Trabalho de Conclusão de Curso

Adicionando Contratos a Interfaces Emergentes

Francisco Dalton Barbosa Dias
fdbd@ic.ufal.br

Orientador:
Márcio de Medeiros Ribeiro

Maceió, Julho de 2014

Francisco Dalton Barbosa Dias

Adicionando Contratos a Interfaces Emergentes

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciências da Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientador:

Márcio de Medeiros Ribeiro

Maceió, Julho de 2014

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciências da Computação do Instituto de Computação da Universidade Federal de Alagoas, aprovada pela comissão examinadora que abaixo assina.

Márcio de Medeiros Ribeiro - Orientador
Instituto de Computação
Universidade Federal de Alagoas

Rodrigo de Barros Paes - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Baldoino Fonseca dos Santos Neto - Examinador
Instituto de Computação
Universidade Federal de Alagoas

Maceió, Julho de 2014

Resumo

A realização de manutenção de código de *software* é uma tarefa difícil e que exige cuidados para que o sistema continue funcionando corretamente. Em uma linha de produtos de software, tais manutenções são ainda mais difíceis de serem efetuadas, pois mudanças em algumas funcionalidades podem impactar (através de inserção de erros e comportamentos imprevistos) diversos produtos e, consequentemente, seus clientes e usuários.

Desse modo, é necessário que existam formas de dar suporte ao desenvolvedor durante a execução de tarefas de manutenção de código. Na intenção de prover esse suporte, é apresentado neste trabalho as interfaces emergentes com contratos, na tentativa de tornar as tarefas de manutenção menos suscetíveis a erros e mais eficientes.

A ideia por trás das interfaces emergentes com contratos é que, dado um ponto de manutenção, são encontrados outros pontos no código fonte que podem ser impactados por eventuais mudanças em tal ponto e, então, a partir de contratos definidos no código, são calculadas restrições as quais este ponto deve respeitar para manter o correto funcionamento de todos os outros produtos da linha.

Para avaliar as interfaces emergentes com contratos, realizou-se um experimento controlado com estudantes, cujos resultados foram objetos de análise deste trabalho.

Como resultado, chegou-se à conclusão de que as interfaces emergentes com contratos ajudam a evitar erros e diminuem o tempo necessário para a conclusão de manutenções.

Agradecimentos

Agradeço primeiramente a Deus por me dar forças para lutar e continuar lutando por meus objetivos e proporcionar que eu os alcance. Agradeço também a minha família e principalmente aos meus pais, que sempre depositaram muita confiança em mim e deram total apoio nas minhas decisões. Agradeço em especial a minha irmã, Barbara, que sempre me deu conselhos e me ajudou no que foi necessário. Não posso deixar de agradecer a minha namorada, Isadora, que durante todo o tempo da minha graduação esteve presente nas aflições que o curso de Ciência da Computação me causou, ajudando e dando forças pra continuar.

Agradeço aos professores que, acima de tudo, mostraram que são educadores preocupados com seus alunos e que, por isso, passaram conhecimento e sabedoria muito além do habitual da sala de aula. Agradeço muito ao professor e meu orientador, Márcio Ribeiro, que me mostrou que para fazer ciência é preciso, acima de tudo, ter muita persistência e calma nos momentos de desespero. É um exemplo de idoneidade moral, de profissional e cientista dedicado ao que faz. E que, mesmo sem perceber, me ensinou lições que levarei para o resto da vida.

Por fim, agradeço a todos os amigos que fiz durante a graduação e que estiveram ao meu lado todos esses anos.

Obrigado!

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Uma linha de produtos de software é uma família de sistemas desenvolvidos através da reutilização de um código base que, geralmente, melhora a produtividade e a velocidade na entrega do software [Clements and Northrop 2001]. Porém, devido a grande quantidade de diferentes produtos a serem produzidos, as tarefas de engenharia tornam-se mais desafiadoras quando comparadas com o desenvolvimento de um único produto. Assim, por exemplo, o alto número de *features* possíveis que precisam ser checadas e testadas após cada mudança no código base faz com que as linhas de produtos sejam de difícil manutenção [Kim et al. 2011, Kästner 2010].

Esse cenário é mais complexo quando as linhas de produtos contêm dependências entre *features* (*Cross-feature dependencies*), isto é, quando uma *feature* compartilha elementos do programa, como variáveis e métodos com outras *features* [Ribeiro et al. 2011, Ribeiro et al. 2014].

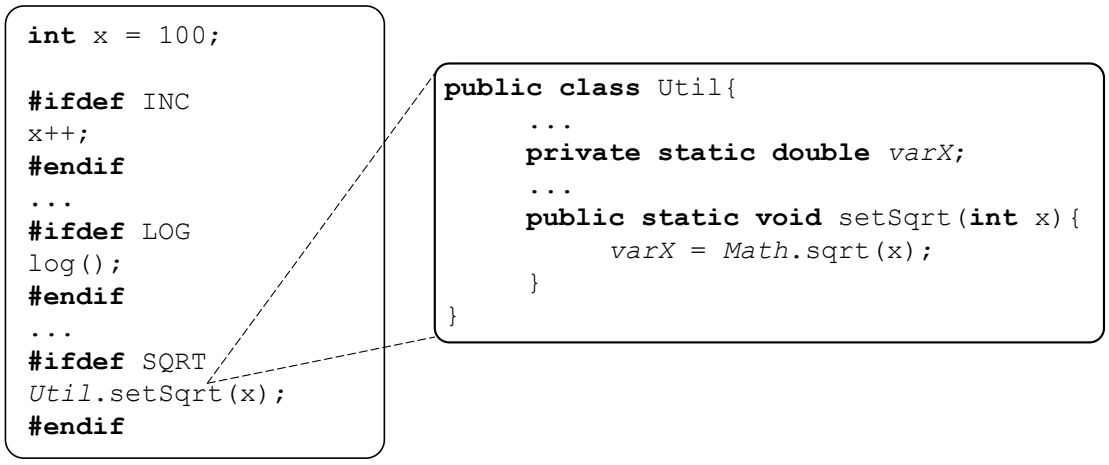


Figura 1.1: Representação das dependências entre *features*.

Na Figura 1.1 é apresentado um exemplo de dependências entre *features*. Nesse exemplo, a variável `x` é definida no código base do *software*, comum a todas as *features* e usada

no código das *features* `INC` e `SQRT`. Em uma eventual modificação na definição da variável `x`, como, por exemplo, uma modificação em seu tipo ou alteração de seu valor inicial, os desenvolvedores precisam rastrear essas dependências para garantir que a modificação do código não irá causar problemas nas *features* [Ribeiro et al. 2010].

Nesse contexto, a tarefa de buscar as dependências ocorrerá frequentemente e o não reconhecimento dessas dependências entre as *features* poderá ocasionar erros sintáticos, perceptíveis durante a compilação dos produtos ou, pior ainda, erros comportamentais que só serão percebidos durante a execução de um produto específico [Cataldo et al. 2009, Ribeiro et al. 2014]. Em um estudo envolvendo 43 *softwares* de código aberto, entre eles: Linux, FreeBSD e GCC, constatou-se que dependências entre *features* são comuns na prática [Ribeiro et al. 2011].

Visando minimizar os problemas causados pelas dependências, foi proposto o conceito de Interfaces Emergentes (*Emergent Interfaces*) [Ribeiro et al. 2010, Ribeiro et al. 2014]. Uma interface emergente é uma abstração das dependências do fluxo de dados de uma *feature* [Ribeiro et al. 2014]. Essas interfaces são responsáveis por descrever as dependências entre as *features*.

Desta forma, uma *feature* pode fornecer dados para outras, da mesma maneira que poderá requisitá-los. No entanto, diferente de uma interface comum, os desenvolvedores não precisam escrever as interfaces emergentes. Em vez disso, eles devem requerê-las sob demanda, selecionando o trecho de código que será alvo da tarefa de manutenção—o ponto de manutenção—e, então, as interfaces são inferidas e emergem no ambiente de desenvolvimento integrado(IDE).

