visão de configuração. Também um conjunto de diretrizes de projeto e implementação de características transversais encontradas em frameworks compõe nossa abordagem. Essas diretrizes podem ser usadas para endereçar a modelagem e codificação de uma arquitetura de família de aplicações que determina o espaço de solução de nosso modelo generativo.

2.4. Potencial de Integração entre as Abordagens

Desenvolvimento Generativo e Desenvolvimento Orientado a Aspectos são abordagens com um grande potencial para serem integradas e serem usadas de forma complementar. Técnicas orientadas a aspectos podem ser usadas na definição de abordagens generativas, tanto: (i) no espaço de problema para, por exemplo, permitir a representação de relações tranversais entre características existentes no domínio endereçado; assim como (ii) no espaço de solução, para permitir a codificação de características transversais que não podem ser bem modularizadas por mecanismos OO. Esse uso integrado pode trazer benefícios diretos [74, 79, 80], tais como: (i) separação clara entre características não transversais e transversais nos espaços de problema e solução, e ao longo de todo o desenvolvimento de software; (ii) mapeamento direto de características transversais em aspectos; (iii) diminuição da complexidade de desenvolvimento

de geradores de código, porque a composição de características transversais pode ser realizada por compiladores de aspectos (*aspect weavers*); e (iv) melhoria no reuso de artefatos relacionados a características transversais.

Baseado na nossa experiência de desenvolvimento de uma abordagem generativa orientada a aspectos [79, 80], foram identificados requisitos úteis para a integração das abordagens generativas e de desenvolvimento orientado a aspectos. Tais requisitos guiaram o desenvolvimento dessa tese. A seguir é apresentada uma síntese dos mesmos:

(i) **suporte para representação de características transversais na análise de domínio**: a modelagem de características transversais em fases preliminares do desenvolvimento (tais como, modelagem de requisitos e da arquitetura) tem sido reconhecida recentemente como um tema de pesquisa importante em DSOA, denominado *Early Aspects* [10, 103]. Dessa forma, a extensão de modelos usados na análise e projeto de domínio para representar características transversais é fundamental para a sua modelagem e de suas respectivas interações com outras características de uma família de sistemas;

(ii) **especificação de arquiteturas orientadas a aspectos**: uma atividade fundamental quando construindo famílias de sistemas e linhas de produto é a modelagem de uma arquitetura de software que endereça características comuns e variáveis. O uso da abstração de aspectos traz novas possibilidades para a modularização de características transversais existentes em tais arquiteturas. Dessa forma, a definição de diretrizes, notações, princípios e padrões para a especificação de arquiteturas OA, com a adequada modularização e composição de suas características não tranversais e transversais, é um tópico importante que deve também ser tratado;

(iii) **especificação do conhecimento de configuração**: o conhecimento de configuração em um modelo generativo define como combinações de características no espaço de problema podem ser mapeadas para combinações específicas de componentes no espaço de solução. Diferentes tecnologias podem ser usadas para automatizar o conhecimento de configuração [32, 33], tais como, configuradores de produto [26] e tecnologias de geração de código baseada em templates [33, 34], em sistemas de transformação [33, 34], etc. A definição explícita de regras de mapeamento entre (I) características transversais presentes em DSLs usadas para modelar ou especificar produtos no espaço de problema, e

(II) artefatos de implementação (aspectos, frameworks, componentes, classes, templates) existentes no espaço de solução, é um requisito fundamental para habilitar a geração automática de membros de uma arquitetura de família de sistemas ou linha de produto;

(iv) **implementação de linguagens específicas de domínio**: DSLs permitem especificar membros de uma família da aplicação a serem instanciados ou gerados a partir de um conjunto de artefatos definidos para uma abordagem generativa. A modelagem de características variáveis transversais em arquiteturas de família de sistemas pode também requerer a sua explícita representação em DSLs, de forma a habilitar a sua customização. DSLs aspectuais [33, 110] têm sido propostas para endereçar tal requisito. O uso combinado de DSLs aspectuais e não aspectuais para expressar a composição de características no espaço de problema é outro tópico importante para ser endereçado na integração entre DG e DSOA;

(v) **implementação de frameworks com aspectos**: frameworks OO são uma tecnologia comum e bastante útil para a implementação de famílias de sistemas. Entretanto, como mencionado anteriormente (Seção 2.1.1), a modularização de determinadas características transversais em frameworks com mecanismos OO pode trazer dificuldades para o entendimento, gerência e manutenção de tais características. A tecnologia de aspectos habilita desenvolvedores a oferecer implementações modulares para características transversais. Dessa forma, a definição de diretrizes que auxiliem desenvolvedores no uso integrado das técnicas de framework e aspectos é também um elemento fundamental para a integração das abordagens generativa e de desenvolvimento orientado a aspectos.

A abordagem proposta nessa tese endereça vários dos requisitos discutidos para a integração das abordagens. Nossa abordagem é composta por: (1) um conjunto de diretrizes para modularização de características transversais encontradas em frameworks OO usando programação orientada a aspectos (Capítulo 4) que endereça o requisito (v); e (2) um modelo generativo orientado a aspectos (Capítulo 5) que propõe a extensão do modelo de características para explicitar a existência de características transversais endereçando assim o requisito (i) e um conjunto de regras de mapeamento entre características transversais e elementos de implementação baseados em aspectos que endereçam

o requisito (iii) permitindo a especificação de modelos de configuração. O requisito (ii) de especificação de arquiteturas orientadas a aspectos foi também explorado, embora não no contexto direto desta tese de doutorado, através da investigação do uso da abstração de aspectos em linguagens de descrição de arquiteturais (ADLs) [12, 13] e notações baseadas em UML [23, 24]. Finalmente, o requisito (iv) foi endereçado parcialmente já que nossa extensão do modelo de características para representar relações tranversais pode ser usada como uma DSL para a derivação de frameworks e linhas de produtos baseadas em aspectos.

2.5. Sumário

Esse capítulo apresentou três abordagens que vêm sendo exploradas no desenvolvimento de famílias de sistemas e linhas de produto, são elas: (i) frameworks orientados a objetos; (ii) desenvolvimento de software orientado a aspectos (DSOA); e (iii) desenvolvimento generativo (DG). Foram descritos diversos problemas de modularização de cararacterísticas transversais que são encontrados durante o desenvolvimento de frameworks, quando usando mecanismos OO para implementá-las. Dada a natureza transversal de tais características, elas são candidatas naturais a serem implementadas com técnicas orientadas a aspectos. O capítulo foi concluído apresentando uma discussão sobre o potencial de sinergia existente entre as três abordagens. O capítulo seguinte apresenta uma abordagem para o desenvolvimento de frameworks, centrada em técnicas oferecidas pelas abordagens de DSOA e DG. Tal abordagem lida com os problemas de modularização de frameworks descritos nesse capítulo.