Trabalho de Conclusão de Curso

Integrando JML a Interfaces Emergentes

Luiz Américo Mesquita Jardim
lulajardim@gmail.com

Orientadores:

Márcio de Medeiros Ribeiro
Flávio Mota Medeiros

Maceió, Julho de 2012

Agradecimentos

Inicialmente, agradeço em especial os meus pais pelo e apoio financeiro em boa parte de minha vida, pela paciência e por acreditar em mim. Agradeço também a minha namorada pelos ótimos momentos que sempre passamos juntos, pelo apoio e pela força que ela sempre me dá para que eu alcance meus resultados.

Agradeço também aos meus orientadores: Márcio Ribeiro, que apesar de estar sempre muito ocupado em sua tese de doutorado e seus trabalhos de pesquisa, sempre arrumou tempo e fez um ótimo trabalho; Flávio Medeiros, que me ajudou muito no experimento executado neste trabalho. Obrigado pela força e pela paciência de vocês.

Agradeço aos meus chefes no Banco do Nordeste que sempre foram prestativos e flexíveis para que eu pudesse realizar as reuniões para preparação do trabalho, a pesquisa e o trabalho de fato.

Finalmente, agradeço aos meus amigos que me dão forças e apoio sempre que preciso e todos as pessoas que sempre acreditaram em mim e me deram forças para que eu pudesse finalizar este trabalho.

Muito obrigado!

Resumo

Em Linhas de Produto de Software (LPS) uma das técnicas utilizadas para implementação de *features* é a compilação condicional. Nesta técnica as *features* são delimitadas através da utilização de diretivas, como por exemplo *#ifdef* e *#endif*. Porém, ao utilizar tais diretivas o desenvolvedor polui e ofusca o código e dificulta a separação de interesses. A separação virtual de interesses, do inglês, *Virtual Separation of Concerns* (VSoC), foi proposta para reduzir essas dificuldades encontradas com a utilização da técnica de compilação condicional. Nesta abordagem o desenvolvedor consegue esconder o código de determinadas *features* para focar seu esforço nas *features* de interesse. Embora esconder *features* diminua a poluição e ofuscamento do código ajudando o desenvolvedor a focar no seu interesse, esta abordagem pode também esconder dependência entre *features*, levando a introdução de erros por parte do desenvolvedor. Com o objetivo de amenizar este problema e atingir a modularidade indo além do VSoC, pesquisadores propuseram recentemente o conceito de interfaces emergentes (*Emergent Interfaces - EI*). Para atingir o objetivo desejado, EI busca no código da LPS as dependências existentes entre *features* mostrando estas dependências para o desenvolvedor. A interface emergente ajuda o desenvolvedor a detectar dependências existentes na LPS, porém, como estas interfaces não detalham tais dependências, elas possuem uma baixa expressividade. Como consequência disto, o desenvoveldor precisa revelar as *features* que apresentam dependências para entender como tais dependências influenciam sua tarefa de manutenção. Este trabalho busca melhorar a expressividade das interfaces emergentes através do uso da *Java Modeling Language* (JML) que segue o paradigma *Design by Contract* (DbC). Além disto, este trabalho mostra um experimento utlizado para avaliar o nível da expressividade destas novas interfaces emergentes (*Emergent Interface with JML - EIJ*) e então determinar se elas possuem um maior nível de expressividade do que as interfaces emergentes regulares.

Abstract

In Software Product Lines (SPL), conditional compilation is one of the techniques used to implement features. This technique uses preprocessors directives, such as #ifdef and #endif, to encompass feature code. The problem of this approach is that the excessive use of these directives pollutes and obfuscates the source code and makes separation of concerns hard to accomplish. The Virtual Separation of Concerns (VSoC) was proposed to reduce these difficulties encountered while implementing features using the conditional compilation approach. Thus, a developer can hide the code of certain features to focus on a specific feature. Although hiding features diminishes the pollution and obfuscation of the code, it can also hide dependencies among features, which may lead to the introduction of errors by developers. Some researchers have recently proposed a new approach to reach modularization beyond the usage of VSoC. The concept of Emergent Interfaces (EI) searches the code of a SPL to find dependencies among features and show these to the developers. The emergent interface helps developers detect existing dependencies between features, but, since it does not give any specific detail of these dependencies, it has a low expressiveness level. The low expressiveness of the EI creates a need for developers to understand the hidden features in order to realize how the dependencies affect their current maintenance job. In this context, this work attempts to raise the expressiveness level of the EI by using the Java Modeling Language (JML), which follows the Design By Contract paradigm. We also show an experiment used to evaluate the expressiveness level of these new interfaces, called Emergent Interfaces with JML (EIJ), and then determine if they have a higher expressiveness level than the regular Emergent Interfaces.

Conteúdo

Lista de Figuras

Lista de Códigos

Lista de Tabelas

Capítulo 1

Introdução

Uma Linha de Produto de Software (*LPS*) consiste em um conjunto de sistemas que possuem um núcleo em comum e um conjunto de *features* que diferenciam os sistemas entre si. *Feature* é uma unidade semântica utilizada para representar a variabilidade entre os produtos [11]. O conjunto de *features* de uma *LPS* pode ser representado através de um *feature model* [7]. Além do conjunto de *features*, o *feature model* também define o escopo da *LPS*, suas restrições e os produtos que podemos formar.

Uma das técnicas para implementação de *features* é a compilação condicional. Esta técnica consiste em utilizar diretivas de pré-processadores, por exemplo *#ifdef* e *#endif* [8, 9, 21] para delimitar o código de cada *feature*. Um pré-processador recebe a configuração da *LPS*, ou seja, recebe a lista das *features* que devem estar presentes no produto, analisa o código, e comenta o código das *features* que não fazem parte do produto.

A técnica de compilação condicional permite uma granularidade fina na implementação das *features* já que é possível delimitar *features* em um código das mais diferentes maneiras. Porém, a liberdade gerada pelo fino nível de granularidade pode também dificultar o entendimento do código. A separação virtual de interesses, do inglês, *Virtual Separation of Concerns* (*VSoC*), propõe uma forma de diminuir as diversas desvantagens da compilação condicional como o ofuscamento do código [1]. Com *VSoC* é possível ocultar o código de *features* que não são necessárias para a tarefa atual de um determinado desenvolvedor. Assim, este se concentra apenas no que interessa para a realização da manutenção ou desenvolvimento.

1.1 Problema

Embora *VSoC* facilite o entendimento do código, *VSoC* também facilita a introdução de erros por parte do desenvolvedor visto que podem existir dependências entre a *feature* sendo mantida pelo desenvolvedor e *features* escondidas. Como forma de tentar sanar este problema, Ribeiro e Borba [16] propuseram recentemente o conceito de interfaces emergentes (*EI*). Tal conceito tem como objetivo apresentar sob demanda as dependências entre as *features* presentes

no código de uma *LPS*. Isto evita que o desenvolvedor perca o foco no ponto de interesse da manutenção e permite que ele tome conhecimento das dependências entre a feature sendo mantida e as outras existentes no código da *LPS*.

Entretanto, existem situações em que a informação gerada pela *EI* não é suficiente para auxiliar o desenvolvedor na tomada de decisão ou no entendimento do código. Isto ocorre pelo baixo nível de expressividade da mesma já que a mensagem apenas informa que uma linha de código de uma *feature* depende de uma outra linha de código de uma outra *feature* ou até da mesma *feature*. Como consequência o desenvolvedor precisa revelar e analisar outras *features* para entender a dependência existente no sistema e, portanto, aumenta o tempo necessário para realizar sua manutenção.

1.2 Visão geral da solução proposta

Este trabalho propõe uma forma de melhorar a expressividade das interfaces emergentes atráves do uso do conceito de *Design by Contract* (*DbC*), em que estabelece benefícios e obrigações para um cliente e seu implementador.

Os contratos tem por objetivo melhorar a confiabilidade dos sistemas de software [12]. Através destes contratos é possível especificar o comportamento esperado para as classes e seus métodos, como por exemplo: um método de uma classe que calcula a raíz quadrada de um valor x espera que seu chamador informe um valor $x \geq 0$; um método de busca binária em um vetor espera que o vetor de busca esteja ordenado.

Com a informação gerada pelos contratos do paradigma DbC, é possível mostrar uma interface emergente que, além de explicitar a existência de uma dependência entre diferentes trechos do código, também explica de que forma se dá esta dependência.

Para implementar os contratos do paradigma DbC em Java utilizamos uma linguagem chamada *Java Modeling Language* (JML) e, portanto, chamaremos a nova interface emergente de EIJ - *Emergent Interface with JML*, traduzindo, interface emergente com JML.

Uma das principais vantagens da utilização da EIJ é a diminuição na quebra desnecessária da modularidade durante a realização de uma manutenção: o desenvolvedor não precisa revelar a *feature* dependente para restringir a dependência pois a EIJ já mostra a dependência e a restrição para o desenvolvedor. Outra vantagem da EIJ é a diminuição do tempo necessário para realizar uma manutenção visto que com a informação sobre a dependência e sua restrilção, o desenvolvedor não precisa perder tempo analisando *features* que não fazem parte do escopo da sua manutenção.

1.3 Organização do Trabalho de Conclusão de Curso

Os conceitos aqui introduzidos serão vistos de forma mais detalhada no Capítulo 2 deste trabalho. O Capítulo 3 introduz algumas situações em que o problema da baixa expressividade das *EI*s é percebido. Logo após, no Capítulo 4, mostramos como podemos utilizar as informações geradas pelos contratos firmados entre um método e seu chamador para aumentar a expressividade das interfaces emergentes (*EIJ*). No Capítulo 5 apresentamos um experimento para avaliar se a solução proposta melhora a expressividade das interfaces emergentes. Esse capítulo também explica a execução do experimento, apresenta os dados coletados e realiza a análise dos dados. Finalmente, o Capítulo 6 apresenta as considerações finais sobre o trabalho.

Capítulo 2

Fundamentação Teórica

O objetivo deste capítulo é realizar uma breve explicação sobre os conceitos necessários para compreensão deste trabalho de conclusão de curso.

2.1 Linha de Produtos de Software (LPS)

Atualmente, o conceito mais utilizado para definir linha de produtos de software, é o conceito proposto por Clements e Northrop [5].

> *Uma linha de produtos de software é um conjunto de sistemas de software que compartilham um conjunto de características comuns e gerenciadas, que satisfazem as necessidades de um segmento em particular de mercado, e que são desenvolvidos a partir de um núcleo comum de artefatos e de uma forma preestabelecida [5]*

Em outras palavras, isto significa que uma linha de produtos de software é um conjunto de sistemas que possuem um núcleo em comum e um conjunto de funcionalidades (*features*) que diferenciam os sistemas entre si.

Ainda de acordo com Clements e Northrop [5], existem diversos benefícios com as práticas de linhas de produtos. São alguns deles:

- **Customização em massa:** A abordagem *LPS* explicitamente separa a parte fixa da parte variável e isto facilita a customização em massa dos sistemas;
- **Redução de custo e tempo de desenvolvimento de um novo produto:** A utilização de um núcleo em comum significa que ao criar um novo produto dentro de uma linha de produto de software estamos reutilizando diversas linhas de código junto com seus requisítos e arquitetura, portanto, estamos reduzindo o tempo e o custo de desenvolvimento de um novo produto. Além disso, mesmo a parte variável pode ser reutilizada visto que a parte em comum entre alguns sistemas de uma linha de produto de software pode ir além do seu núcleo;

- **Melhoria da qualidade do sistema:** Como as funcionalidades de uma *LPS*, tanto as funcionalidades do núcleo quanto as da parte variável, são utilizadas em diversos produtos, elas são testadas e revisadas diversas vezes. Isto aumenta a chance de detectar falhas causando uma melhoria individual na qualidade dos componentes e, como consequência, na *LPS* de uma forma geral.

De acordo com Kang et al., o *feature model* de uma linha de produtos de software é a representação do domínio de suas *features* e de seus relacionamentos [7]. O *feature model* pode conter os seguintes tipos de features: obrigatórias/mandatórias, que estão presente em todas as configurações; opcionais, que podem ou não estar presentes. Quanto as filhas de uma mesma *feature* mãe, o *feature model* pode conter os seguintes tipos: alternativas (ou exclusivo), onde apenas uma *feature* filha deve estar presente; ou inclusivo, onde pelo menos uma *feature* filha deve estar presente. Um *feature model* também pode conter restrições representadas por uma implicação lógica ($A \rightarrow B$) que permitem forçar a presença, ou não, de uma *feature B* quando a *feature A* estiver presente. Todos estes elementos aqui explicados podem ser visto na Figura 2.1 que apresenta um exemplo de *feature model* de um carro.

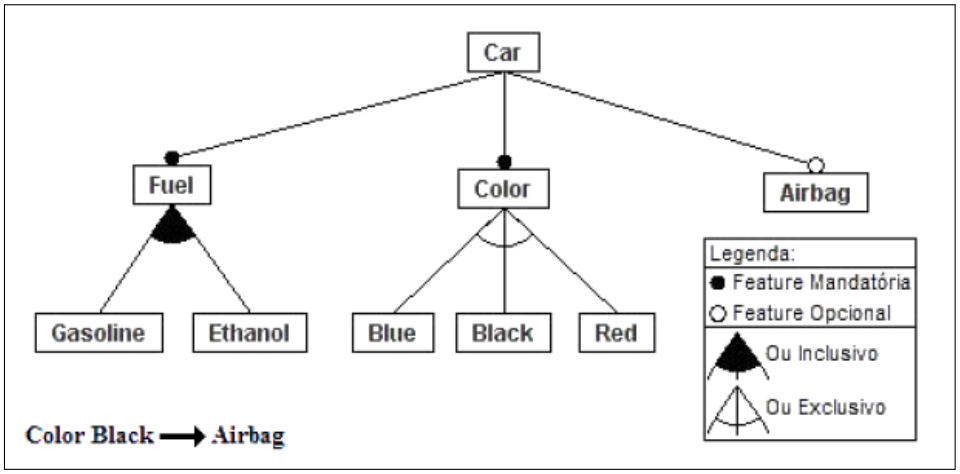


Figura 2.1: Exemplo de feature model.