Assim, quando um desenvolvedor for realizar uma modificação no código de uma *feature*, ele poderá requisitar uma interface emergente. Desta maneira, ao analisar a interface, ele tomará consciência das dependências entre a *feature* que está modificando e as demais, identificando trechos de código que precisarão de sua atenção e, assim, evitando a introdução de erros [Ribeiro et al. 2014].

Entretanto, apesar dos benefícios das interfaces emergentes, elas ainda não são suficientes para fornecer a modularização das *features*, que visa alcançar a compreensibilidade e capacidade de manutenção das *features* de maneira independente [Parnas 1972]. Nesse contexto, considere um desenvolvedor que deve modificar o valor da variável `x` no código base, apresentado na Figura 1.1. Uma interface emergente indica que as *features* `INC` e `SQRT` usam tal variável, fazendo com que o desenvolvedor ignore a *feature* `LOG`. No entanto, não obstante a diminuição do esforço decorrente da não verificação de *features* que não estão relacionadas ao ponto de manutenção, o desenvolvedor ainda terá que analisar as *features* que usam a variável `x` e que, potencialmente, não estão sob sua responsabilidade, uma vez que sua tarefa é modificar apenas o código base e não as *features*.

Tal cenário dificulta a independência da compreensibilidade e da capacidade de manutenção, devido a falta de informação semântica: As interfaces emergentes não fornecem

nenhuma informação sobre `x`, como o intervalo de valores válidos para `x`. Por exemplo, se o método `sqrt` da classe `Math` for chamado com um parâmetro negativo, o resultado será um `NaN`. Assim, cabe ao desenvolvedor verificar quando as modificações pretendidas irão manter o correto funcionamento das *features* que dependem da variável `x`.

Para tentar reduzir esse problema, o presente trabalho apresenta os conceitos iniciais das interfaces emergentes com contratos (*Emergent Contract Interfaces*) que compreendem as interfaces emergentes, agregando informação semântica sobre elementos do programa. Essa informação é importante para melhorar a compreensão e a capacidade de manutenção, já que, por exemplo, uma interface emergente com contratos pode indicar que a *feature* `SQRT` requer que `x` seja maior ou igual a zero para funcionar corretamente. Desta maneira, o desenvolvedor que deve realizar uma modificação no valor de `x` sabe exatamente o intervalo de valores válidos para a variável sem, necessariamente, observar as outras *features*.

A informação semântica é capturada a partir de contratos como os escritos em linguagens, como, por exemplo, *Eiffel* [Meyer 1988], *Java Modeling Language* [Leavens et al. 2006] ou *Spec#* [Barnett et al. 2011]. Como sugerido por *design by contracts* [Meyer 1992], assume-se que esses contratos foram especificados previamente durante o desenvolvimento do software. Além das interfaces emergentes com contratos, as aplicações dos contratos são diversas e vão de documentação e *runtime assertion checking*, até verificação formal [Burdy et al. 2005, Hatcliff et al. 2012].

Em comparação às interfaces emergentes, as interfaces emergentes com contratos podem:

(i) diminuir o tempo necessário para que o desenvolvedor determine as dependências de um ponto de manutenção;

(ii) melhorar o entendimento do desenvolvedor das dependências de um ponto de manutenção.

Para avaliar as interfaces emergentes com contratos, este trabalho reporta a execução de um experimento controlado comparando as interfaces emergentes e interfaces emergentes com contratos, executado com estudantes de uma disciplina de linha de produtos de software da Universidade de Magdeburg, Alemanha. Os estudantes, em poucas linhas, executaram tarefas de compreensão, e os resultados indicaram que as interfaces emergentes com contratos diminuem a quantidade de erros durante a tarefa de compreensão, mas sem efeito estatisticamente significativo no tempo de resposta.

Em resumo, são realizadas as seguintes contribuições:

- São introduzidos os conceitos inicias das interfaces emergentes com contratos; e
- É avaliado o potencial das interfaces emergentes com contratos na melhora da compreensão de trechos de um programa durante uma tarefa de manutenção, com a

análise do experimento controlado realizado com 22 (vinte e dois) participantes, trabalhando em duas linhas de produtos. São avaliados, ainda, o tempo de resposta e a quantidade de respostas incorretas em cada tarefa.

Por fim, o restante deste trabalho de conclusão de curso está organizado da seguinte forma:

- Capítulo 2 apresenta e revisa conceitos que serão indispensáveis ao pleno desenvolvimento do trabalho ora apresentado;
- Capítulo 3 Traz a problemática trabalhada, com suas especificações;
- Capítulo 4 apresenta os conceitos iniciais da solução proposta: as interfaces emergentes com contratos;
- Capítulo 5 descreve o experimento realizado para avaliar o desempenho das interfaces emergentes com contratos em comparação com as interfaces emergentes convencionais;
- Capítulo 6 apresenta trabalhos que estão relacionados a temas discutidos nesse trabalho de conclusão de curso; e
- Capítulo 7 resume as conclusões obtidas desse trabalho, lista possíveis trabalhos futuros e finaliza com algumas considerações importantes.

Capítulo 2

Fundamentação Teórica

No presente capítulo, são tratados os principais conceitos abordados neste trabalho, o que facilitará a compreensão e interpretação do tema proposto.

2.1 Linha de Produtos de Software

No mercado de software é comum encontrar sistemas que apresentam diversas versões, todas partilhando de um núcleo em comum, chamado *CORE*. Poderão ser diferentes, seja por suas funcionalidades, forma de implementação ou o modo como atendem as necessidades dos usuários. A esse conjunto de sistemas que compartilham de uma base em comum, é dado o nome de família de software. Essas famílias apresentam sistemas similares, diferenciados no que toca ao conjunto de funcionalidades, o que proporciona melhorias na produtividade e na velocidade de entrega dos softwares [Clements and Northrop 2001].

Contudo, para realizar a construção em massa de diferentes sistemas baseados nessas famílias de software [Pohl et al. 2005], é necessário considerar uma Linha de Produtos de Software (LPS), que pode ser definida como um conjunto de sistemas de software que compartilham características e que são desenvolvidos a partir de um núcleo comum e de forma preestabelecida para atender as necessidades de um mercado específico [Clements and Northrop 2001]. Além da velocidade na criação de diferentes produtos para diferentes públicos, as linhas de produtos ainda possibilitam diversos benefícios [Clements and Northrop 2001]:

- **Redução dos custos de desenvolvimento:** A reutilização de diferentes funcionalidades para compor diversos produtos, tornam o desenvolvimento das LPS mais acessíveis, uma vez que não será necessário recriar, para cada software, as mesmas funcionalidades partindo-se de seu início, do zero.

- **Melhoria na qualidade:** A reutilização de um único núcleo e das funcionalidades em diferentes produtos, fazem com que esses códigos sejam testados e melhorados

diversas vezes, aumentando a probabilidade de se encontrar falhas, caso existam.

- **Redução do tempo de entrega:** Após a conclusão do projeto e dos códigos que fazem parte do núcleo da LPS, o tempo de entrega de um novo produto tende a diminuir, pois a quantidade de funcionalidades já implementadas diminui o esforço de criação de um novo produto.

Contudo, é essencial desenvolver um modo de gerenciar as *features* que compõem os diferentes produtos possíveis. Para isso, é comum a utilização de uma forma de representação chamada de *Feature Model*. Um *feature model* é um conjunto de *features* organizadas hierarquicamente [Batory 2005]. As *features* podem apresentar um relacionamento de mãe e filhas, que podem ser visualizados na Figura 2.1 e são caracterizados em:

- *E*—todas as *features* filhas devem ser selecionadas;
- *Alternativa*—apenas uma *feature* filha pode estar selecionada;
- *Ou*—uma ou mais *features* podem ser selecionadas;
- *Obrigatórias*—são *features* que devem estar selecionadas; e
- *Opcionais*—são *features* opcionais.

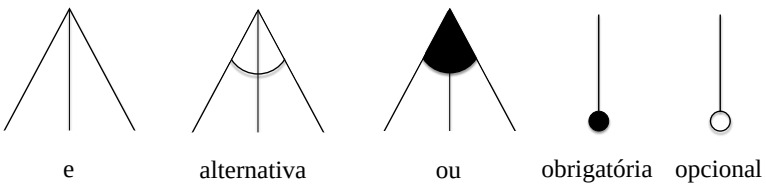


Figura 2.1: Representações gráficas dos relacionamentos do *feature model* [Batory 2005].

Na Figura 2.2 destaca-se a representação do *feature model* de um *e-shop*.[1] Nele é possível notar os relacionamentos existentes entre as *features*. Por exemplo, as *features* `Catalogue`, `Payment` e `Security` são todas obrigatórias e, por isso, devem estar presentes em qualquer produto gerado, ao passo que a *feature* `Search` é opcional e sua inclusão em algum produto fica a critério do desenvolvedor. É possível notar, ainda, que as *features* `Bank Transfer` e `Credit card` podem estar presentes em um mesmo produto, mas as *features* `High` e `Standard` não. Adicionalmente, está representado através da restrição `Credit card implies High` que a existência da *feature* `Credit card` em um produto, obriga a presença da *feature* `High` nesse mesmo produto.

[1] Disponível em http://en.wikipedia.org/wiki/Feature_model

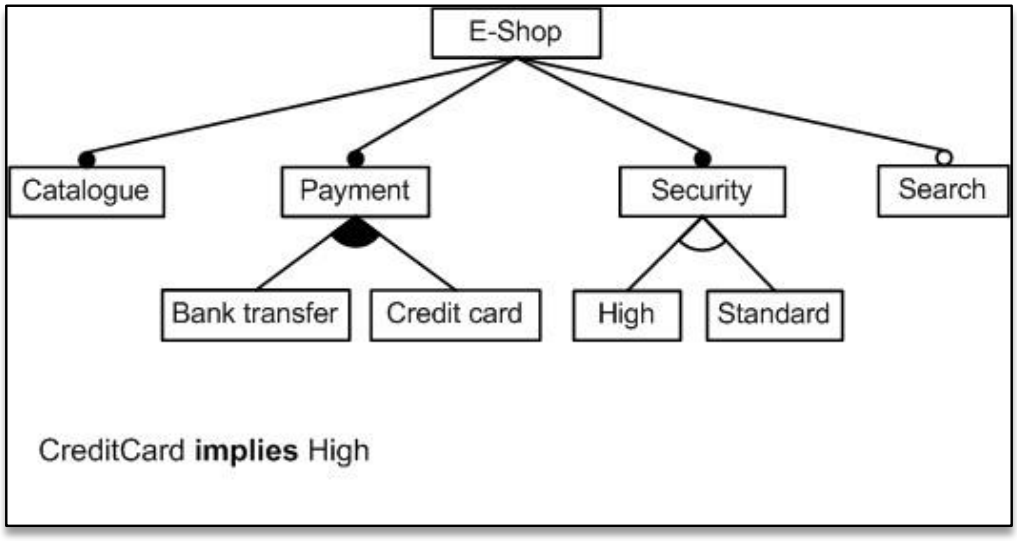


Figura 2.2: *Feature model* de um e-shop.

2.2 Compilação Condicional

Uma das formas mais comuns no desenvolvimento de linhas de produtos consiste em anotar *features* através de compilação condicional, um mecanismo simples e largamente utilizado [Kästner et al. 2011] que, usualmente, utiliza pré-processadores léxicos, como o *C preprocessor* [ISO 2011] e *Antenna*[2] para Java [Figueiredo et al. 2008]. Seu funcionamento é simples: diretivas de compilação como `#ifdef` e `#endif` envolvem instruções de códigos que estão associados à uma determinada *feature*, fazendo com que essas instruções apenas sejam compiladas quando a *feature* estiver selecionada para compor o produto final ou, então, completamente ignoradas e excluídas do produto final no caso contrário, por exemplo.

As *features* de um produto são selecionadas através da passagem de parâmetros de configuração para o compilador, o que pode ser feito através de um arquivo de configuração ou através de linhas de comando. Um dos exemplos mais notáveis que fazem uso dessa forma de implementação de linhas de produtos é o *Linux*, que possui milhares de *features* [Liebig et al. 2010].

[2] `http://antenna.sourceforge.net/`

```
649  void notify_cpu_starting(unsigned int cpu)
650  {
651        unsigned long val = CPU_STARTING;
652
653  #ifdef CONFIG_PM_SLEEP_SMP
654        if(fronzen_cpus != NULL && cpumask_test_cpu(cpu, fronze_cpus))
655              val = CPUT_STARTING_FRONZEN;
656  #endif
657        cpu_notify(val, (void *)(long)cpu);
658  }
```

Figura 2.3: Trecho de código do *Linux* envolto por uma *feature*.

A Figura 2.3 mostra um método[3] do *kernel* do *Linux* que possui um pequeno trecho de código envolto por uma *feature* chamada `CONFIG_PM_SLEEP_SMP`. Dessa forma, as linhas 654 e 655 só serão compiladas quando a *feature* `CONFIG_PM_SLEEP_SMP` estiver selecionada.

Uma das principais vantagens—e também um dos maiores problemas—dos pré-processadores é que eles são muito flexíveis. Com eles, é possível definir *features* das mais variadas granularidades, desde arquivos inteiros até sequências de *tokens* da linguagem. Assim, os pré-processadores não estão restritos a estruturas sintáticas [Kästner et al. 2011].

Na linguagem Java, é possível adicionar diretivas de pré-processamento através de comentários no código. Por exemplo, um comentário no formato `//#ifdef` e `//#endif` é interpretado pelo pré-processador Antenna como sendo uma diretiva de pré-processamento.

2.3 Interfaces Emergentes

Ao realizar a manutenção em um código fonte, a introdução de erros é algo comum, principalmente quando existem falhas no reconhecimento de dependências entre módulos e *features* [Cataldo and Herbsleb 2011]. No contexto de *features*, isso pode acontecer com frequência, uma vez que é comum encontrar dependências entre *features* (i.e., *features* que compartilham elementos e métodos umas com as outras [Ribeiro et al. 2011]). Desse modo, uma alteração realizada em uma *feature* pode impactar negativamente em outras, podendo ocasionar comportamentos imprevistos.

Para minimizar os problemas ocasionados pelas dependências existentes entre as *features* que nem sempre são percebidas pelos desenvolvedores, foi proposta uma técnica chamada de Interfaces Emergentes (*Emergent Interfaces - EI*) [Ribeiro et al. 2010, Ribeiro et al. 2014]. Essa técnica consiste no estabelecimento, sob demanda, de interfaces para trechos de códigos de acordo com uma tarefa de manutenção.

[3]Disponível em https://github.com/torvalds/linux/blob/master/kernel/cpu.c

Uma interface emergente é uma abstração das dependências encontradas no fluxo de dados de uma *feature* a partir do ponto de manutenção. Representada através de um conjunto de cláusulas *provides* e *requires* que descrevem quais dados a *feature* fornece e, ao mesmo tempo, quais dados ela necessita. Assim, as dependências de dados entre as *features* tornam-se explícitas. Diferentemente de uma interface convencional, as interfaces emergentes não precisam ser escritas, em vez disso, elas são geradas sob solicitação do desenvolvedor. São, assim, inferidas e emergem no ambiente de desenvolvimento integrado (IDE), tornando o desenvolvedor consciente de eventuais dependências entre a *feature* que ele está mantendo e as outras. Existe, portanto, um aumento das chances da tarefa ser concluída com sucesso sem ocasionar efeitos colaterais nas outras [Yin et al. 2011].