Cada produto em uma linha de produto de software tem sua *configuração*, isto é, um conjunto de *features* habilitadas. Com base em um *feature model* é possível determinar todas as configurações válidas e inválidas de uma LPS. Na Figura 2.1 temos um exemplo de *feature model* de um carro qualquer e com base nele podemos formar os seguintes exemplos de configurações válidas:

- $\mathbb{C} = \{Fuel\ Gasoline, Fuel\ Ethanol, Color\ Blue, Airbag\}$;
- $\mathbb{C} = \{Fuel\ Gasoline, Color\ Red\}$;
- $\mathbb{C} = \{Fuel\ Gasoline, Color\ Black, Airbag\}$.

Ainda no mesmo *feature model*, é possível perceber que os itens a seguir são exemplos de configurações inválidas e portanto não são produtos da referida LPS:

- $\mathbb{C} = \{Color\ Black, Airbag\}$ - Todo carro precisa ter ao menos um tipo de combustível;
- $\mathbb{C} = \{Fuel\ Gasoline, Fuel\ Ethanol, Color\ Black\}$ - Todo carro preto deve ter *Airbag*.

2.2 Compilação Condicional

Existem diversas técnicas para implementar uma *feature*. De acordo com Kästner et al. [8], as diversas técnicas se dividem em dois tipos. São elas:

- Composicionais: As *features* são implementadas em módulos distintos e depois unidas para formar um produto. Como esta técnica foge do escopo deste trabalho, não iremos explicá-las. Os detalhes desta podem ser encontrados em Kästner et al. [8];
- Anotativas: As *features* são implementadas atráves de anotações implícitas ou explícitas no código.

Compilação condicional é uma técnica para implementação de *features* que se encaixa na categoria anotativa. Esta técnica se utiliza de anotações, como por exemplo *#ifdef* e *#endif*, para delimitar as *features* e um pré-processador que recebe a configuração de um produto para processar estas anotações e gerar o referido produto. Geralmente a configuração do produto é passada para o pré-processador através de um arquivo com uma lista de definições de variáveis. Cada uma das variáveis representa uma *feature* e a presença da mesma no arquivo implica na presença *feature* no produto. Analogamente, a ausência da variável implica na ausência *feature* na configuração do produto.

```
1  public class Math {
2    // ...
3    //#ifdef SQRT
4    //public double sqrt(double x){
5    //  // ...
6    //}
7    //#endif
8
9    //#ifdef POW
10   public double pow(double x, int y){
11     // ...
12   }
```

```
13    //#endif
14
15    // ...
16  }
```

Código 2.1: Código com anotações de compilação condicional.

O Código 2.1 apresenta o exemplo de uma classe com funções matemáticas com anotações para marcar as features *SQRT* e *POW*. Observe que no referido código as anotações estão dentro de comentários pois as mesmas não fazem parte do escopo de palavras reservadas do Java. No exemplo também vemos que as linhas 4, 5 e 6, referentes a *feature SQRT*, estão comentadas. Isto significa que o pré-processador não encontrou a definição da variável *SQRT*, ou seja, este é um produto com uma configuração que não contêm a *feature SQRT*. O mesmo não acontece com as linhas referentes a *feature POW*, ou seja, o pré-processador encontrou uma definição para a variável *POW*.

```
1  public class Stack {
2    public void push(Object o,
3  //#ifdef TXN
4    , Transaction txn
5  //#endif
6    ) {
7      if (o == null
8    //#ifdef TXN
9      || txn == null
10   //#endif
11     ) return;
12   //#ifdef TXN
13     Lock l = txn.lock(o);
14   //#endif
15     elementData[size++] = o;
16   //#ifdef TXN
17     l.unlock();
18   //#endif
19     fireStackChanged();
20   }
21 }
```

Código 2.2: Código com granularidade fina.

Uma das principais vantagens da compilação condicional, e dos métodos anotativos de uma forma geral, é sua granularidade fina pois é possível delimitar *features* das mais diversas formas. Um exemplo disto pode ser observado no Código 2.2 aonde a *feature* TXN esta delimitando desde linhas de código simples (linhas 13 e 17) até parâmetros de uma função (linha 4) ou cláusulas de um *if* (linha 9).

2.3 Virtual Separation of Concerns (VSoC)

Apesar dos métodos anotativos serem conhecidos por sua granularidade fina, esses são também conhecidos por ofuscar o código. Isto significa que a legibilidade do código pode ser drasticamente reduzida. O Código 2.2 além de exemplificar a granularidade fina, também demonstra o quão ilegível o código pode se tornar com a utilização da compilação condicional.

A separação virtual de interesses, em inglês *Virtual Separation of Concerns* (VSoC), é uma abordagem utilizada por Kästner et al. [8] na ferramenta *CIDE - Colored Integrated Development Environment* [8]. Esta abordagem permite ao desenvolvedor esconder *features* irrelevantes à sua tarefa atual, fazendo com que ele possa se concentrar apenas na *feature* de interesse.

```
1  public class Stack {
2    public void push(Object o,
3    [redacted]
4    ) {
5      if (o == null
6    [redacted]
7      ) return;
8    [redacted]
9      elementData[size++] = o;
10   [redacted]
11     fireStackChanged();
12   }
13 }
```

Código 2.3: Código com utilização de VSoC.

O Código 2.3 exemplifica como ficaria o Código 2.2 com a utlização de VSoC para esconder virtualmente a *feature* TXN através da ferramenta CIDE. Note que o referido código apresenta quadrados cinzas representando que a *feature* TXN está escondida. Isto se deve a semântica utilizada pela ferramenta CIDE que utiliza diferentes cores para delimitar cada uma das *features*. Na referida ferramenta, uma *feature* escondida será representada por um pequeno quadrado colorido e uma *feature* aparente terá cada uma de suas respectivas linhas pintadas com um fundo colorido.

2.4 Interfaces Emergentes (EI)

Quando um desenvolvedor utiliza VSoC para esconder *features* irrelevantes à sua tarefa atual pode facilitar a compreensão do código e, consequentemente, facilitar sua atual tarefa de manutenção. Entretanto, esta abordagem pode esconder trechos de código que possuam alguma dependência, como por exemplo uma mesma variável utilizada em *features* distintas. A consequência disto é que o desenvolvedor não terá conhecimento da dependência existente enquanto não revelar os trechos de código ocultos. Portanto, o mesmo tem uma maior probabilidade de quebrar a referida dependência causando um erro de compilação ou um erro em tempo de execução.

```
1 public static void main(){
2   int x = 9;
3
4   // ...
5   //#ifdef sqrt
6   x = Math.sqrt(x);
7   //#endif
8   // ...
9 }
```

Código 2.4: Código com trechos interligados.

Considere que um desenvolvedor deve realizar uma manutenção no Código 2.4 com o objetivo de alterar o valor da variável *x* para um número negativo. Considere também que o desenvolvedor iniciará sua tarefa com a *feature* SQRT escondida. Observe que se o desenvolvedor completar sua manutenção sem o devido cuidado, o mesmo introduzirá um erro que somente será identificado em tempo de execução. Ou seja, para que o desenvolvedor identifique a dependência existente no código, o mesmo precisa revelar a *feature* SQRT. É importante salientar que este exemplo é muito simplificado contemplando um método com apenas uma *feature* e uma variável presente. Em sistemas reais um mesmo método pode conter trechos de diversas *features* dificultando ainda mais a tarefa do desenvolvedor.

Inicialmente proposto por Ribeiro e Borba, o conceito de interfaces emergentes, em inglês *Emergent Interfaces* (*EI*), tenta alcançar modularidade indo além do VSoC, ou seja, o conceito tenta alcançar a independência de desenvolvimento, mutabilidade e compreensibilidade entre as features [17].

Este conceito estabelece interfaces, geradas sob demanda, entre os trechos de código das diferentes *features*. Seu uso é dado através de uma ferramenta em que os pontos de manutenção devem ser selecionados, e a interface emergente é computada através de uma análises de fluxo de dados. As interfaces emergentes resultantes estabelecem contratos entre os pontos de

manutenção selecionados e linhas que possuam algum tipo de dependência com os pontos de manutenção selecionados. Por exemplo, a interface emergente para a linha 2 do Código 2.4 é a seguinte:

Feature SQRT requires x [line 6]

Com o contrato fornecido pela interface emergente, o desenvolvedor não precisa revelar a *feature* SQRT para saber que existe uma dependência entre a variável *x* e a referida *feature*. Porém, o desenvolvedor não sabe como se dá esta dependência e, sendo assim, o mesmo ainda precisa revelar a *feature* SQRT para analisar a linha apontada pela EI e saber de que forma o mesmo pode completar sua tarefa.

Este trabalho propõe uma solução para melhorar a expressividade das EIs através da utilização de contratos JML. Para compreender a solução que será proposta é necessário compreender os conceito de *Design by Contract* e *Java Modeling Language* que serão explicados nas seções a seguir (Seções 2.5 e 2.6).

2.5 Design by Contract (DbC)

Proposto por Meyer, o conceito Design by Contract (DbC) visa melhorar a confiabilidade dos sistemas de software [12]. De acordo com Meyer, confiabilidade é definida como uma combinação de robustez e nível de acerto ou, de uma forma direta, é a ausência de *bugs* [13].

Os contratos utilizados no nosso dia a dia estabelecem vínculos entre duas ou mais pessoas. Em um contrato cada parte tem suas obrigações e seus benefícios sendo que os benefícios de uma parte são as obrigações da outra parte e vice-versa. A Tabela 2.1 ilustra um exemplo de um contrato estabelecido entre um cliente e uma empresa fornecedora de sinal de TV em qualidade *HD* (*High Definition*).

Parte	Obrigações	Benefícios
Cliente	Pagar a quantia mensal de R$ 60,00; Possuir uma TV com entrada *HDMI*; Possuir rede elétrica estabilizada.	Receber 40 canais de TV em qualidade *HD*
Empresa	Fornecer 40 canais de TV em qualidade *HD*	Não precisar lidar com: clientes que não pagaram a fatura, TVs incompatíveis e instabilidades na rede elétrica.

Tabela 2.1: Contrato entre cliente e empresa.

O conceito de DbC faz uma analogia aos contratos reais estabelecendo contratos entre as classes de um sistema e seus clientes. Ao estabelecer estes contratos o cliente precisa garantir certas condições antes de chamar um método definido por uma classe (pré-condições) e, em retorno, a classe garantirá certas propriedades mantidas após a chamada do referido método (pós-condições). Assim como num contrato real, a obrigação do cliente é benefício para a

classe e a obrigação da classe é benefício para o cliente. A Tabela 2.2 mostra um exemplo de contrato entre o método *sqrt* do Código 2.1 e seu possível chamador.

Parte	Obrigações	Benefícios
Chamador	Fornecer $x \geq 0$	Não precisar validar se $y * y \approx x$
Método *SQRT*	Fornecer um valor y tal que $y * y \approx x$	Não precisar se preocupar se a raiz quadrada de x é um número real.

Tabela 2.2: Exemplo de contrato entre um método e seu chamador.

2.6 Java Modeling Language (JML)

Java Modeling Language (JML) é uma linguagem de modelagem para Java que segue os paradigmas de DbC, isto é, JML especifica como podemos estabelecer, em Java, as obrigações e benefícios para um cliente e seu implementador [10].

Como as palavras-chaves da especificação JML não fazem parte do escopo de palavras-chaves do Java, as anotações JML devem ser inseridas no código Java na forma de comentários iniciados ou delimitados pelo símbolo arroba (@). O código JML pode ser inserido tanto em comentários de linha única (//) quanto em comentários de múltiplas linhas (/* ... */). O Código 2.5 ilustra o contrato exemplificado na Tabela 2.2 escrito das duas formas.

```
1  //@ requires x >= 0.0
2  //@ ensures JMLDouble.approximatelyEqualTo(x, \result * \result
       , 0.00001);
3  public double sqrt(double x){
4    // ...
5  }
6
7  /*@ requires x >= 0.0
8    @ ensures JMLDouble.approximatelyEqualTo(x, \result * \result
       , 0.00001); @*/
9  public double sqrt(double x){
10   // ...
11 }
```

Código 2.5: Contrato escrito em JML em um código Java.

No Código 2.5 além da sintaxe para inserir as anotações JML no código Java, vemos também algumas das diversas palavras-chaves presentes na especificação da linguagem JML. São elas:

- ***requires***: Esta chave estabelece as obrigações que o chamador do referido método deve obedecer. A chave *requires* é pré-condição para que o método inicie sua execução, ou seja,

um método so iniciará sua execução se seu chamador obedecer todas as pré-condições estabelecidas.

- ***ensures***: Esta é a chave de pós-condição do método e estabelece as obrigações do referido método para com seu chamador. O método deve atender todas as cláusulas da pós-condição ao termino de sua execução para não quebrar o contrato.

- ***JMLDouble***: Implementação do tipo de dado *Double* para JML.

- ***approximatelyEqualTo***: Método pertencente à classe JMLDouble que calcula se um número real é aproximadamente igual a outro com base em um grau epsilon de precisão.

- ***\result***: Esta chave se refere ao retorno do método em questão.

Este trabalho se utiliza apenas da chave que estabelece as obrigações do chamador do método, a chave *requires*. Para detalhes aprofundados da sintaxe JML sugerimos consultar outras fontes [10].

Outra chave que aparecerá neste trabalho, especificamente na Seção 4.1 do Capítulo 4, é a chave *also*. Tal chave indica que o referido método está herdando as especificações JML do seu supertipo. Esta chave é obrigatória sempre que um método possuir um supertipo, ou seja, a chave *also* é obrigatória nos métodos da subclasse que forem uma redefinição ou customização de um método da superclasse. A chave *also* só está presente neste trabalho por uma particularidade da linguagem JML, mesmo assim, ela não faz parte do escopo deste trabalho.

A utilização de JML traz diversos benefícios para o sistema como por exemplo:

- Documentação do sistema: Ao utilizar JML o desenvolvedor está automaticamente documentando o sistema utilizando uma linguagem formal.

- Melhora na performance para sistemas estáveis: Sistemas estáveis podem ter sua performance melhorada ao remover checagens desnecessárias e excessivas do comportamento das variáveis do sistema. Utilizando JML é possível desabilitar facilmente a validação dos contratos e, consequentemente, checagens desnecessárias e excessivas de variáveis.