Além disso, a utilização da ferramenta que implementa as interfaces emergentes, chamada *Emergo* [Ribeiro et al. 2012], proporciona diminuição do esforço necessário para a realização de manutenção no código, uma vez que a busca pelas dependências através do código é realizada pela ferramenta. Esse fato foi comprovado em um experimento controlado [Ribeiro et al. 2014, Ribeiro 2012], onde os participantes tiveram que realizar tarefas de manutenção de código. Os resultados desse experimento mostraram que quando os participantes usaram a ferramenta, tiveram diminuições no tempo total para a conclusão das tarefas e na quantidade de erros cometidos quando comparados aos participantes que realizaram as tarefas sem o auxílio da ferramenta.

Na Figura 2.4 é mostrada a interface emergente referente a uma requisição, tomando como ponto de manutenção a variável `val` na linha 651 do código apresentado na Figura 2.3. É importante salientar que essa interface emergente está limitada ao escopo do método `notify_cpu_starting`. Nela é possível notar que a *feature* `CONFIG_PM_SLEEP_SMP` requisita a variável `val` do código base e, ao mesmo tempo, fornece a variável `val` para outra instrução do código base.

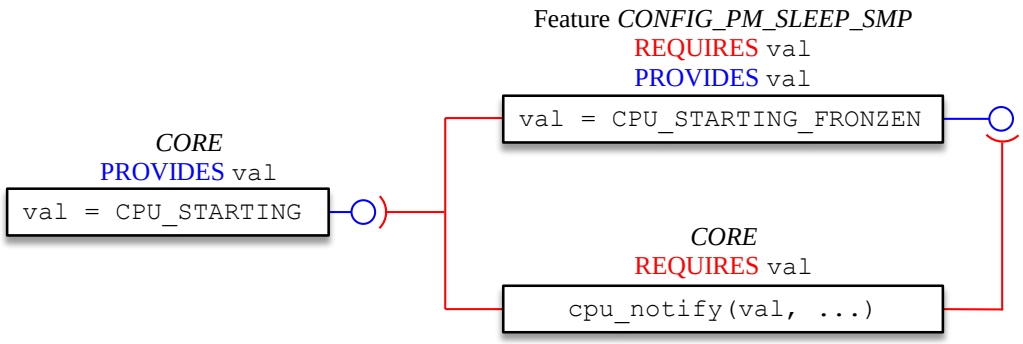


Figura 2.4: Interface Emergente para para a variável `val` do trecho de código do Linux.

Uma forma mais simples de representar essa interface emergente é mostrada a seguir:

"Feature `CONFIG_PM_SLEEP_SMP` **requires** `val`"

2.4 Design by Contracts

Neste trabalho, a solução proposta faz uso dos conceitos do *Design by Contracts* (DBC). Por isso, nesta seção são apresentados os conceitos fundamentais para o seu pleno entendimento.

A convivência humana está, naturalmente, cercada e regida por contratos. Sempre que se terceirizam tarefas, contratos estão sendo estabelecidos. Existem duas importantes propriedades que caracterizam os contratos humanos, entre as partes [Meyer 1992]:

- Ambas as partes de um contrato desejam algum benefício do contrato e estão dispostas a cumprir alguma obrigação para recebê-lo; e
- Os benefícios e obrigações de cada parte devem estar descritos em um contrato formal.

Retomando o exemplo apresentado na introdução deste trabalho (ver Figura 1.1), é apresentado na Tabela 2.1 uma lista de benefícios e obrigações do cliente—o chamador do método `setSqrt`—e do fornecedor—o método `setSqrt`—. Assim, através dos contratos, as partes estão protegidas, uma vez que, para que um lado garanta o cumprimento de suas obrigações, o outro precisa lhe fornecer os benefícios exigidos. Consequentemente, o que é uma obrigação para uma parte acaba sendo, geralmente, um benefício para a outra (ônus e bônus).

Parte	Obrigações	Benefícios
Cliente	Chamar o método `setSqrt` passando um parâmetro maior ou igual a zero.	Ter o valor da variável `varX` calculado e atribuído corretamente.
Fornecedor	Calcular a raiz quadrada do parâmetro `x` e atribuí-la a variável `varX`.	Não trabalhar com valores que causarão erros em sua execução.

Tabela 2.1: Lista de benefícios e obrigações estabelecidas em um contrato.

A ideia central para o funcionamento do DBC é que cada classe e seus clientes possuem contratos uns com os outros [Leavens and Cheon 2006]. Desse modo, para que um cliente tenha certeza de que as obrigações expostas por uma classe serão cumpridas, ele precisa garantir algumas condições antes de invocar um dos métodos oferecidos por essa classe.

2.5 Java Modeling Language

Os contratos usados neste trabalho são representados através da *Java Modeling Language* (JML), uma vez que a linguagem de programação Java é usada como linguagem base para este trabalho. Assim, maiores detalhes sobre a JML serão apresentados a seguir.

A JML [Leavens et al. 2006] é uma linguagem de especificação formal de interfaces de comportamento que, assim como *Eiffel* [Meyer 1988] e *Spec#* [Barnett et al. 2011], permite a definição de contratos, como sugerido pelo DBC. A JML permite que sejam definidas interfaces sintáticas de código Java e seu comportamento [Leavens and Cheon 2006]. Informações de checagem de tipos, visibilidade e outros modificadores fazem parte das interfaces sintáticas. Um método pode, por exemplo, especificar em seu cabeçalho uma interface sintática que consiste em tipo de retorno, tipos de parâmetros e outros modificadores. Já o comportamento descreve o que deve ocorrer, em tempo de execução, quando aquele trecho de código for executado. As especificações JML são escritas em comentários de anotações especiais iniciados pelo símbolo arroba (`@`), em comentários de apenas uma linha (`//@`) ou em blocos (`/*@ ... @*/`).

Na JML, as especificações das obrigações do cliente são definidas através da cláusula `requires`, ao passo que as obrigações do fornecedor são definidas pela cláusula `ensures`. Um contrato é, geralmente, expresso através de precondições e pós-condições de um método. A precondição declara o que precisa ser verdade para que o método funcione corretamente, ao passo que a pós-condição especifica o que será verdade ao final da execução. Por exemplo, na Figura 2.5 é retomado o método `setSqrt` apresentado anteriormente, acrescido de anotações JML. A anotação `//@requires x >= 0;` indica que, para o método funcionar corretamente, o valor do parâmetro `x` deve ser maior ou igual a zero. Do contrário, algum comportamento imprevisto ou algum erro pode acontecer. Já a anotação `//@ensures this.varX >= 0.0;` especifica que no caso da precondição ser satisfeita, a variável `varX` da classe `Util` terá seu valor calculado corretamente e que esse valor será maior ou igual a zero.

```
public class Util{
      ...
      private static double varX;
      ...
      //@requires x >= 0;
      //@ensures varX >= 0.0;
      public static void setSqrt(int x){
            varX = Math.sqrt(x);
      }
}
```

Figura 2.5: Exemplo de método anotado.

Os contratos JML são, ainda, uma rica fonte de documentação, uma vez que para cada método presente em uma classe ou interface os contratos especificam o que é necessário para o correto funcionamento dos métodos e quais as garantias que eles dão após a execução. Além disso, são usados para *runtime assertion checking* e verificação formal [Burdy et al. 2005, Hatcliff et al. 2012] através de ferramentas de suporte.[4]

2.6 Weakest Precondition

Para capturar a informação semântica adicionada ao código através das anotações JML e, por fim, calcular a interface emergente com contratos, este trabalho faz uso das ideias apresentadas no cálculo da *Weakest precondition*, que é explicado a seguir.

E. W. Dijkstra propôs uma estratégia para a verificação de programas, denominada *weakest precondition predicate transformer* (wp) [Dijkstra 1976]. Essa estratégia foca em sistemas que apresentam em seu estado inicial uma combinação de parâmetros e, esses parâmetros, definem o estado final que o sistema irá se encontrar após sua execução, isto é, os valores de entrada—parâmetros de um método, por exemplo—representam a escolha do estado inicial e os valores de saída—o resultado da execução do método—representam o estado final.