Capítulo 3

Problemática

O objetivo deste capítulo é mostrar como as interfaces emergentes auxiliam a detectar ou evitar os problemas de manutenção. Mostraremos também por que, em alguns aspectos, as *EI*s não são suficientemente expressivas para que no capítulo seguinte possamos mostrar como podemos melhorar as interfaces emergentes utilizando JML.

Para demonstrar tais fatos, utilizaremos três exemplos que motivam a utilização das interfaces emergentes. Nos cenários a seguir, as features podem ou não estar escondidas como forma de fazer com que o programador foque apenas no que interessa.

Os trechos de código apresentados neste capítulo foram simplificados para facilitar o entendimento do texto.

3.1 PDF

Nesta seção temos o exemplo de uma LPS com uma feature que gera um *PDF* a partir de um formulário quando o mesmo não possui erros de preenchimento.

```
1  public Error getError(){
2    Error e = null;
3
4    // ...
5
6    return e;
7  }
8
9  public void validateForm(){
10   Error result = getError();
11
12   // ...
13
```

```
14    //#ifdef PDF
15 }
```

Código 3.1: Método *validateForm()* com a feature *PDF* oculta.

Conforme o código 3.1, percebe-se que o método *getError()* retorna apenas um erro. Neste caso, o primeiro erro que ele encontra no formulário. Embora seja válida e funcione, o ideal seria que esta implementação fosse feita de forma que todos os erros fossem mostrados ao clicar no botão de validação.

Percebe-se que para chegar no resultado desejado é necessário modificar o método *getError()* e que qualquer solução proposta resultará na mudança do tipo de retorno do referido método e, como consequência, teremos que mudar o tipo da variável *result* do método *validateForm()*. Portanto, antes de iniciar as modificações é importante verificar qual o impacto da mudança do tipo da variável *result*. Tendo em vista a presença de *features* escondidas, iniciaremos a análise utilizando a interface emergente referente à linha 10 do Código 3.1.

Feature PDF requires result.

A mensagem apresentada mostra que a variável *result* está sendo utilizada pela *feature* PDF, porém, não é possível afirmar com certeza que uma alteração no tipo da referida variável resultará em um erro no sistema. Para ter tal certeza, precisamos revelar o trecho da *feature* PDF no método *validateForm()*.

```
14 //#ifdef PDF
15 Error empty = Error.EMPTY;
16 if (empty.equals(result)) {
17   // ...
18 }
19 //#endif
```

Código 3.2: Código da feature *PDF* no método *validateForm()*

Vemos, conforme o Código 3.2, que a variável *result*, do tipo *Error*, está sendo comparada à variável *empty*, também do tipo *Error*. Sendo assim, uma alteração no tipo da variável *result* resultará em um erro no sistema.

Observe que com o uso da interface emergente é possível evitar erros de manutenção, porém, a mensagem não é expressiva o suficiente. Este problema de expressividade ocorre pois a interface emergente consegue apenas reconhecer a existência de uma dependência entre a variável *result* e a *feature* PDF, não conseguindo detalhes sobre tal dependência. A consequência disto é que perdemos tempo pois precisamos revelar a *feature* PDF para conseguir os detalhes desejados sobre a referida dependência.

Apesar dos códigos na situação real serem maiores e mais complexos, o método *equals()* é intuitivo, o que facilita e reduz o tempo necessário para entendimento do código. Todavia,

precisamos revelar a *feature* PDF para entender a dependência existente aumentando o tempo necessário para a manutenção.

3.2 Best Lap

Neste cenário temos um jogo de corrida para celulares conhecido como *Best Lap*. O jogo possui um método chamado *computeLevel()* utilizado para computar o *score* total de uma corrida, conforme o Código 3.3. Sabe-se também que este método possui uma *feature* chamada *ARENA*, que gerencia o *ranking online* do jogo.

```
1 public void computeLevel(){
2   double totalScore = perfectCuvesCounter * CURVE_BONUS +
       perfectStraightCounter * STRAIGHT_BONUS + gc_levelManager.
       getCurrentCountryId();
3
4   // ...
5
6   //#ifdef ARENA
7
8 }
```

Código 3.3: Método *computeLevel()* com a feature *ARENA* oculta.

Como *perfectCurvesCounter*, *perfectStraightCounter* e as constantes de bonus (*CURVE_BONUS* e *STRAIGHT_BONUS*) são números maiores ou iguais a zero e não existem números negativos identificando os países ($gc\_levelManager.getCurrentCountryId() \geq 0$), teremos um *score* sempre positivo ou zero ($totalScore \geq 0$).

Baseado nisso, podemos imaginar que em algum momento alguém envolvido no projeto do Best Lap queira adicionar uma penalidade ao *score*, por exemplo, baseada no melhor e no pior tempo da corrida. Com tal penalidade, teríamos a possibilidade do *score* assumir valores negativos. Para iniciar a manutenção vamos analisar o código utilizando a interface emergente para a linha 2 do código em questão.

Feature ARENA requires totalScore.

Novamente a mensagem apresentada pela interface emergente não é suficiente para definir se alguma modificação no comportamento da variável *totalScore* quebraria a feature *ARENA*, principalmente numa situação em que a modificação proposta não afetaria o tipo da variável em questão. Precisamos então acessar a feature *ARENA* para garantir que nenhum erro seja causado em tempo de execução.

```
1 //#ifdef ARENA
2 NetworkFacade.setScore(totalScore);
3 //#endif
```

Código 3.4: Trecho da feature *ARENA* no método *computeLevel()*

Conforme o Código 3.4, a variável *totalScore* está sendo repassada para o método *setScore()* da classe *NetworkFacade*. Novamente precisamos acessar outra parte do código, desta vez a classe *NetworkFacade*.

```
1 public class NetworkFacade{
2   // ...
3   public static void setScore(double score){
4     // ...
5     if (score < 0) score = 0;
6     // ...
7   }
8   // ...
9 }
```

Código 3.5: Classe *NetworkFacade*.

Vemos no Código 3.5 que sempre que o *score* passado for negativo, ele será definido como zero. Nesta situação, se realizarmos a modificação proposta anteriormente, teremos um problema em que os *scores* negativos serão gravados como zero no *ranking online*.

Portanto, é possível evitar o erro com ajuda da EI, porém, a mensagem não é expressiva o suficiente. Isto ocorre porque a EI consegue apenas obter informação sobre a existência de dependências entre o ponto de manutenção e outras *features*, porém, não consegue detalhar estas dependências. Como resultado disto perdemos muito tempo pois precisamos acessar a feature *ARENA* e posteriormente a classe *NetworkFacade* para finalmente conseguir o detalhamento da dependência necessário para prosseguir com a manutenção da LPS ($totalScore \geq 0$).

É importante salientar que, assim como no cenário anterior, os métodos e classes aqui apresentados estão simplificados. Na situação real, o programador precisaria analisar muitas linhas de código antes de encontrar a linha representada neste texto pela linha 5 do Código 3.5.

3.3 Xadrez

No xadrez, uma das regras opcionais é a regra de tempo. Geralmente utilizada em torneios, esta regra possui diversas variações, sendo uma delas a determinação de um tempo limite para cada jogador executar sua jogada.

Neste cenário temos um jogo de Xadrez com esta regra implementada como uma *feature* opcional chamada *TIMER*. Para controlar a regra, a classe Timer é usada em associação com a classe *Player*. Se essa *feature* estiver ativa, o tempo limite em minutos será perguntado ao jogador antes do início do jogo e o valor será passado para o construtor da classe *Player* da variável *minutes* do tipo *integer*.

```
1  public void createPlayers() {
2    // ...
3
4    //#ifdef TIMER
5    int minutes = 0;
6    //#endif
7
8    // ...
9
10   //#ifdef TIMER
11   minutes = Integer.parseInt(JOptionPane.showInputDialog(null,
        "Qual o tempo limite de cada jogada?", "Xadrez",
        JOptionPane.QUESTION_MESSAGE));
12   //#endif
13
14   // ...
15
16   this.player1 = newPlayer(player1Name, Color.WHITE
17   //#ifdef TIMER
18   , minutes
19   //#endif
20   );
21   this.player2 = newPlayer(player2Name, Color.WHITE,
22   //#ifdef TIMER
23   , minutes
24   //#endif
25   );
26 }
```

Código 3.6: Método *createPlayers()* sem aplicação de VSoC

Imagine que o desenvolvedor resolva modificar o Código 3.6 e adicionar ao *feature model* do jogo uma nova *feature* opcional chamada *PERSONALIZED_TIMER* tornando opcional a personalização do tempo, ou seja, a *feature* para escolher o tempo da jogada passará a ser

opcional. Como a pergunta solicitando ao jogador o tempo limite de cada jogada está na linha 11, a primeira iniciativa que tal desenvolvedor toma é coletar a interface emergente para a referida linha do código.

Feature TIMER requires minutes

Com essa informação sabemos que apenas a *feature TIMER* utiliza a variável *minutes* e embora o código esteja resumido, na situação real todas as linhas que usam a variável *minutes* estão apresentadas no Código 3.6. A solução consiste em adicionar uma cláusula ao *ifdef* na linha 10 do código em questão. O resultado pode ser visto de forma resumida no Código 3.7.

```
9  // ...
10 //#ifdef TIMER && PERSONALIZED_TIMER
11 minutes = Integer.parseInt(JOptionPane.showInputDialog(null, "
      Qual o tempo limite de cada jogada?", "Xadrez", JOptionPane.
      QUESTION_MESSAGE));
12 //#endif
13 // ...
```

Código 3.7: *Feature PERSONALIZED_TIMER*.

Observe que a variável *minutes* possui valor inicial igual a zero (linha 5 do Código 3.6) e quando a nova *feature PERSONALIZED_TIMER* estiver desativada, a variável não terá o seu valor inicial alterado e será passada ao construtor da classe *Player* sem que nenhuma verificação seja feita. Como não existe nenhuma verificação internamente na classe *Player* e nem na classe *Timer*, quando o jogo iniciar, o contador da classe *Player* iniciará zerado e o jogador com o primeiro movimento perderá o jogo automaticamente. Portanto, a modificação realizada gerou um erro detectável apenas em tempo de execução.

Assim como nos cenários anteriores a interface emergente possui uma baixa expressividade informando apenas a existência de uma dependência entre a *feature TIMER* e a variável *minutes*. A consequência disto é que o desenvolvedor precisa perder tempo e analisar o código de outros métodos e classes para conseguir os detalhes desejados sobre a dependência informada pela interface emergente ($minutes > 0$).

Capítulo 4

Interface Emergentes com JML (*EIJ*)

No Capítulo 3 introduzimos três cenários mostrando que um desenvolvedor, ao realizar uma manutenção no código, pode gerar inconsistências se ele não analisar cuidadosamente o código das outras *features*. Vimos também como as EIs ajudam a identificar problemas que podem ser introduzidos ou que já foram introduzidos num sistema. Porém, devido a sua baixa expressividade, ainda assim precisamos olhar o código de outras *features*, métodos, classes, etc, para consegui detalhes sobre a dependência existente entre as *features*. Em outras palavras, a baixa expressividade das interfaces emergentes resulta em diminuição da modularidade no desenvolvimento de uma *LPS*.

Neste capítulo proporemos a integração de contratos JML a interfaces emergentes (EIJ - *Emergent Interfaces with JML*) para aumentar a expressividade das EIs. Espera-se também que a aplicação das EIJs ajudem a diminuir o tempo necessário para realizar uma manutenção em uma LPS, bem como a probabilidade da introdução de erros por parte do desenvolvedor durante uma manutenção.

4.1 PDF

Como vimos anteriormente (Seção 3.1), este cenário trata de uma LPS que possui um gerador de *PDF* de um formulário livre de erros de preenchimento. O problema aqui foi gerado após uma alteração no método de validação do formulário e, como resultado disso, o botão de gerar o *PDF* passou a estar sempre desativado. Utilizamos uma EI para auxiliar na busca do problema e descobrimos que a *feature PDF* estava comparando um *Error* com uma lista de *Error* (*List<Error>*).

Conseguir detalhes sobre a dependência apresentada na interface emergente da Seção 3.1 é uma tarefa simples já que todas as informações necessárias sobre a dependência encontram-se em um só método (*validateForm()*). Ou seja, basta revelar a *feature PDF* para perceber que a variável *result* precisa ser do tipo *Error*. Todavia, ao revelar a *feature PDF* estamos quebrando a modularidade para poder realizar a manutenção proposta. Na busca de evitar essa quebra,

descreveremos a seguir como podemos utilizar contratos JML neste cenário de forma que a interface emergente melhore sua expressividade.

O problema ocorreu porque por padrão o método *equals()* de qualquer objeto recebe um parâmetro do tipo *Object* e assim pode-se comparar, por exemplo, um *Date* com um *Integer*, ou com um *String*, ou com qualquer outro *Object*.

Se modificássemos o tipo do parâmetro do método *equals()* da classe *Error* para o tipo *Error*, o compilador automaticamente detectaria o problema sem a necessidade do uso de interfaces emergentes, porém este é um padrão Java que não é interessante ser quebrado.

Note que o método *equals* tem como obrigação determinar, de forma confiável, se o objeto a qual ele pertence é igual ao objeto que lhe for passado. Tendo em vista que não faz sentido verificar a igualdade entre objetos de classes distintas, a obrigação do chamador do método *equals* é passar um objeto que seja instância da classe *Error*. Portanto, existe uma relação de contrato entre o método *equals* da classe *Error* e seu chamador. O esquema deste contrato pode ser visualizado na Tabela 4.1.

Parte	Obrigações	Benefícios
Chamador	Fornecer para o referido método um objeto instância da classe *Error*	Ter a garantia de que seu objeto foi comparado corretamente.
Método *equals*	Determinar, de forma confiável, se o objeto a qual ele pertence é igual ao parâmetro recebido	Não precisar se preocupar se o parâmetro recebido é uma instância da classe *Error*.

Tabela 4.1: Contrato entre o método *equals* da classe *Error* e seu chamador.

A solução proposta aqui é utilizar a informação do contrato da Tabela 4.1 para gerar uma nova interface emergente mais expressiva. A integração será através da linguagem JML, portanto, para que seja possível a análise traduzimos o contrato da referida tabela para JML (Código 4.1).

```
1  public class Error{
2    // ...
3
4    //@ also
5    //@ requires obj instanceof Error;
6    public boolean equals(Object obj){
7      // ...
8    }
9
10   // ...
11 }
```

Código 4.1: Classe *Error* com um contrato escrito em JML no método *equals()*.