Dijkstra definiu a precondição mais fraca (*weakest precondition*) através da função *wp(S,R)*, onde S é o sistema (máquina, mecanismo, método, ...), e R é a pós-condição desejada. A *pós-condição* especifica quais condições o estado final do sistema deve satisfazer após sua execução. É importante salientar que diferentes estados iniciais podem levar ao mesmo estado final. Contudo, quando se deseja que o sistema chegue a um estado final que satisfaça a uma pós-condição específica, pode ser desejável saber quais condições iniciais levarão o sistema a esse estado final. Ao conjunto dessas condições iniciais, Dijkstra chamou de "a condição mais fraca correspondente a essa pós-condição" (*the weakest pre-condition corresponding to that post-condition*). O termo "mais fraca" (*weakest*) foi atribuído pelo fato de que quanto mais fraca for a condição, mais estados a satisfazem.

Seu objetivo era caracterizar todos os estados possíveis que levavam o sistema a um certo estado final. Ou seja, podem existir diversas precondições Q que irão levar o sistema S a satisfazer uma pós-condição R, mas existe apenas uma precondição—a precondição mais fraca—que irá descrever o maior conjunto de estados iniciais que irão levar o sistema S a um estado satisfazendo R.

Funcionamento

Suponha um programa S, composto por diversas instruções: s_1; s_2; s_3; ...; s_n e uma pós-condição R. Suponha ainda que se deseje realizar a verificação da precondição mais

[4] http://www.jmlspecs.org

fraca P do programa S. A Figura 2.6 mostra que o cálculo é iniciado de baixo para cima, a partir da instrução s_n e da pós-condição R, produzindo a precondição P_{n-1}, a primeira das diversas condições de verificação que devem ser estabelecidas [Popov 2003]. P_{n-1} é a precondição mais fraca da instrução s_n e, ao mesmo tempo, pós-condição desejada da instrução s_{n-1}. Isso se repete até acabarem as instruções intermediárias do programa S.

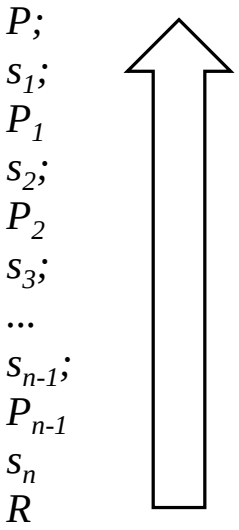


Figura 2.6: Fluxo do cálculo da precondição mais fraca.

Ao final da execução, é determinada a precondição mais fraca P que define o conjunto de estados inicias que, ao serem executados no programa S, irão levá-lo a um estado final que satisfaz a pós-condição R.

Exemplo

Para tornar mais clara a execução da função *wp(S,R)* suponha que S seja um programa compostos pelas instruções s_1; s_2 e R a pós-condição cujo estado final do programa S deve satisfazer. Seja:

$$s_1 : x = x + 3$$

$$s_2 : y = y - 4$$

$$R : x + y > 0$$

Como o programa S é composto por duas instruções, a função wp assume a seguinte forma:

$$wp(s_1, wp(s_2, R))$$

Substituindo a instrução s_2 e a pós-condição R, tem-se:

$$wp(s_1, wp(y = y - 4, x + y > 0))$$

A execução da função wp é simples, sendo necessário apenas que a variável y na pós-condição seja isolada, o que resulta em $y > -x$. Uma vez que na instrução s_2 temos $y = y - 4$, podemos concluir que $y - 4 > -x$. Ao reorganizar essa inequação tem-se:

$$wp(s_1, x + y > 4)$$

Dessa forma, $x + y > 4$ é a precondição mais fraca da instrução s_2 e pós-condição desejada da instrução s_1. Agora, substituindo s_1 por $x = x + 3$:

$$wp(x = x + 3, x + y > 4)$$

Por fim, ao isolar a variável x da pós-condição, tem-se $x > -y + 4$. Aplicando-a na instrução s_1, chega-se a: $x + 3 > -y + 4$. Assim, a precondição mais fraca do programa S, é:

$$x + y > 4 - 3$$

$$x + y > 1$$

Logo, para que o programa S termine sua execução com $x+y > 0$, é condição necessária que a soma $x + y$ seja, inicialmente, maior que 1. Por exemplo, se inicialmente $x = 2$ e $y = -3$ tem-se $x + y = -1$, o que fere a precondição $x + y > 1$. Assim, ao executar as instruções, tem-se:

$$x = 2 + 3 \therefore x = 5$$

$$y = -3 - 4 \therefore y = -7$$

Logo,

$$x + y = -2$$

Como demonstrado acima, o resultado ao final da execução da expressão $x + y$ não faz parte do conjunto de estados que satisfazem a pós-condição desejada. Isso mostra que o não cumprimento da precondição mais fraca fez com que o programa S terminasse em um estado indesejado.

Capítulo 3

Problema

Neste capítulo são apresentados os benefícios da utilização das interfaces emergentes na manutenção de uma linha de produtos, bem como relatadas suas deficiências, que são alvo da solução proposta neste trabalho. Para melhorar o entendimento, é apresentado um exemplo onde aplica-se os conceitos das interfaces emergentes em uma tarefa de manutenção hipotética.

A finalidade, de tal explanação é deixar claro, a partir do exemplo apresentado, os benefícios e deficiências do uso das interfaces emergentes nas tarefas de manutenção de *software*.

O exemplo apresentado neste capítulo representa um cenário de um jogo comercial de corridas de carros para celulares, denominado *Best Lap*.[1] Seus jogadores têm o dever de conseguir obter a volta mais rápida para classificar-se na primeira posição. No código do jogo existe o método chamado `computeLevel`, apresentado na Figura 3.1, responsável por computar a pontuação dos jogadores.

Neste método, encontram-se duas *features* opcionais: `PENALTIES`, que adiciona penalidade à pontuação do jogador—na hipótese de bater durante o jogo, por exemplo—e `ARENA`, que publica a pontuação do jogador em um *ranking online*. Para o correto funcionamento da *feature* `ARENA`, a pontuação deve ser maior ou igual a zero.

Um de seus desenvolvedores é encarregado de modificar a forma de cálculo da pontuação dos jogadores. Para tanto, ele deve alterar a inicialização da variável `totalScore`, encontrada dentro do método `computeLevel` (retângulo pontilhado na Figura 3.1), sendo esse o ponto de manutenção da tarefa. Pontos de manutenção são definidos como instruções que o desenvolvedor pretende alterar durante a realização da manutenção.

[1] Desenvolvido por http://www.meantime.com.br

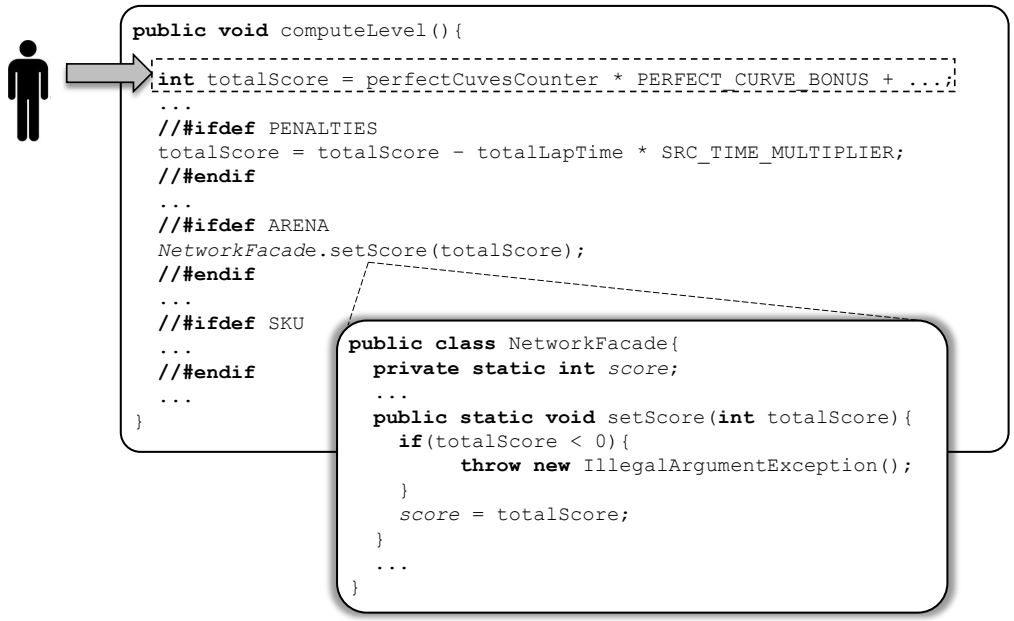


Figura 3.1: Código do *Best Lap* que será alvo de uma manutenção.

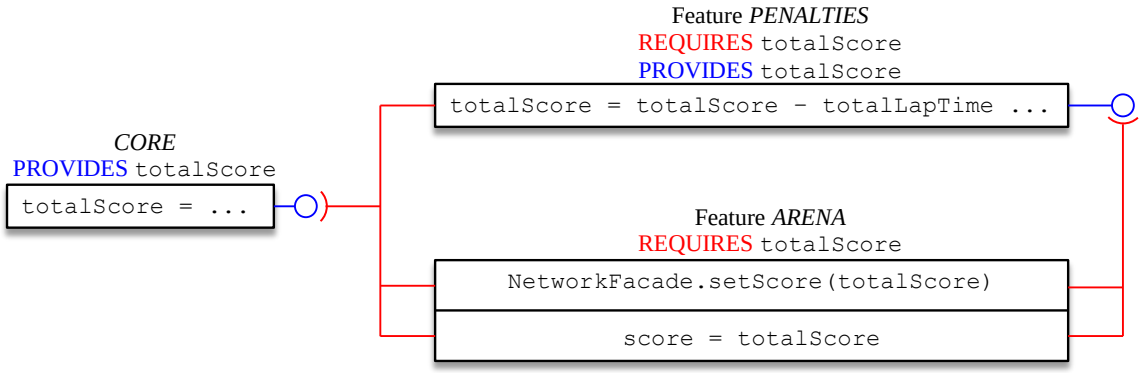


Figura 3.2: Interface emergente para a variável `totalScore`.

Assim, suponha que antes da modificação da atribuição da variável `totalScore`, o desenvolvedor requisite as interfaces emergentes, com a finalidade de melhor compreender as dependências existentes. Na Figura 3.2 ilustra-se a interface emergente para o ponto de manutenção indicado anteriormente. Pode-se perceber que as *features* fornecem e requisitam dados, umas às outras. A *feature* `PENALTIES`, por exemplo, requisita dados do `CORE`—o código base—e, da mesma forma, fornece dados para a *feature* `ARENA`.

As interfaces emergentes são capazes de fornecer benefícios, quando se fala em diminuição do esforço e número de erros introduzidos [Ribeiro et al. 2014]. Assim, uma vez que a *feature* `SKU` não utiliza a variável `totalScore`, a interface emergente não indicará a existência de dependências entre o ponto de manutenção e essa *feature*. Desse modo, as interfaces emergentes ajudam os desenvolvedores a focarem apenas nas *features* que são

dependentes do ponto de manutenção, evitando perda de tempo e esforço do desenvolvedor ao analisar *features* que não apresentam nenhuma relação com a tarefa que está sendo executada.

Todavia, é importante destacar que o conjunto de cláusulas de fornecimento e requisição das interfaces emergentes continuam insuficientes para prover a modularização das *features*, que visa alcançar a compreensibilidade e capacidade de manutenção das *features* de maneira independente [Parnas 1972]. Isso significa que, com a assistência prestada pelas interfaces emergentes, o desenvolvedor deveria ser capaz de entender e modificar uma *feature* sem que se quebrasse ou fosse necessário checar o código de outras *features*. Porém, essa afirmativa não é verdadeira quando se analisa o exemplo exposto, pois a simples informação de que as *features* `ARENA` e `PENALTIES` requisitam a variável `totalScore`, não presta garantias ao desenvolvedor de que tais *features* estarão funcionando corretamente após a realização da modificação do código. Por exemplo, se a modificação que será feita pelo desenvolvedor permitir que a variável `totalScore` assuma valores negativos, isso irá causar erros nos produtos que possuem a *feature* `ARENA`.

Por tal razão, o desenvolvedor ainda precisará investigar manualmente o código fonte e, provavelmente, a documentação das *features*, para decidir se a modificação a ser realizada afetará a *feature* `PENALTIES` ou a `ARENA`, antes de continuar com a manutenção.

Entretanto, apesar dos benefícios provenientes das interfaces emergentes, elas ainda não são suficiente para alcançar a modularização das *features*, uma vez que:

- São pouco expressivas devido a falta de informações semânticas;
- O desenvolvedor ainda deve desperdiçar tempo e esforço verificando as *features* que possuem dependências com a *feature* que está sendo mantida em vez de focar apenas nela.

Visando minimizar os problemas acima relatados e proporcionar melhora na modularização das *features*, será introduzida, no capítulo seguinte, a ideia das interfaces emergentes com contratos, onde as interfaces emergentes serão melhoradas com a adição de informações existentes em contratos escritos no código fonte, provendo, assim, informação semântica sobre os elementos integrantes do programa.

Capítulo 4

Interfaces Emergentes com Contratos

O conceito de interfaces emergentes com contratos (*Emergent Contract Inferfaces - ECI*) partiu da adição de contratos aos modelos convencionais de interfaces emergentes. Ambas possuem informações sobre os blocos de código que são influenciados por um dado ponto de manutenção, mas é necessário pontuar que as interfaces emergentes com contratos diferenciam-se pelo fato de fornecerem contratos capazes de descrever as condições em que o ponto de manutenção deve ser mantido para que não ocasione erros em outras *features*.

A seguir, o exemplo apresentado no Capítulo 3 é retomado e mostrado como os contratos podem melhorar a expressividade das interfaces emergentes.

4.1 Enriquecendo as interfaces emergentes

Para melhorar a expressividade das interfaces emergentes é necessário, primeiramente, definir uma forma de fornecer informações semânticas a respeito dos elementos que compõem o *software*. Neste trabalho, a proposta é que as informações semânticas sejam explicitadas pelos desenvolvedores a partir da utilização de anotações JML. Para ilustrar como as anotações JML podem enriquecer o código de um *software* com informações semânticas, considere o exemplo apresentado na Figura 4.1. Essa figura ilustra o método `setScore` que foi inicialmente introduzido na Figura 3.1.

A restrição existente citada anteriormente de que o método `setScore` exige que o valor do seu parâmetro seja maior ou igual a zero, é descrita através da anotação `//@requires totalScore >= 0` no cabeçalho do método, sendo essa a precondição para que o método `setScore` funcione corretamente. Adicionalmente, a anotação `//@ensures this.score == totalScore` assegura que se a precondição for respeitada, o valor de `totalScore` será atribuído à variável `score` da classe `NetworkFacade`. Além disso, as anotações tornam essas informações explicitas tanto para outros desenvolvedores quanto para ferramentas desenvolvidas para realizar verificações a partir das anotações JML.

```
public class NetworkFacade{
  private static int score;
  ...
  //@requires totalScore >= 0;
  //@ensures score == totalScore;
  public static void setScore(int totalScore){
     if(totalScore < 0){
         throw new IllegalArgumentException();
    }
    score = totalScore;
  }
  ...
}
```

Figura 4.1: Método da *feature* `ARENA` com anotações JML.

O contrato apresentado no método `setScore` é útil na realização de tarefas de manutenção, pois modificar a forma de cálculo da variável `totalScore` pode tornar possível que `totalScore` assuma valores negativos, o que infringirá o contrato estabelecido por `setScore`, podendo levar o sistema a um estado inválido.