Com a informação fornecida pelo contrato JML, a interface emergente referente à linha 10 do Código 3.1, seria a seguinte:

Feature PDF requires result as instance of Error.

Esta nova interface emergente é mais expressiva, ou seja, o desenvolvedor obtêm mais informações sobre a dependência entre a *feature PDF* e a variável *result* sem precisar revelar a *feature PDF* e sem perder seu foco no ponto central da manutenção.

Observe que apenas utilizando esta nova interface emergente é possível perceber que para o desenvolvedor realizar sua manutenção (mostrar todos os erros do formulário) será necessário também alterar o código da *feature PDF*. E então, agora com as informações detalhadas sobre a variável *result* e suas dependências, o desenvolvedor pode tomar as devidas providências: continuar com sua manutenção e somente revelar a *feature PDF* para modificar o contrato; ou desistir de sua manutenção.

4.2 Best Lap

Na Seção 3.2 vimos que este cenário trata de um jogo de corrida para celulares conhecido como *Best Lap*. Este jogo utiliza um cálculo, que assume apenas valores maiores ou iguais a zero, para gerar uma pontuação de uma corrida.

Imagine que um segundo desenvolvedor, desconhecendo a situação apresentada na Seção 3.2, decida criar a possibilidade de pontuações negativas. Assim como anteriormente, este outro desenvolvedor se utiliza da EI para garantir que sua manutenção não causará problema algum.

Feature ARENA requires totalScore.

Esta mensagem da EI permite apenas saber que a *feature ARENA* está utilizando a variável *totalScore*, ou seja, este segundo desenvolvedor poderia imaginar que sua modificação não quebraria a *feature ARENA* já que sua manutenção alteraria apenas o valor da varíavel e não o tipo da varíavel. Através das informações da Seção 3.2, já sabemos que ele se engana e se continuar com esta modificação, ele criará uma inconsistência no jogo.

Olhando novamente o método *setScore* da classe *NetworkFacade*, observamos que a primeira linha do referido método verifica se o parâmetro recebido (*score*) é menor do que zero. Se redefenirmos o comportamento do referido método de forma que o mesmo repasse a responsabilidade de checagem do parâmetro *score* para o seu chamador, criamos uma relação de contrato semelhante ao contrato visto na Seção 4.1, entre o método *setScore* e seu chamador. O esquema deste contrato pode ser visto na Tabela 4.2.

Agora, assim como na seção anterior, podemos utilizar a informação do contrato apresentado na Tabela 4.2 para gerar uma nova interface emergente mais expressiva. O referido contrato traduzido para JML pode ser visto no Código 4.2.

Parte	Obrigações	Benefícios
Chamador	Fornecer para o referido método uma pontuação positiva ou nula. (*score* $\geq$ 0)	Ter a garantia de que a pontuação foi gravada de forma ordenada no banco de dados *online*.
Método *equals*	Gravar de forma ordenada no banco de dados *online* a pontuação recebida através do parâmetro *score*.	Pontuações negativas não afetarão seu comportamento.

Tabela 4.2: Contrato entre o método *setScore* da classe *NetworkFacade* e seu chamador.

```
1  public class NetworkFacade{
2    // ...
3    //@ requires score >= 0
4    public static void setScore(double score){
5      // ...
6      if (score < 0) score = 0;
7      // ...
8    }
9    // ...
10 }
```

Código 4.2: Classe *NetworkFacade* com um contrato escrito em JML no método *setScore()*.

Então, utilizando a informação do contrato JML para gerar a nova interface, obteremos o seguinte resultado:

Feature ARENA requires totalScore >= 0.

Agora, com esta nova interface emergente, o desenvolvedor sabe que se realmente for necessário seguir em frente com a manutenção, ele precisará acessar a *feature ARENA* para buscar o referido contrato e alterar o comportamento da referida *feature* de modo que a mesma aceite pontuações negativas. Se o desenvolvedor ignorar tal fato, ele estará quebrando o contrato e a referida *feature* e não será possível prever qual o comportamento terá a LPS.

Observe que conseguimos chegar a conclusão mostrada no parágrafo anterior utilizando apenas a informação da EIJ, ou seja, novamente não foi necessário revelar a *feature ARENA* ou qualquer outra *feature* sem relação à atual tarefa de manutenção. A nova interface emergente apresentada se mostrou mais expressiva do que a interface mostrada na Seção 3.2 pois além de mostrar a existência de uma dependência entre a variável *totalScore* e a *feature ARENA*, conseguiu restringir a mesma (*totalScore >= 0*).

4.3 Xadrez

Na Seção 3.3, mostramos um jogo de Xadrez com a regra de limite de tempo por jogada de um jogador implementada em uma *feature* opcional chamada *TIMER*. Vimos que uma inconsistência foi causada após a criação de uma nova *feature* chamada *PERSONALIZED_TIMER* criada para tornar opcional a personalização da variável *minutes*.

Vimos também que nesta situação o desenvolvedor causou a inconsistência mesmo possuindo acesso às *features* e a interface emergente não acrescentou nenhuma informação ao mesmo. Assim como nos cenários anteriores, podemos adicionar um contrato ao código e melhorar a expressividade da EI.

Como foi dito na Seção 3.3, quando a *feature TIMER* está ativa, o construtor da classe *Player* recebe um tempo em minutos e repassa para uma nova instância de objeto da classe *Timer*. A instância deste nova objeto será responsável pelo controle do tempo de cada jogada do jogador determinado pela instância do objeto da classe *Player*. Um cronômetro regressivo com valor inicial nulo não faz sentido portanto, podemos existe uma relação de contrato entre o construtor da classe *Timer* e seu chamador. O referido contrato pode ser visualizado na Tabela 4.3

Parte	Obrigações	Benefícios
Chamador	Fornecer para o referido método um tempo válido em minutos. ($minutes > 0$)	Ter a garantia de que a pontuação foi gravada de forma ordenada no banco de dados *online*.
Método *equals*	Gravar de forma ordenada no banco de dados *online* a pontuação recebida através do parâmetro *score*.	Pontuações negativas não afetarão seu comportamento.

Tabela 4.3: Contrato entre o construtor da classe *Timer* e seu chamador.

Agora temos que traduzir o contrato da Tabela 4.3 para a linguagem JML e então podermos utilizar a informação para aumentar a expressividade da interface emergente vista na Seção 3.3. O resultado da tradução do contrato para a linguagem JML pode ser visto no Código 4.3.

```
1 //@ requires minutes > 0;
2 public Timer(int minutes) {
3   // ...
4 }
```

Código 4.3: Construtor da classe *Timer* com um contrato escrito em JML.

Então, o resultado da integração do contrato JML proposto à interface emergente mostrada na Seção 3.3 é o seguinte:

Feature TIMER requires minutes > 0.

Esta nova interface emergente auxilia a detecção do problema pois chama a atenção do desenvolvedor para a variável que ele está realizando a manutenção, fazendo com que ele tome cuidados extras. Sem esta informação o desenvolvedor depende totalmente da sua atenção e apesar da informação ser óbvia para alguns desenvolvedores - um cronômetro regressivo com valor inicial nulo não faz sentido - até mesmo os mais atentos dos desenvolvedores estão sujeitos a erros.

Observe que, assim como nos cenários anteriorores, o desenvolvedor não precisa acessar outras classes e métodos, neste caso as classes *Player* e *Timer*, para descobrir novas informações sobre a dependência entre a *feature TIMER* e a variável *minutes*. Sendo assim, mesmo apresentando informações sobre dependências dentro da *feature* de interesse, a nova interface emergente (EIJ) ainda assim possui uma melhoria na sua expressividade.

Capítulo 5

Avaliação

No Capítulo 4, propusemos uma forma de melhorar a expressividade das interfaces emergentes utilizando JML. Este capítulo trata de avaliar a aplicação de contratos JML a interfaces emergentes com o objetivo de medir e avaliar se existe melhora na expressividade das interfaces emergentes.

Para avaliar a solução executamos um experimento onde participantes analisaram cenários que motivam a utilização das interfaces emergentes utilizando interfaces emergentes puras e interfaces emergentes com JML. Os cenários de análise foram baseados nos exemplos PDF e *Best Lap* ambos mostrados nos Capítulos 3 e 4 deste trabalho.

Os dados das análises destes participantes foram utilizados para medir e avaliar a expressividade da nova interface emergente (EIJ). As definições, regras e resultados deste experimento estão detalhados nas seções deste capítulo.

5.1 Definição

Começamos com as definições fundamentais deste experimento. Se não definirmos corretamente os parâmetros deste experimento ele pode perder sua validade e fugir do real objetivo traçado, portanto, esta fase do experimento é de extrema importância. O objetivo desta seção é determinar os parâmetros de análise deste experimento de acordo com a abordagem *Goal Question Metrics* (GQM) proposta por Basili e Rombach [2].

O modelo GQM é baseado na suposição de que uma organização interessada em mensuração deve primeiro especificar os seus objetivos para si mesmo e para seus projetos, traçar esses objetivos até os dados projetados para definir esses objetivos operacionalmente, e finalmente prover um framework para interpretar os dados em relação aos objetivos traçados inicialmente. A aplicação do GQM resulta na especificação de um sistema de mensuração focado em um conjunto particular de questões, e num conjunto de regras para interpretar os dados mensurados. O resultado final é um modelo de mensuração em três níveis [2]:

- Nível conceitual (Objetivo - *GOAL*): Um objetivo é definido para um objeto por diversas

razões, em respeito a diversos modelos de qualidade, a partir de diversos pontos de vista e relativo a um ambiente em particular.

- Nível operacional (Pergunta - *QUESTION*): Um conjunto de perguntas é usado para definir modelos do objeto em estudo e então focar no objeto para caracterizar a avaliação ou realização de um objetivo específico.
- Nível quantitativo (Métrica - *METRIC*): Um conjunto de dados, definidos em métricas, é associado a cada pergunta com o objetivo de responder elas de uma forma mensurável.

As definições dos parâmetros GQM, ou seja, objetivo, perguntas e métricas, estão explicadas nas subseções seguintes.

5.1.1 *Goal* (Objetivo)

G1. O objetivo desta avaliação é analisar as interfaces emergentes com JML em relação à sua expressividade, tempo e nível de entendimento, na visão do desenvolvedor e no contexto de projetos de linhas de produtos de software.

5.1.2 *Questions* (Perguntas)

Q1. As interfaces emergentes com JML ajudam a evitar erros de manutenção de código nas linhas de produtos de software?

Q2. As interfaces emergentes com JML ajudam a diminuir o tempo de entendimento do código de uma linha de produtos de software?

5.1.3 *Metrics* (Métricas)

M1. Nível de entendimento (NE): O nível de entendimento é definido conforme equação a seguir:

$$NE(t) = \frac{A(t)}{n(t)} * 100$$

Onde:

$A(t)$ é o número de participantes que acertaram o questionamento de um cenário (c) qualquer analisado com a técnica (t).

$n(t)$ é o número de participantes que utilizaram a técnica (t).

Ou seja, o nível de entendimento é definido como a porcentagem de participantes que acertaram o questionamento de um cenário analisado com a técnica (t). Esta métrica se preocupa apenas com a técnica definida para o participante desconsiderando o cenário analisado.

M2. Tempo de análise (TA): Esta métrica é definida como a média do tempo de análise de todos os participantes (p) para uma técnica (t):

$$TA(t) = \frac{\sum T(p,t)}{n(t)} \tag{5.1}$$

Onde:

$T(p,t)$ é o tempo de análise, em segundos, do participante (p) para a técnica (t) independente do cenário analisado.

$n(t)$ é o número de participantes do experimento que utilizaram a técnica (t).

O tempo de análise do participante é definido como o módulo da diferença entre o momento inicial de análise da configuração (cenário + técnica) e o momento final em que o participante confirma sua resposta. Assim como a métrica M1, esta métrica desconsidera o cenário analisado e leva em conta apenas a técnica utilizada.

5.2 Planejamento

Na seção anterior definimos as bases que motivam o experimento a ser conduzido. Nesta seção definir emos como o experimento deverá ser conduzido, ou seja, definiremos o planejamento do experimento. Os planejamentos devem ser cuidadosamente traçados nesta fase e fielmente seguidos durante a fase de execução sob pena de causar ruídos aos dados ou até mesmo total inutilização dos mesmos. Como todo o planejamento foi realizado antes da execução do experimento, durante esta seção utilizaremos o tempo verbal futuro.

5.2.1 Contexto

O objetivo desta avaliação é analisar as interfaces emergentes com JML em relação à sua expressividade (tempo e nível de entendimento). A avaliação deverá ser executada em uma sala isolada na Universidade Federal de Alagoas com um computador com acesso à internet.

Os participantes deverão analisar sem limite de tempo e responder questionamentos em dois cenários que motivam a utilização das interfaces emergentes. Os participantes estarão isolados durante a execução do experimento a fim de diminuir possíveis ruídos na medição do tempo.

5.2.2 Treinamento

Os participantes não receberão treinamento pois os cenários estarão preparados para que estes necessitem somente um bom conhecimento em programação orientada à objetos e Java. Todos os cuidados deverão ser tomados para não envolver outros conceitos na explicação ou questionamento do cenário.

5.2.3 Hipótese Nula

Neste experimento, a hipótese nula determinará que não existe diferença de expressividade (tempo e nível de entendimento) entre as interfaces emergentes puras e as interfaces emergentes com JML. Com isto e com as métricas definidas na Seção 5.1.3, as seguintes hipóteses nulas foram definidas:

$$H_0 1\text{: } \mu_{NE_{(EI)}} = \mu_{NE_{(EIJ)}}$$
$$H_0 2\text{: } \mu_{TA_{(EI)}} = \mu_{TA_{(EIJ)}}$$

5.2.4 Hipótese Alternativa

Neste experimento, a hipótese alternativa determinará que as interfaces emergentes com JML possuem uma maior expressividade (tempo e nível de entendimento) em comparação com as interfaces emergentes puras. Com base nisto e com as métricas definidas na Seção 5.1.3, estas são as hipóteses alternativas:

$$H_1 1\text{: } \mu_{NE_{(EI)}} < \mu_{NE_{(EIJ)}}$$
$$H_1 2\text{: } \mu_{TA_{(EI)}} > \mu_{TA_{(EIJ)}}$$

5.2.5 Seleção dos Participantes

O ideal para esta avaliação seria que os participantes tivessem experiência em toda a fundamentação teórica abordada por este trabalho, porém, não é fácil encontrar pessoas com tal nível de conhecimento. Como forma de expandir as possibilidades de participantes e simplificar a avaliação, os cenários serão preparados para envolver apenas conceitos de orientação a objetos utilizando Java. A seleção dos participantes será realizada de forma aleatória.