Porém, a simples visualização da precondição do método `setScore` como parte da interface emergente, assim como feito na Figura 4.2, ainda é insuficiente uma vez que, como será mostrado a seguir, a dependência entre *features* pode modificar os contratos. Logo, ainda é necessário incorporar a variabilidade presente nas linhas de produtos de *software*.

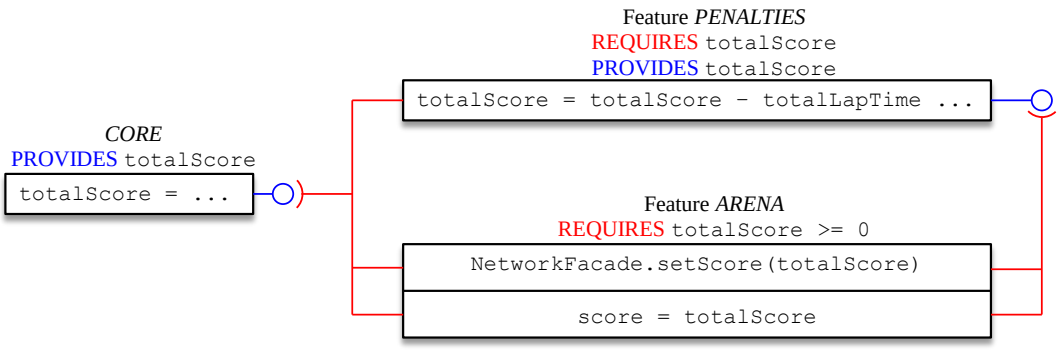


Figura 4.2: Interface emergente com apresentação dos contratos das *features*.

Observa-se que no método `computeLevel` (ver Figura 3.1), existem duas *features* opcionais, tornando possível a geração de quatro produtos diferentes:

- P_1 (nenhuma *feature* selecionada);
- P_2 (apenas a *feature* `PENALTIES` selecionada);

- P_3 (apenas a *feature* `ARENA` selecionada); e
- P_4 (ambas as *features* selecionadas).

Por isso, uma interface emergente com contratos deve levar em consideração todos os quatro produtos possíveis. Cada um desses produtos poderá usar diferentes métodos, e, em consequência, diferentes contratos. Por exemplo a precondição `totalScore >= 0`, aplicável apenas ao ponto de manutenção se `ARENA` $\wedge \neg$`PENALTIES` é verdade, ou seja, a *feature* `ARENA` está selecionada, e a *feature* `PENALTIES` não está, o que ocorre no P_3. No caso de ambas as *features* estarem selecionadas simultaneamente, P_4, a precondição será `totalScore >= totalLapTime * SRC_TIME_MULTIPLIER`, pois a *feature* `PENALTIES` está modificando o valor da variável `totalScore` antes do método `setScore` ser chamado (ver Figura 3.1). Para os produtos P_1 e P_2, o contrato do método `setScore` não se aplica ao ponto de manutenção, uma vez que o código da *feature* `ARENA` não está presente.

Assim, a ideia das interfaces emergentes com contratos é tornar o desenvolvedor ciente dos contratos, possibilitando a redução da introdução de erros e diminuição do esforço necessário para a realização das tarefas, uma vez que, possivelmente, nem as *features* que possuem dependências com a *feature* que está sendo mantida precisariam ser analisadas de forma individual.

Consequentemente, a tarefa de manutenção é agilizada, já que existe menos código para o desenvolvedor analisar. Além disso, existe um aumento na modularização das *features*, pois nem sempre há necessidade de verificar *features* além da qual o ponto de manutenção pertence.

4.2 Computando as interfaces emergentes com contratos

A ideia de como as interfaces emergentes com contratos são computadas e incorporam as necessidades de diferentes *features*, baseia-se no cálculo da precondição mais fraca (*weakest precondition calculos*) [Dijkstra 1976]. De forma simplificada, a precondição do último método do fluxo de dados é utilizada como porta de entrada para inicializar o cálculo de uma precondição geral de determinado trecho de código.

Seguindo tal linha, o cálculo será realizado de baixo para cima, o que quer dizer que ele será iniciado com a última instrução e seu resultado usado para calcular a precondição da penúltima, e assim sucessivamente.

Para melhor entender a computação da interface emergente com contratos utilizando o cálculo da precondição mais fraca, será realizada a computação da interface emergente com contratos do produto P_4, citado acima.

Assumindo que no produto P_4 o último método (ver Figura 4.1) do fluxo de dados, iniciado na variável `totalScore`, é o método `setScore`, temos:

Seja S o método `computeLevel`, composto pelas instruções s_1, s_2 e s_3.

s_1: `totalScore = perfectCurvesCounter * PERFECT_CURVE_BONUES + ...`

s_2: `totalScore = totalScore - totalLapTime * SRC_TIME_MULTIPLIER`

s_3: `NetworkFacade.setScore(totalScore)`

Por definição, a função de cálculo da precondição mais fraca recebe como parâmetros um trecho de código e uma pós-condição desejada, e sua saída é a precondição necessária (ver Seção 2.6). Entretanto, o método `setScore` já possui uma precondição e uma pós-condição estabelecida através de anotações JML no próprio código. Por isso, não é necessário definir uma pós-condição R. Assim, a função wp assume a seguinte forma:

$$wp(s_1, wp(s_2, wp(s_3))$$

Como dito anteriormente, o método `setScore` já possui anotações indicando sua precondição e pós-condição. Por isso, em vez de calcular a precondição, é apenas necessária a recuperação da anotação existente no cabeçalho desse método e aplicá-la no cálculo da próxima precondição. Ao substituir $wp(s_3)$ pela precondição do método `setScore`, tem-se:

$$wp(s_1, wp(s_2, \texttt{totalScore >= 0})$$

Uma vez que temos uma nova pós-condição, podemos aplicá-la no corpo da instrução s_2:

$$wp(s_1, wp(\texttt{totalScore = totalScore - totalLapTime * ... } , \texttt{totalScore >= 0}))$$

$$wp(s_1, \texttt{totalScore - totalLapTime * SRC\_TIME\_MULTIPLIER >= 0})$$

Reorganizando a inequação resultante do cálculo da precondição de s_2, tem-se:

$$wp(s_1, \texttt{totalScore >= totalLapTime * SRC\_TIME\_MULTIPLIER})$$

Por fim, a instrução s_1 é o ponto de manutenção. Por isso, o cálculo da precondição para e a inequação resultante é a precondição mais fraca que precisa ser respeitada para que o método `totalScore` chegue a um estado final válido.

Conclui-se, desta forma, que ao estarem simultaneamente selecionadas, as *features* `PENALTIES` e `ARENA` farão com que a interface emergente com contratos afirme que, para manter o correto funcionamento do software, será necessário que a variável `totalScore` seja sempre maior ou igual ao valor resultante da expressão `totalLapTime`

`* SRC_TIME_MULTIPLIER`. Desse modo, antes da concretização da tarefa de manutenção, o desenvolvedor tem consciência das restrições que envolvem a variável `totalScore`, podendo até verificar se sua modificação pode ferir a restrição existente, antes mesmo da compilação do código. Uma versão simplificada da interface emergente com contratos para esse ponto de manutenção poderia ser expressa da seguinte forma:

"Features `PENALTIES` && `ARENA` **requires** `totalScore >= totalLapTime * SRC_TIME_MULTIPLIER`"

Comparando-se com uma interface emergente convencional, essa seria expressa, de maneira simplificada, da seguinte forma:

"Features `PENALTIES` && `ARENA` **requires** `totalScore`"

Assim, a interface emergente apenas informa ao desenvolvedor da existência das dependências entre o ponto de manutenção e as *features* `ARENA` e `PENALTIES`. Nesse caso, não há informação semântica alguma. Assim, o desenvolvedor precisa investigar o código de ambas as *features* aumentando seu esforço e tempo necessário para a conclusão da tarefa. Além disso, a investigação realizada de forma independente em cada *feature* poderá mascarar a interação existente entre as *features*, acarretando problemas detectáveis apenas quando um produto específico for gerado e testado, ou em uma situação ainda pior, quando em funcionamento.