5.2.6 Instrumentação

Os participantes deverão analisar dois cenários: um envolvendo o conceito de interfaces emergentes e outro envolvendo o conceito de interfaces emergentes com JML; não necessariamente nesta ordem.

Os cenários serão baseados nos exemplos PDF (Seções 3.1 e 4.1) e *Best Lap* (Seções 3.2 e 4.2) vistos anteriormente neste trabalho. Eles serão desenhados na linguagem HTML e serão divididos em duas partes:

- Introdução: Nesta parte é criada uma situação fictícia aonde um desenvolvedor recebe uma determinada tarefa de manutenção. A introdução deverá explicar qual o resultado esperado com a manutenção repassada ao desenvolvedor.

- Questionamento: Nesta parte é mostrado o código original do cenário e o código alterado com o resultado da manutenção realizada pelo desenvolvedor. O participante deverá analisar a manutenção e responder se manteria as alterações exatamente como elas estão apresentadas. Logo após confirmar sua resposta, o participante deverá explicar o porquê da mesma.

A análise dos cenários deverá ser executada em um ambiente fechado e em um computador com acesso à web. Cada participante receberá um código de acesso único. Suas respostas e tempo de análise serão enviadas automaticamente via e-mail.

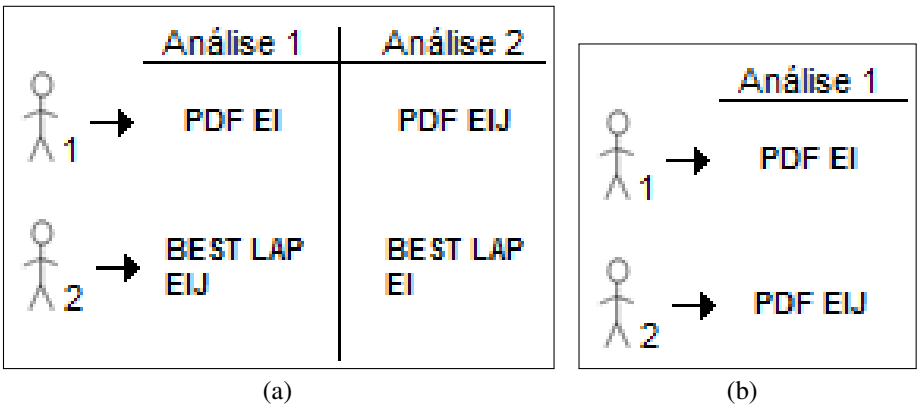
Como forma de minimizar possíveis ruídos no experimento, os cenários e técnicas serão sorteados para cada participante utilizando quadrados latinos de ordem 2, conforme explicação a seguir.

5.2.7 Quadrados Latinos

Neste experimento detectamos os seguintes fatores como sendo possíveis ruídos:

- Se um participante realizar duas análises em um mesmo cenário, cada vez utilizando uma técnica diferente, conforme exemplo mostrado na Figura 5.1(a), estaremos favorecendo a técnica utilizada na segunda análise pois o participante já entendeu o funcionamento do cenário;

- Se um participante realizar apenas uma analise, conforme exemplo na Figura 5.1(b), estaremos favorecendo a técnica utilizada para tal análise pois alguns participantes podem possuir uma maior facilidade em entender o problema apresentado no cenário.



(a) (b)

Figura 5.1: Exemplos de distribuição de cenário e técnica com introdução de ruídos

A técnica ideal para distrubuir os cenários e técnicas de forma a eliminar os possíveis ruídos encontrados é a técnica dos Quadrados Latinos. As duplas que farão parte de cada um dos quadrados latinos serão geradas conforme ordem de convocação respeitando as seguintes regras:

- Participantes de número **ímpar**: Será sorteado a primeira configuração, ou seja, cenário e técnica. Como o experimento contempla apenas dois cenários e duas técnicas, automaticamente a segunda configuração deste participante será o que restou;
- Participantes de número **par**: Com base no participante imediatamente anterior (P_{n-1}), o participante de número par realizará as análises invertendo a ordem das técnicas e mantendo a ordem dos cenários.

Na Figura 5.2 temos um exemplo de um quadrado latino aonde para o participante P_1 foi sorteado o cenário c_1 com EIJ, assim, o mesmo deverá analisar o cenário c_2 utilizando EI. A mesma figura mostra também que automaticamente foi atribuído ao participante P_2 a ordem invertida das técnicas mantendo a ordem dos cenários.

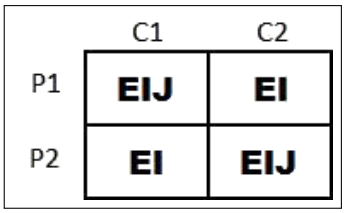


Figura 5.2: Configuração de um quadrado latino.

5.3 Projeto piloto

Previamente à execução do experimento rodamos um projeto piloto com dois participantes para descobrir erros e consertá-los antes da real execução. Com este projeto piloto fizemos as seguintes melhorias em ambos os cenários:

- **Remoção de informações tendenciosas:** Tais informações foram removidas na tela de introdução dos cenários pois consideramos que elas induziam o participante a responder corretamente a pergunta;
- **Menos ênfase na informação de que o código "compila com sucesso":** Retiramos o negrito da referida informação na tela de questionamento dos cenários pois notamos que o excesso de ênfase na informação induziu os participantes do projeto piloto ao erro;
- **Remoção das explicações de código:** Estas explicações, que estavam na tela de questionamento, foram removidas pois elas enfatizavam muito a localização do problema do cenário;
- **Mudança na cor da interface emergente de vermelho para azul:** Inicialmente colocamos as interfaces emergentes em vermelho para enfatizar a mensagem, porém, um dos

participantes do piloto informou que com esta cor a interface aparentava ser uma mensagem de erro. Com base nisto modificamos a cor para azul removendo assim a aparência de mensagem de erro e sem perder a ênfase desejada.

5.4 Execução

O experimento foi rodado com dez participantes conforme a instrumentação especificada na Seção 5.2.6. Nenhum dos participantes tiveram acesso a outros materiais e todos eles estiveram isolados durante toda a execução do experimento.

5.4.1 Cenários

Os cenários de avaliação foram desenvolvidos com base nos exemplos PDF (Seções 3.1 e 4.1) e Best Lap (Seções 3.2 e 4.2). Ambos foram divididos em duas partes:

- **Parte 1 - Introdução:** Esta parte introduz o sistema em questão junto com uma modificação no código com objetivo de melhorar o sistema em um cenário imaginário. Esta parte é idêntica para ambas as técnicas (EI e EIJ). As Figuras 5.3 e 5.4 mostram o resultado final da primeira parte de cada um dos cenários;

- **Parte 2 - Questionamento:** Nesta parte o participante recebe o código da versão inicial do sistema e o código da versão com a melhoria, proposta na introdução do problema, implementada. Logo em seguida é mostrada a interface emergente de acordo com a técnica sorteada para o praticipante (*EI* ou *EIJ*).

  Ao final desta parte o participante foi questionado se manteria as modificações implementadas e também o porque da sua escolha. O participante explicou sua resposta inicial independente de qual tenha sido ela.

  As Figuras 5.5 e 5.6 mostram o resultado final da segunda parte de cada um dos cenários na versão EIJ. Para a versão EI existe apenas uma diferença nas mensagens das interfaces emergentes que não contempla informações sobre os contratos JML inseridos no código.

5.4.2 Participantes

Os participantes deste experimento foram escolhidos aleatoriamente obedecendo aos critérios estabelecidos na Seção 5.2.5. Todos são alunos ou professores nas instituições de ensino UFAL ou IFAL e possuem um mínimo de experiência em programação Java e programação orientada a objetos.

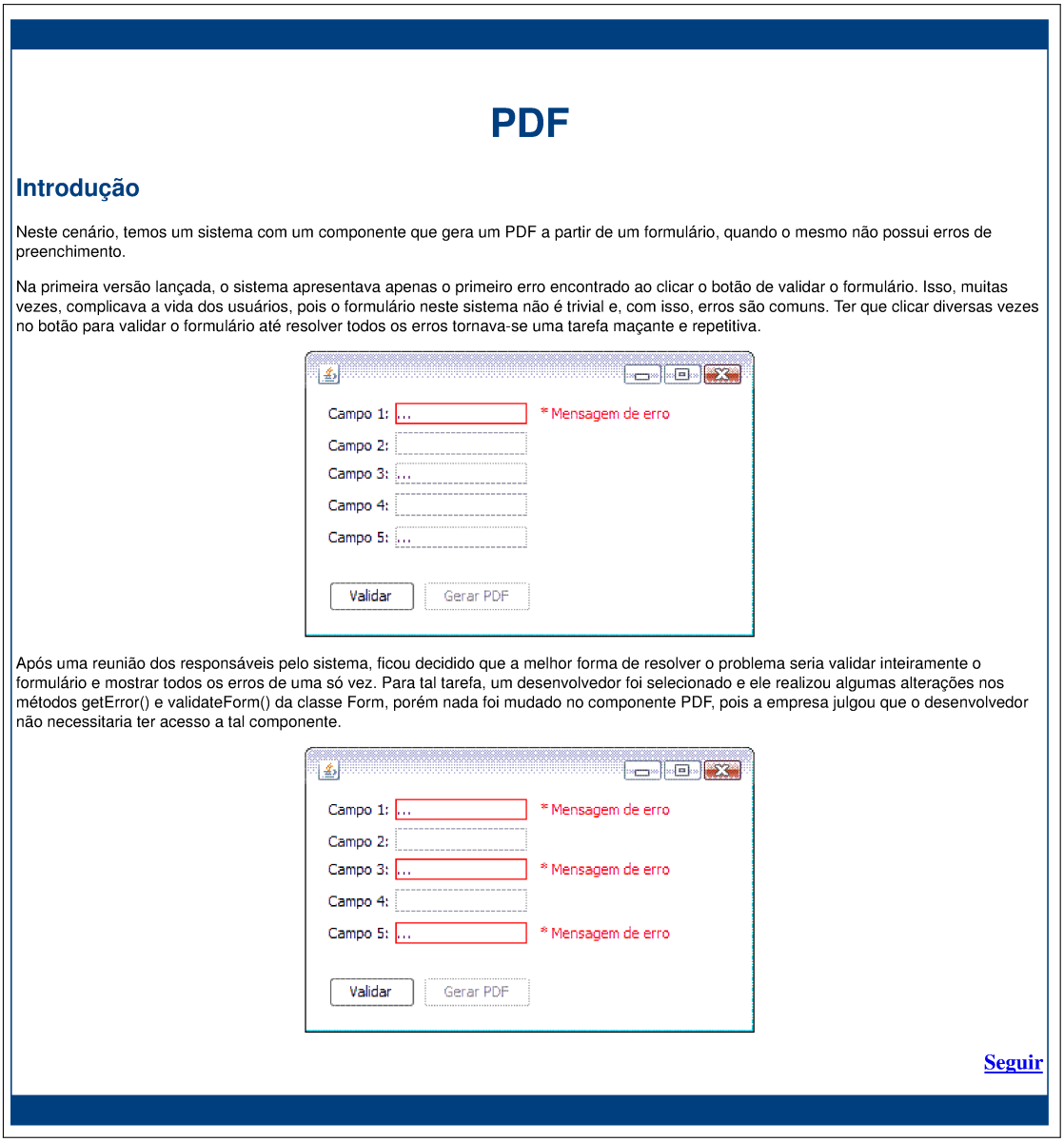


Figura 5.3: Introdução do cenário PDF

5.4.3 Sorteio dos Quadrados

Os participantes foram inseridos em quadrados latinos conforme convocação para realização do experimento. Sempre que convocado um participante de número ímpar (ex: primeiro participante, terceiro participante) foi gerado um novo quadrado latino. Os participantes de número par (ex: segundo participante, quarto participante) foram encaixados no quadrado latino já disponível. Ao todo foram gerados cinco quadrados latinos com as configurações mostradas na Figura 5.7.

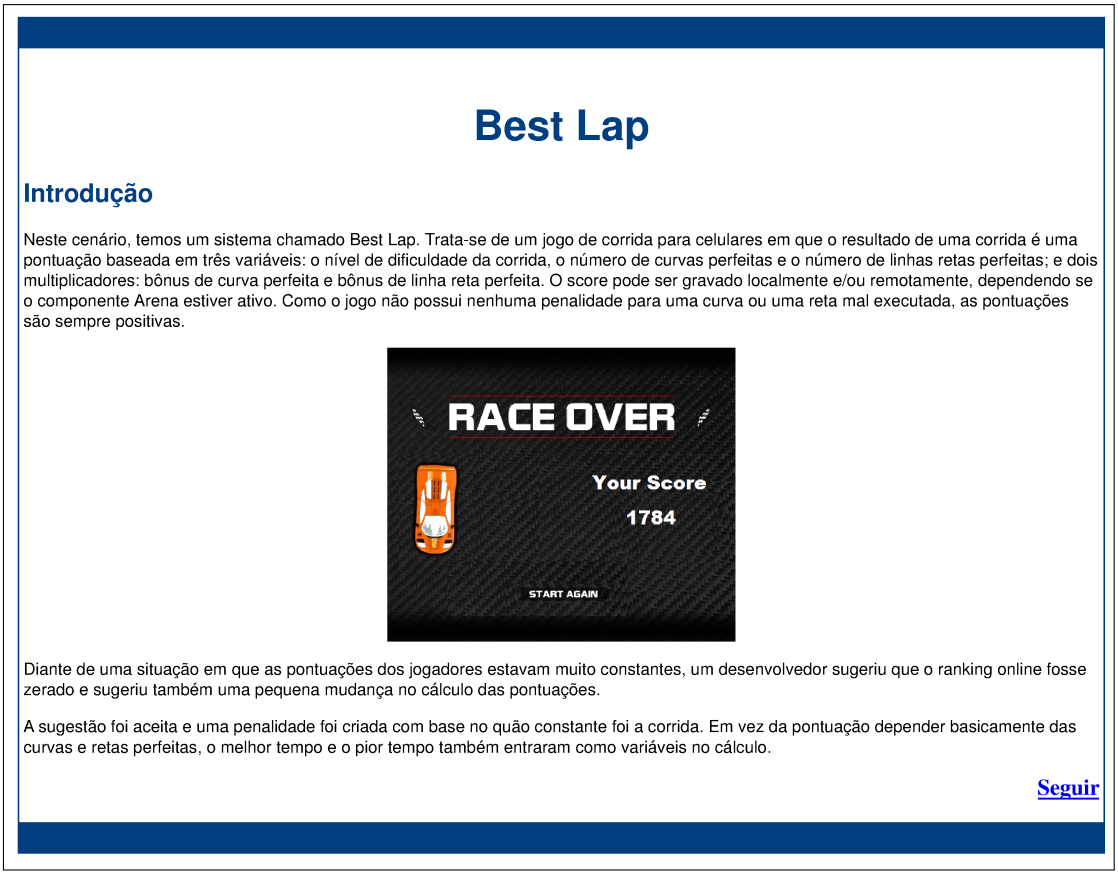


Figura 5.4: Introdução do cenário Best Lap

5.5 Análise e interpretação

Nesta seção veremos os resultados obtidos com a execução deste experimento. Esta seção está dividida em duas partes, de acordo com as métricas definidas na Seção 5.1.3.

5.5.1 Análise das Respostas

Como dito anteriormente, os cenários deste experimento foram baseados nas Seções 3.1 e 4.1 (Cenário PDF) e Seções 3.2 e 4.2 (Cenário *Best Lap*) deste trabalho, ou seja, em ambos os casos os desenvolvedores fictícios introduziram erros durante suas respectivas manutenções. Sendo assim, quando os participantes foram perguntados, na segunda tela de cada cenário, se manteriam as modificações executadas pelo desenvolvedor, os mesmos deveriam responder *"não"* para que sua resposta fosse considerada correta. Assim, conforme tabela de respostas (Tabela 5.1), chegamos aos seguintes resultados para a métrica NE:

$$NE(EI) = 40\%$$
$$NE(EIJ) = 70\%$$

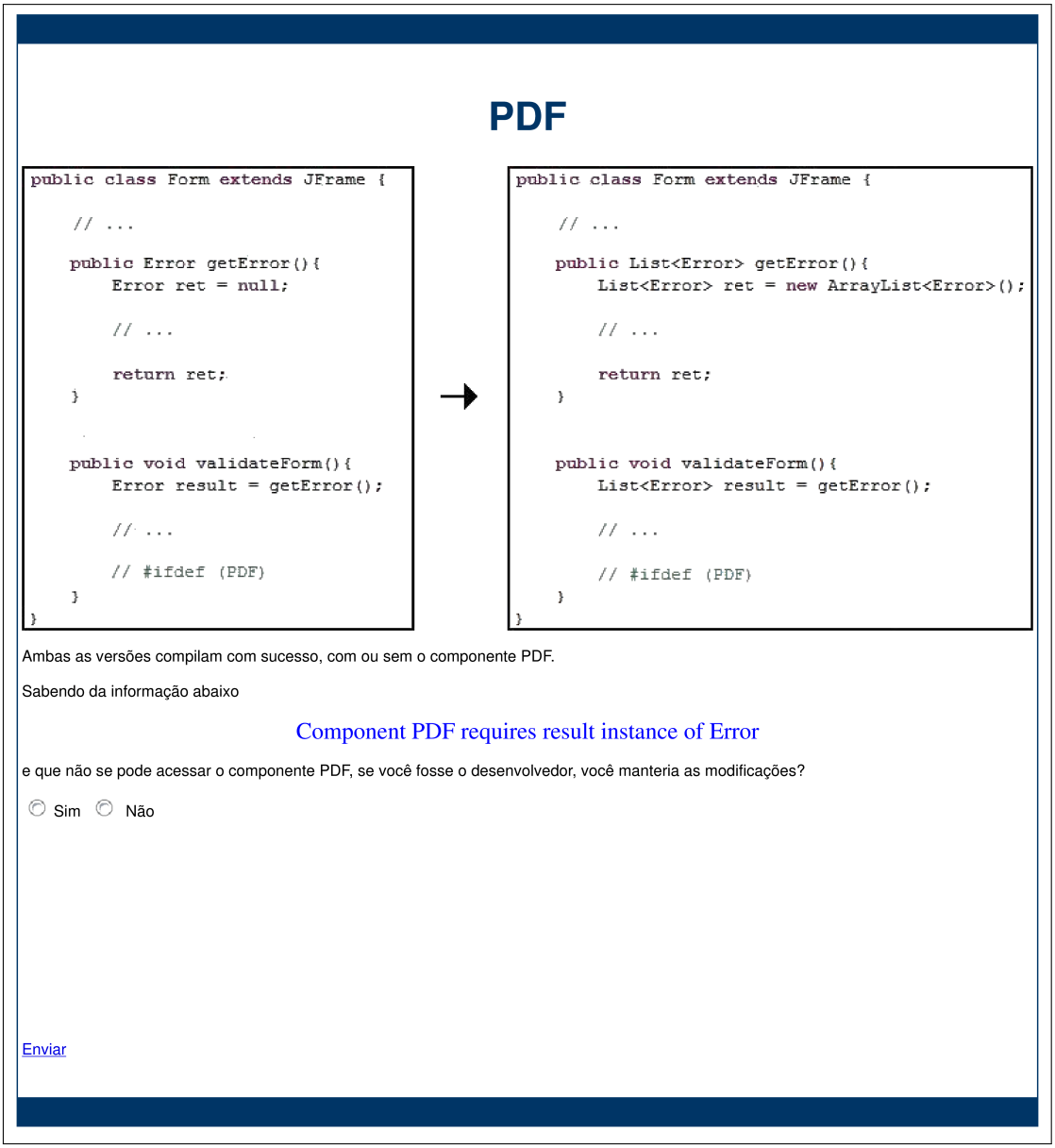


Figura 5.5: Questionamento do cenário PDF (Versão EIJ).

Os resultados da métrica (NE) apontam inicialmente para a rejeição de sua respectiva hipótese nula ($H_0 1$) e em favor de sua respectiva hipótese alternativa ($H_1 1$). Ainda assim, precisamos analisar cuidadosamente os dados.

Sabemos que a métrica NE ignora o cenário analisado e utiliza apenas a técnica usada pelos participantes e, consequentemente, a hipótese nula $H_0 1$ avalia apenas a técnica independente do cenário utilizado. Todavia, podemos analisar os dados por uma perspectiva diferente criando uma nova métrica NE que desta vez levará em consideração também o cenário utilizado. Com esta nova abordagem obtemos os seguintes resultados:

$$NE(PDF, EI) = 60\%$$
$$NE(PDF, EIJ) = 80\%$$

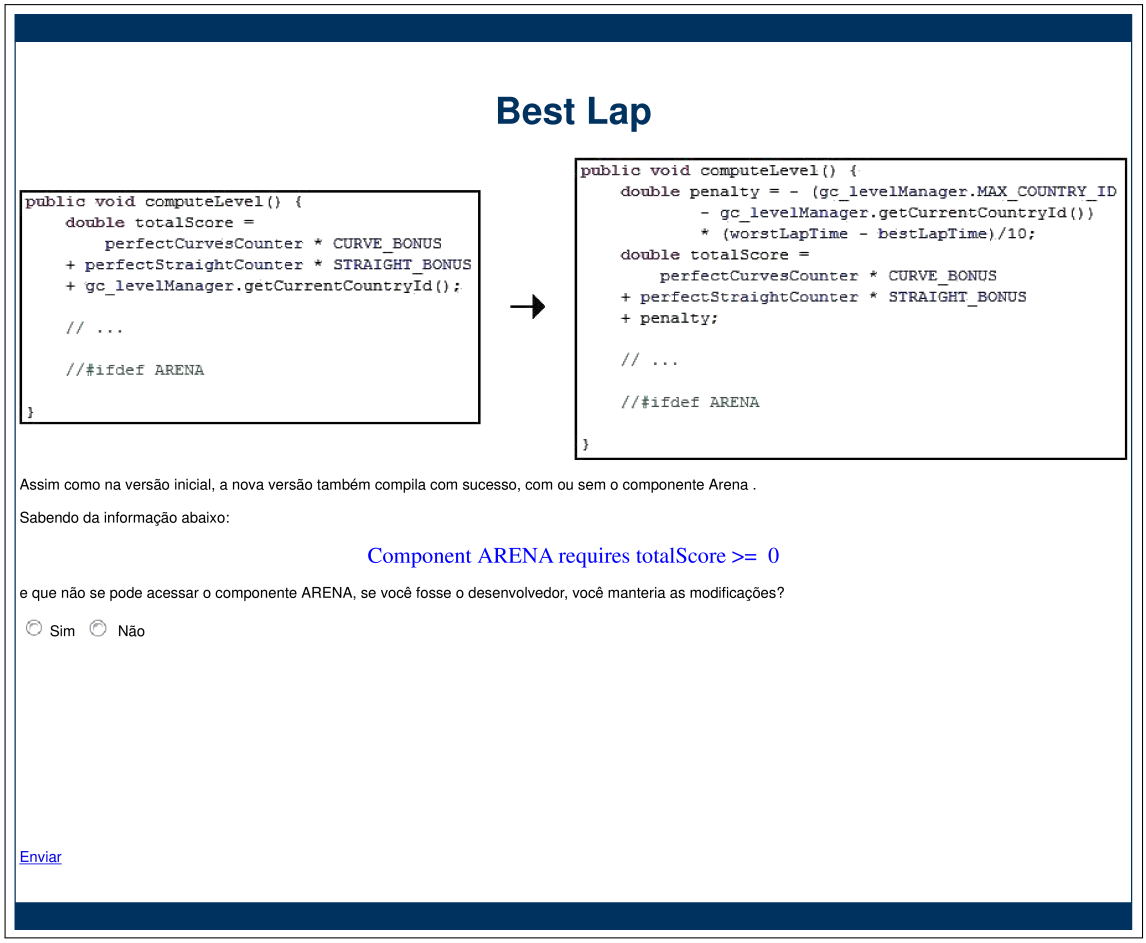


Figura 5.6: Questionamento do cenário Best Lap (Versão EIJ).

$$NE(BESTLAP, EI) = 20\%$$
$$NE(BESTLAP, EIJ) = 60\%$$

Outra abordagem seria confrontar o número de acertos para a técnica EI com o número de acertos para a técnica EIJ. Neste caso teriamos em um total de 11 acertos, 7 para a técnica EIJ, o que representa 63.64% do total de acertos, contra 4 acertos para a técnica EI, representando 36.36% do total de acertos.

Poderíamos realizar a mesma análise anterior porém levando em consideração além da técnica, o cenário. Ainda assim, perceberíamos um aumento no número de acertos em vantagem da técnica EIJ para ambos os cenários.

Concluímos então que independente da abordagem utilizada existe uma melhora no nível de entendimento dos cenários quando os participantes utilizam interfaces emergentes com JML (EIJ). Ainda assim, rejeitar a hipótese nula $H_0 1$ e/ou afirmar a hipótese alternativa $H_1 1$ pode ser um tanto precipitado devido ao baixo número de participantes.

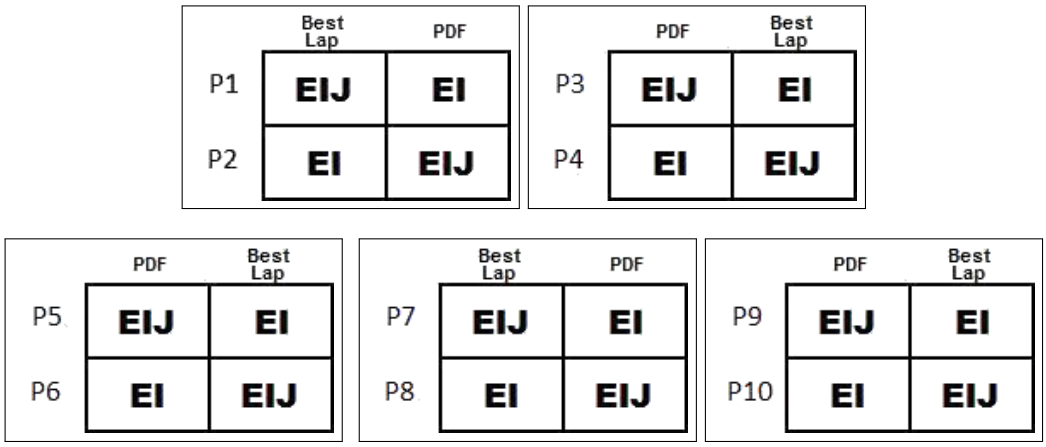


Figura 5.7: Quadrados Latinos sorteados durante o experimento.

Técnica EI			Técnica EIJ		
Participante	Cenário	Resposta	Participante	Cenário	Resposta
1	PDF	NÃO	1	BEST LAP	NÃO
2	BEST LAP	SIM	2	PDF	SIM
3	BEST LAP	SIM	3	PDF	NÃO
4	PDF	SIM	4	BEST LAP	SIM
5	BEST LAP	NÃO	5	PDF	NÃO
6	PDF	NÃO	6	BEST LAP	NÃO
7	PDF	SIM	7	BEST LAP	SIM
8	BEST LAP	SIM	8	PDF	NÃO
9	BEST LAP	SIM	9	PDF	NÃO
10	PDF	NÃO	10	BEST LAP	NÃO

Tabela 5.1: Resposta dos participantes.

5.5.2 Análise dos Tempos

Na Tabela 5.2 temos os tempos de todos os participantes do experimento. Com esses dados chegamos aos seguintes resultados para a métrica TA:

$$TA(EI) = 145.5$$
$$TA(EIJ) = 143.5$$

Estes resultados obtidos para a métrica TA apontam inicialmente para a rejeição da hipótese nula $H_0 2$ e também apontam para uma afirmação da hipótese alternativa $H_1 2$, porém, a proximidade dos valores é muito grande e o desvio padrão dos dados é muito alto. Além disto, observe que nesta análise não estamos levando em consideração se o participante acertou ou errou a resposta do cenário, ou seja, estamos realizando uma análise utlizando dados incompatíveis. Em resumo, não podemos chegar a nenhuma conclusão com esta análise inicial.

Técnica EI			Técnica EIJ		
Participante	Cenário	Tempo	Participante	Cenário	Tempo
1	PDF	134	1	BEST LAP	119
2	BEST LAP	193	2	PDF	43
3	BEST LAP	51	3	PDF	103
4	PDF	121	4	BEST LAP	91
5	BEST LAP	114	5	PDF	126
6	PDF	327	6	BEST LAP	307
7	PDF	36	7	BEST LAP	121
8	BEST LAP	201	8	PDF	106
9	BEST LAP	59	9	PDF	197
10	PDF	219	10	BEST LAP	219

Tabela 5.2: Tempo dos participantes em segundos.

5.5.2.1 Participantes com respostas corretas

Embora o balanceamento dos dados seja desejável para simplificar e fortalecer a análise estatística dos dados, isto não é obrigátorio [23]. Portanto, para uma análise mais coerente dos tempos, removemos os tempos dos participantes que erraram o questionamento, mesmo que o resultado tenha gerado uma tabela não-balanceada.

Para melhorar a visualização dos dados da Tabela 5.3 e facilitar a análise, os mesmos foram plotados em um gráfico do tipo *boxplot* mostrando a mediana, os quartis e os extremos dos dados e também foram plotados em um gráfico do tipo *beanplot* para mostrar a concentração dos dados (Figura 5.8).

Então, temos agora as seguintes médias:

$$\bar{x}_{EI} = 198.5$$
$$\bar{x}_{EIJ} = 168.1$$

Técnica EI		
Participante	Cenário	Tempo
1	PDF	134
5	BEST LAP	114
6	PDF	327
10	PDF	219

Técnica EIJ		
Participante	Cenário	Tempo
1	BEST LAP	119
3	PDF	103
5	PDF	126
6	BEST LAP	307
8	PDF	106
9	PDF	197
10	BEST LAP	219

Tabela 5.3: Tempo dos participantes que acertaram a resposta do questionário.

Nesta situação as médias também apontam para uma melhoria em favor da técnica EIJ e, com base no gráfico da Figura 5.8, notamos também que houve uma diminuição na mediana

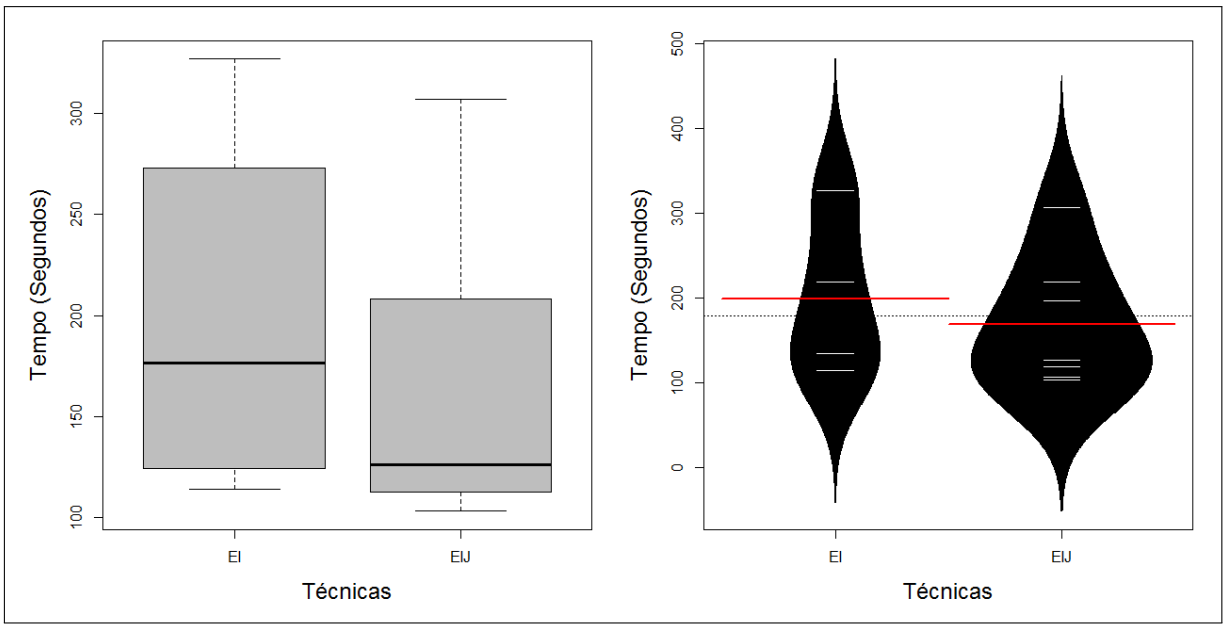


Figura 5.8: Gráficos *boxplot* e *beanplot* com o tempo de análise.

dos dados. Na realidade notamos melhoria em favor da técnica EIJ nos mais diversos aspectos dos tempos: valor mínimo e máximo, valores dos quartis, variância, desvio padrão etc.

Um ponto importante a ser observado no gráfico *boxplot* da Figura 5.8 é a ampla distância entre o valor máximo para a técnica EIJ e o terceiro quartil para a mesma técnica. Esta distância é um indício de que existe pelo menos um *outlier* nos tempos da técnica EIJ.

Um método simples para detectar um *outlier* é o método do intervalo inter-quartil. Com $k = 1$ notamos que um valor x será considerado *outlier* se estiver fora do intervalo $[Q_1 - k * (Q_3 - Q_1), Q_3 + k * (Q_3 - Q_1)] = [17, 303.5]$. Portanto, uma última análise será realizada na próxima seção, desta vez sem o participante 6 para a técnica EIJ que realizou a análise em 307 segundos.

5.5.2.2 Análise sem *outliers*

Como estamos realizando uma análise não-balanceada, retiramos apenas o tempo do participante 6 para a técnica EIJ, mantendo o tempo do mesmo para a técnica EI. Então com base na nova tabela 5.4 sem o *outlier* para a técnica EIJ, temos as seguintes médias:

$$\bar{x}_{EI} = 198.5$$
$$\bar{x}_{EIJ} = 145.0$$

As melhorias são ainda mais notáveis ao remover o *outlier* para a técnica EIJ pois os valores de tempo para o participante 6 são muito altos em ambas as técnicas, e, ao remover o tempo

Técnica EI		
Participante	Cenário	Tempo
1	PDF	134
5	BEST LAP	114
6	PDF	327
10	PDF	219

Técnica EIJ		
Participante	Cenário	Tempo
1	BEST LAP	119
3	PDF	103
5	PDF	126
8	PDF	106
9	PDF	197
10	BEST LAP	219

Tabela 5.4: Tempo dos participantes que acertaram a resposta do questionário e sem *outliers*.

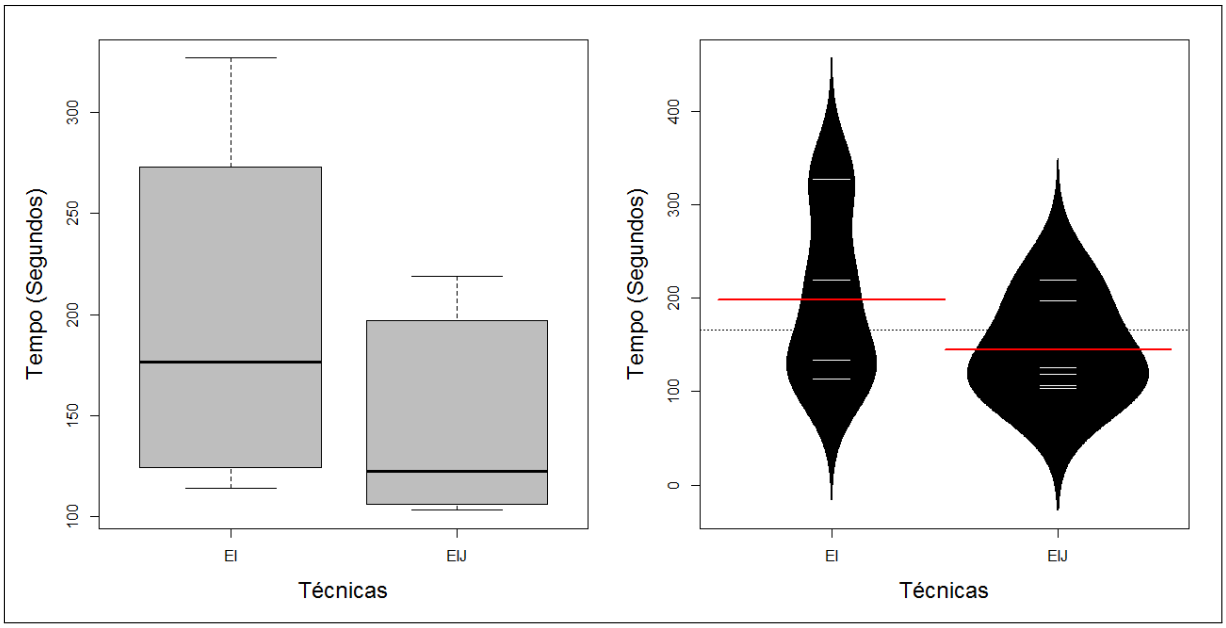


Figura 5.9: Gráficos *boxplot* e *beanplot* sem *outliers*.

apenas para a técnica EIJ, esta melhoria fica mais evidente. Os dados desta nova análise podem ser melhor visualizados nos gráficos do tipo *boxplot* e *beanplot* da Figura 5.9.

É importante salientar que mesmo se removessemos o tempo do participante 6 também para a técnica EI, a melhora no tempo médio ainda existiria. Isto porque o tempo do referido participante para ambos as técnicas foram muito próximos entre si (aumento de aproximadamente 6% em relação ao menor valor).

5.5.3 Confiança de uma regra

Durante a análise dos resultados deste experimento notamos um fato interessante que pode ajudar a perceber a melhora na expressividade, no aspecto do nível de entendimento, das interfaces emergentes com JML. Estamos falando das regras $A_{EI} \Rightarrow A_{EIJ}$ e $A_{EIJ} \Rightarrow A_{EI}$ e das suas respectivas confianças.

Antes de detalhar o significado destas regras, é importante entender o significado e o cálculo da confiança de uma regra. Esta métrica é utilizada para avaliar uma regra de associação e

representa a certeza de tal regra. O cálculo desta métrica utiliza o conceito de suporte de um conjunto que é definido como a proporção de transações que contém um determinado conjunto. O suporte de um conjunto também pode ser aplicado para avaliar uma regra e, neste caso, o suporte de uma regra é definido como a proporção de transações que contém todos os elementos participantes da regra. O suporte de uma regra representa a utilidade da mesma [6]. O cálculo das métricas suporte e confiança podem ser vistos nas equações 5.2 e 5.3.

$$supp(X \Rightarrow Y) = P(X \cup Y) \tag{5.2}$$

$$conf(X \Rightarrow Y) = \frac{supp(X \cup Y)}{supp(X)} \tag{5.3}$$

Para entender melhor as métricas, imagine que uma empresa revendedora de eletrônicos percebeu uma grande quantidade de consumidores comprando computador e impressora juntos. O desenvolvedor responsável pelo sistema da empresa criou um algoritmo para calcular o suporte e a confiança da regra $\{computador\} \Rightarrow \{impressora\}$. O algoritmo resultou em um suporte de 10% e confiança de 90%. Este suporte encontrado mostra que computadores e impressoras são comprados juntos em 10% de todas as transações análisadas. Quanto a confiança encontrada, significa que 80% dos consumidores que compraram um computador, compraram também uma impressora. Ou seja, a regra percebida pela empresa pode ser considerada forte.

Voltando as regras encontradas neste experimento, a primeira regra, $A_{EI} \Rightarrow A_{EIJ}$, diz que se um participante acertou a análise utilizando a técnica *EI*, o mesmo também acertou a análise utilizando a técnica *EIJ*. Para esta regra temos uma confiança de 100%, ou seja, todos os participantes que acertaram a resposta do cenário analisado utilizando a técnica EI, também acertaram a resposta do cenário analisado utilizando EIJ (Ver demonstração na equação 5.4).

$$conf(A_{EI} \Rightarrow A_{EIJ}) = \frac{supp(A_{EI} \cup A_{EIJ})}{supp(A_{EI})} = \frac{0.4}{0.4} = 1 \tag{5.4}$$

Na segunda regra temos o inverso da primeira, ou seja, se um participante acertou a análise utilizando a técnica *EIJ*, o mesmo também acertou a análise utilizando a técnica *EI* ($A_{EIJ} \Rightarrow A_{EI}$). Neste caso, percebemos que esta regra possui a confiança de aproximadamente 57% (Ver demonstração na equação 5.5). Isto significa que houveram participantes que somente acertaram a resposta do cenário analisado utilizando a ténica EIJ, ou seja, acertar a análise com a técnica EIJ não significa acertar a análise com a técnica EI.

$$conf(A_{EIJ} \Rightarrow A_{EI}) = \frac{supp(A_{EIJ} \cup A_{EI})}{supp(A_{EIJ})} = \frac{0.4}{0.7} = 0.5714 \tag{5.5}$$

Quando combinamos as informações fornecidas pelos participantes com as informações das confianças destas regras, concluímos que os participantes que acertaram a análise utilizando a técnica EI, e consequentemente com a técnica EIJ, são provavelmente muito cautelosos e

atentos. As figuras 5.10 e 5.11 apresentam algumas das respostas dos participantes que apoiam este fato.

Pergunta: "Se você fosse o desenvolvedor, você manteria as modificações?" (Em referência às modificações realizadas no respectivo cenário pelo desenvolvedor fictício.)

Resposta:

"Não. Como o componente pdf usa a variável result, deve haver um contrato no tipo do parâmetro (Error). Provavelmente, essa mudança afetaria o funcionamento do componente pdf."

(Participante 1 - Cenário PDF / Técnica EI)

Figura 5.10: Resposta do participante 1

Pergunta: "Se você fosse o desenvolvedor, você manteria as modificações?" (Em referência às modificações realizadas no respectivo cenário pelo desenvolvedor fictício.)

Resposta:

"Não. Pois como o componente PDF nao pode ser acessado,(componente sem acesso ao codigo fonte) e ele está esperando uma variável que não é do tipo List e portanto no meu ponto de vista deve compilar, mas em tempo de execuçao deve gerar um erro, onde o compilador não é capaz de detectar (type mismach)."

(Participante 6 - Cenário PDF / Técnica EI)

Figura 5.11: Resposta do participante 6

Para os participantes que acertaram apenas a análise utilizando a técnica EIJ, percebemos que os mesmos tiveram uma maior facilidade em interpretar os cenários utilizando a técnica EIJ e perceber o erro introduzido pelo desenvolvedor fictício. Este fato é sugere que pode existir uma melhora na expressividade, quanto ao nível de entendimento, das interfaces emergentes com JML. Na figura 5.12 temos algumas das respostas dos participantes que apoiam este fato.

5.6 Conclusão do experimento

Os resultados mostrados na seção anterior sugerem uma melhoria na expressividade das interfaces emergentes com JML. Contudo, não temos como rejeitar nenhuma das duas hipóteses nulas e nem confirmar nenhuma das duas hipóteses alternativas devido ao baixo número de participantes.

O baixo número de participantes também impediu uma análise mais aprofundada dos tempos dos participantes utilizando a técnica de Análise de Variância (ANOVA) visto que ao re-

Pergunta: "Se você fosse o desenvolvedor, você manteria as modificações?" (Em referência às modificações realizadas no respectivo cenário pelo desenvolvedor fictício.)

Resposta 1:

"Sim. Manteria a modificação, pois a variavel totalScore foi mantida, inclusive o seu tipo, modificando apenas a regra para cálcula-la, o que não irá afetar o componente Arena."

Resposta 2:

"Não. Pela informação, meu entendimento é que há um requisito do componente PDF que exista uma variável result do tipo Error. Dessa forma uma instancia de result do tipo List acarretaria uma erro na hora de usar o componente PDF."

(Participante 3 - Respectivamente: Cenário BEST LAP / Técnica EI & Cenário PDF / Técnica EIJ)

Figura 5.12: Respostas do participante 3

mover os tempos dos participantes que erraram a resposta da análise não foi mais possível combinar os demais participantes em novos quadrados latinos. E mesmo com os todos os dados mostrados na Tabela 5.2 não seria muito significativo uma análise utilizando ANOVA e um teste de hipótese.

A idéia inicial deste experimento era gerar no mínimo 10 (dez) quadrados latinos, ou seja, no mínimo conseguir 20 participantes. Devido a época de realização do experimento, época de férias na UFAL, tivemos muita dificuldade em conseguir participantes para o experimento, mesmo com tão poucos requisitos exigidos.

Outro fator que dificultou a escolha de participantes foi a restrição de utilização de um ambiente fechado e livre de interferências para o participante. Isto impossibilitou a execução do experimento com participantes remotos pois seria impossível garantir que o mesmo esteve realmente livre de interferências durante a análise. De toda forma, esta não é uma restrição passível de remoção ou mudança.

Um fato que merece destaque são as explicações dadas nos cenários pelos participantes que erraram a análise utilizando a técnica EI porém acertaram a análise utilizando a técnica EIJ. Tais explicações sugerem que os referidos participantes foram auxiliados pela informação adicional apresentada pela interface emergente com JML.

Com todos os aspectos mostrados neste capítulo concluímos que precisaríamos realizar um novo experimento com mais participantes para chegar de fato ao objetivo do experimento. Todavia, os dados aqui apresentados sugerem uma melhora na expressividade (tempo de análise e nível de entendimento) utilizando interfaces emergentes com JML. Acreditamos que, com mais participantes, é possível atingir plenamente o objetivo proposto neste experimento, rejeitar as hipóteses nulas e afirmar as hipóteses alternativas.

É importante salientar que esta análise foi realizada com cenários de manutenção simples

com variáveis simples, ou seja, qualquer resultado obtido nesta análise são para sistemas parecidos com os apresentados nos cenários PDF e *Best Lap*. Portanto, para sistemas mais complexos não podemos afirmar nada.

Capítulo 6

Conclusão

Mostramos neste trabalho que as interfaces emergentes demonstram problemas de expressividade em algumas situações específicas. Com a baixa expressividade das interfaces emergentes, os desenvolvedores precisarão analisar outras *features* dependentes resultando na diminuição da modularidade ao desenvolver/manter uma *LPS*. Estas situações específicas vão além das mostradas no Capítulo 3 deste trabalho.

Apresentamos, no Capítulo 4, uma forma de melhorar a expressividade das interfaces emergentes através da utilização de contratos JML entre métodos e seu chamadores. Com a informação gerada por estes contratos podemos apresentar interfaces emergentes mais expressivas.

Para confirmar a hipótese de que a utilização desses contratos JML firmados entre os métodos e seus chamadores melhoram a expressividade das interfaces emergentes, propusemos uma avaliação no Capítulo 5 onde participantes analisaram cenários utilizando as duas técnicas: interfaces emergentes (*EI*) e interfaces emergentes com JML (*EIJ*).

Vimos nesta avaliação que existem indícios de melhoria na expressividade das interfaces emergentes com JML, como por exemplo o número de acertos dos participantes utilizando a técnica *EIJ* versus o número de acertos dos participantes utilizando a técnica *EI*.

6.1 Trabalhos Relacionados

Esta seção apresenta outros trabalhos relacionados com o presente trabalho.

6.1.1 Interfaces Emergentes

Por ser uma solução muito recente, atualmente existem poucos artigos publicados na literatura sobre interfaces emergentes. O artigo publicado por Ribeiro e Borba [17] introduziu a utilização de interfaces emergentes como uma forma de atingir a modularidade no desenvolvimento de *features* com abordagens anotativas.

Em 2011, Tarsis Tolêdo propos em seu trabalho de conclusão de curso a ferramenta CIDE EI, uma implementação para interfaces emergentes em forma de *plug-in* para a IDE Eclipse

[20]. Sua implementação teve como base a abordagem anotativa proposta por Kästner et al. [8], *Colored Integrated Development Environment* - CIDE, para desenvolvimento de *features* e utilizou a ferramenta SOOT [22] para realizar as análises de fluxos de dados e atingir o objetivo de apresentar interfaces emergentes para o desenvolvedor.

Ainda em 2011, Ribeiro et al. publicaram um artigo mostrando o impacto da dependência entre as *features* ao manter LPS baseadas em pré-processadores [18]. Este artigo foca em analisar a frequência em que métodos com diretivas de pré-processadores contêm dependência entre features e responder como estas dependências impactam no esforço de manutenção ao utilizar VSoC e interfaces emergentes.

No ano de 2012, até a entrega deste trabalho de conclusão de curso, houveram duas publicações relacionadas com interfaces emergentes. A primeira trata de uma abordagem para realizar análise de fluxo de dados intraprocedural em linhas de produtos de software [3]. A segunda publicação propõe uma nova ferramenta para implementação das interfaces emergentes chamada EMERGO [19].

Este trabalho busca aperfeiçoar as interfaces emergentes através da integração do conceito de *Design by Contract* [12], utilizando a linguagem JML, ao conceito de interfaces emergentes [16] objetivando melhorar a expressividade das mesmas.

6.1.2 JML

Na área de Java Modeling Language, JML, existem diversos trabalhos que buscam aprimorar esta área. A seguir serão listados alguns exemplos:

- ***A Runtime Assertion Checker for the Java Modeling Language*** [4]: Em sua tese de doutorado, Cheon apresentou um compilador, conhecido como *JML compiler* (*jmlc*), para checagem de assertivas em tempo de execução projetado para JML. Este trabalho define um conjunto de regras de tradução de predicados JML para código Java que manipula os diversos tipos de especificações e expressões. Cheon também propõe neste trabalho uma abordagem chamada *wrapper* em que as pré e pós-condições de um método são transformadas em métodos de checagem de assertivas, *assertion checking methods*.

- **JCML - *Java Card Modeling Language*** [14]: Proposto em 2007 por Neto, JCML é um subconjunto da linguagem JML compatível com a tecnologia *Java Card*, utilizada para implementar aplicativos, chamados de *applets*, compatíveis com *Smart Cards*. O compilador JCML (JCMLc) não estende o compilador original da JML e não realiza análise semântica das construções. Seu papel é traduzir as especificações JML em funções de verificação compilando em *bytecode* compatível com aplicações *Java Card*. Estas funções de verificação preservam o máximo possível a estrutura do compilador JML baseado na abordagem *wrapper* [4]. O compilador JCML traduz apenas algumas especificações JML e não possui suporte a herança de especificações.

- ***Implementing JML Contracts with AspectJ*** [15]: O atual *JML compiler* (jmlc) não funciona corretamente quando aplicado as variações do Java, como por exemplo Java ME. O programa objeto resultante do jmlc utiliza o mecanismo *Java reflection* e estrutura de dados não suportadas por Java ME. Proposto por Rebêlo, este trabalho tem como objetivo superar essas limitações, impostas pelo jmlc para Java ME, através da utilização da ferramenta AspectJ para instrumentar código Java com predicados JML. A ferramenta proposta, chamada de *Asject JML Compiler* (ajmlc), define um conjunto de regras de tradução de predicados JML para AspectJ que evitam as construções AspectJ que não são suportadas pelo Java ME. O código resultante é um código compatível com Java SE e Java ME.

Este trabalho propõe uma forma diferenciada de utilização das anotações JML. Em vez de compilar ou traduzir o código JML, o objetivo deste trabalho é interpretar as pré-condições (*requires*) dos métodos para mostrar ao desenvolvedor uma nova interface emergente mais expressiva.

6.2 Trabalhos Futuros

Este trabalho teve como objetivo apenas propor e avaliar uma solução para melhorar a expressividade das interfaces emergentes. Acreditamos que este trabalho servirá como base para diversos trabalhos futuros, como por exemplo:

- **Nova execução do experimento:** Devido a diversos fatores, dentre eles a época em que o experimento foi rodado, não foi possível conseguir muitos participantes para o experimento abordado neste trabalho. Os resultados em experimentos com 20 ou mais participantes podem ser mais significativos.

- **Implementação da solução:** Este trabalho não teve como objetivo implementar esta solução. Os bons indícios na melhora da expressividade das interfaces emergentes com JML nos encorajam a futuramente implementar esta solução.

6.3 Considerações Finais

De acordo com os resultados obtidos, existem indícios que a integração dos contratos JML a interfaces emergentes acrescentam informações que podem auxiliar o desenvolvedor na sua tomada de decisão. É importante salientar que para que as informações geradas pelas *EIJ* sejam valiosas, a LPS deve ter seus contratos cuidadosamente definidos. Contratos incompletos ou mal definidos podem induzir o desenvolvedor ao erro.

Todo e qualquer resultado obtido nesta análise são para sistemas simples com variáveis simples, seguindo a mesma linha dos sistemas apresentados nos cenários PDF e *Best Lap*. Para sistemas mais complexos não podemos afirmar nada.

Com os resultados demonstrados neste trabalho não conseguimos rejeitar as hipóteses nulas para sistemas simples com variáveis simples. Apesar disto, acreditamos que uma nova rodada do experimento é o caminho correto para demonstrar que os contratos JML de fato melhoram a expressividade das interfaces emergentes.

Bibliografia

[1] Sven Apel and Christian Kästner. Virtual separation of concerns - a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, September 2009. ISSN 1660-1769. doi: 10.5381/jot.2009.8.6.c5.

[2] V. R. Basili and H. D. Rombach. *Goal Question Metric (GQM) Approach*. John Wiley & Sons, Inc., 1994.

[3] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, and Paulo Borba. Intraprocedural data-flow analysis for software product lines. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*. ACM, 2012.

[4] Yoonsik Cheon. *A runtime assertion checker for the Java Modeling Language*. Technical report 03-09, Iowa State University, Department of Computer Science, Ames, IA, April 2003. The author's Ph.D. dissertation.

[5] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[6] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 1st edition, September 2000. ISBN 1558604898.

[7] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, November 1990.

[8] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 311–320, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1.

[9] Ronny Kolb, Dirk Muthig, Thomas Patzke, and Kazuyuki Yamauchi. A case study in refactoring a legacy component for reuse in a product line. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, ICSM '05, pages 369–378, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2368-4. doi: 10.1109/ICSM.2005.5.

[10] Gary Leavens and Yoonsik Cheon. Design by Contract with JML, 2006.

[11] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 3540714367.

[12] Bertrand Meyer. Design by contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

[13] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992. ISSN 0018-9162. doi: 10.1109/2.161279.

[14] Plácido Antônio Souza Neto. JCML - Java Card Modeling Language. Master's thesis, Universidade Federal do Rio Grande do Norte, Departamento de Informática e Matemática Aplicada Programa de Pós-Graduação em Sistemas e Computação, November 2007.

[15] Henrique Rebêlo. Implementing JML Contracts with AspectJ. Master's thesis, Departamento de Sistemas e Computação, Universidade de Pernambuco, Recife, Brazil, May 2008.

[16] Márcio Ribeiro and Paulo Borba. Towards feature modularization. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 225–226, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0240-1.

[17] Márcio Ribeiro, Humberto Pacheco, Leopoldo Teixeira, and Paulo Borba. Emergent feature modularization. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 11–18, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0240-1.

[18] Márcio Ribeiro, Felipe Queiroz, Paulo Borba, Társis Tolêdo, Claus Brabrand, and Sérgio Soares. On the impact of feature dependencies when maintaining preprocessor-based software product lines. In *Proc. Generative Programming and Component Engineering (GPCE 2011)*, October 2011.

[19] Márcio Ribeiro, Társis Tolêdo, Johnni Winther, Claus Brabrand, and Paulo Borba. Emergo: a tool for improving maintainability of preprocessor-based product lines. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development Companion*, AOSD Companion '12, pages 23–26, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1222-6. doi: 10.1145/2162110.2162128.

[20] Társis Tolêdo and Márcio Ribeiro. Uma ferramenta para interfaces emergentes em linhas de produto de software. Universidade Federal de Alagoas, 2011. TCC.

[21] L. Cole P. Borba V. Alves, P. M. Jr. and G. Ramalho. Extracting and evolving mobile games product lines. In *Proceedings of the 9th International Software Product Line Conference (SPLC '05)*, volume 3714 of LNCS, pages 70–81. Springer-Verlag, September 2005.

[22] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '99. IBM Press, 1999.

[23] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Bjöorn Regnell, and Anders Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-8682-5.