Capítulo 5

Avaliação

No Capítulo 4 foram apresentadas as interfaces emergentes com contratos e como elas podem ajudar a diminuir o esforço do desenvolvedor e o tempo necessário para a conclusão de tarefas de manutenção, além de prover uma melhor modularização das *features* quando comparada às interfaces emergentes convencionais.

Neste Capítulo, é apresentada uma avaliação das interfaces emergentes com contratos em comparação com as interfaces emergentes, a fim de verificar se as supostas melhorias decorrentes do uso das interfaces emergentes são reais. Para tanto, um experimento controlado foi realizado. No experimento, os participantes executaram tarefas de compreensão de códigos em duas linhas de produtos, utilizando as interfaces emergentes com e sem contratos. Os dados obtidos no experimento foram objetos de estudo e os resultados são apresentados a seguir, bem como as definições e materiais usados no experimento.

Todos os materiais utilizados nesse experimento e informações adicionais a respeito dos resultados obtidos podem ser consultados no site do projeto.[1]

5.1 Objetivos, Perguntas e Métricas

Nesta seção são determinados e explicados os parâmetros utilizados no projeto e análise dos resultados desse experimento, segundo a abordagem *Goal, Questions and Metrics (GQM)* [Basili et al. 1994].

5.1.1 Objetivo (*Goal*)

G1: O objetivo central desta avaliação consiste na comparação direta do número de erros cometidos pelos desenvolvedores, quando estes utilizaram as interfaces emergentes com e sem contratos, além de analisar o tempo necessário na conclusão das tarefas em ambas as técnicas.

[1] https://sites.google.com/a/ic.ufal.br/emergent-contract-interfaces

5.1.2 Perguntas (*Questions*)

Para alcançar o objetivo da avaliação foram realizados os seguintes questionamentos:

Q1: As Interfaces emergentes com contratos diminuem o tempo necessário para o entendimento do código pelos desenvolvedores?

Q2: As Interfaces emergentes com contratos melhoram o entendimento do código pelos desenvolvedores?

5.1.3 Métricas (*Metrics*)

Para responder a estas perguntas, utilizou-se as seguintes métricas:

- *Tempo* que os participantes levaram para concluir corretamente uma tarefa de compreensão que envolve dependências entre *features*;
- *Número de erros* cometidos pelos participantes na tentativa de concluir uma tarefa de compreensão que envolve dependências entre *features*.

5.2 Definição do experimento

Nesta seção são apresentadas as hipóteses que foram verificadas, os materiais utilizados e traçado o perfil dos participantes. De uma forma geral, nesta seção são apresentados os detalhes do planejamento do experimento.

5.2.1 Hipóteses

A seguir, apresenta-se as hipóteses que foram verificadas, com a execução desse experimento, para que fosse possível responder as perguntas feitas anteriormente.

H1: Interfaces emergentes com contratos aceleram o processo de compreensão, quando comparadas às interfaces emergentes:

$$H1_0 : \mu_{TempoComEI} = \mu_{TempoComECI}$$

$$H1_1 : \mu_{TempoComEI} > \mu_{TempoComECI}$$

H2: Interfaces emergentes com contratos diminuem o número de erros cometidos no processo de compreensão, quando comparadas às interfaces emergentes:

$$H2_0 : \mu_{ErrosComEI} = \mu_{ErrosComECI}$$

$$H2_1 : \mu_{ErrosComEI} > \mu_{ErrosComECI}$$

5.2.2 Variáveis

A técnica utilizada (interfaces emergentes com e sem contratos) foi tratada como uma variável independente com dois níveis. Já como variável dependente, teve-se a compreensão do software, baseada nas tarefas, medida em termos de número de erros e tempo de resposta.

Como parâmetros de pertubação (*confounding parameters*), foram identificados três parâmetros, que poderiam influenciar nos resultados obtidos com o experimento. São eles:

(i) Experiência de programação;

(ii) Linguagem de programação; e

(iii) Familiaridade com as ferramentas.

O primeiro parâmetro listado, experiência de programação, é o mais importante, pois influencia diretamente na compreensão do programa. Foi controlado com a aplicação de um questionário para avaliar a experiência dos participantes. Quanto à linguagem de programação, fora controlada utilizando-se apenas a linguagem Java, da qual os participantes detinham conhecimento prévio.

Por fim, foi utilizado o IDE *Eclipse* na realização das tarefas do experimento, que já havia sido usado por todos os participantes, o que permitiu o controle das ferramentas.

5.2.3 Material

Na construção dos cenários das tarefas do experimento, foram selecionadas duas linhas de produtos de software de tamanho médio escritas na linguagem Java. A primeira, trata-se de um jogo de corrida para celulares, com aproximadamente 15 KLOC, chamado *Best Lap*,[2] ilustrado na Figura 5.1.

Figura 5.1: Telas do *Best Lap*.

A segunda linha de produtos, é um jogo de aviões de guerra para celulares da linha Nokia series 60, com aproximadamente 10 KLOC, chamado *Bomber*,[3] ilustrado

[2] http://www.meantime.com.br/
[3] http://j2mebomber.sourceforge.net/

na Figura 5.2. Ambas já foram utilizadas em outras pesquisas [Figueiredo et al. 2008, Rebêlo et al. 2009]. Além dos materiais já citados, uma pequena linha de produtos foi criada para ser usada como parte das tarefas de aquecimento necessárias à compreensão do funcionamento do experimento.

Figura 5.2: Telas do *Bomber*.

5.2.4 Projeto do experimento

Para avaliar as hipóteses, foi utilizado o quadrado latino (*latin square*) [Box et al. 2005] no projeto do experimento. O uso do quadrado latino ajuda a diminuir ruídos, como o aprendizado, uma vez que não permite que um participante repita a execução das tarefas de um mesmo cenário ou com uma mesma técnica.

Devido ao uso de duas técnicas (interfaces emergentes com e sem contratos) foram selecionadas tarefas das duas linhas de produtos (*Best Lap* e *Bomber*).

Conforme demonstrado na Tabela 5.1, os quadrados latinos proporcionam a utilização sem ordem específica de todas as linhas de produtos e técnicas. Foi possível, ainda, com a sua utilização, dividir os participantes em quatro grupos. Cada grupo apresentava diferenças na execução das tarefas. Assim, por exemplo, o grupo A realizou, primeiro, as tarefas do *Best Lap* com interfaces emergentes com contratos, e, em seguida, as tarefas do *Bomber* com interfaces emergentes. Em contrapartida, o grupo D realizou, inicialmente, as tarefas do *Bomber* com interfaces emergentes para, então, executar as tarefas do *Best lap* com interfaces emergentes com contratos.

Adicionalmente, a última coluna da direita na Tabela 5.1 mostra, em média, a experiência de programação dos participantes de cada grupo, de acordo com o questionário de experiência em programação que foi aplicado [Feigenspan et al. 2012]. Os resultados deste questionário oscilam entre 0.727 até 3.635. Portanto, é possível afirmar que os participantes do experimento possuíam níveis de experiência semelhantes.

Grupo	Número de participantes	Tarefas iniciais	Tarefas finais	Média de experiência de programação
A	5	*Best Lap*, ECI	*Bomber*, EI	1.631
B	5	*Best Lap*, EI	*Bomber*, ECI	1.891
C	6	*Bomber*, ECI	*Best Lap*, EI	2.043
D	6	*Bomber*, EI	*Best Lap*, ECI	1.804

Tabela 5.1: Distribuição dos participantes, tarefas e médias da experiência de programação do grupo.

5.2.5 Participantes

Os participantes do experimento foram estudantes da disciplina de linha de produtos da Universidade de Magdeburg, Alemanha, que possuíam, em decorrência do estudo da matéria, conhecimento prévio sobre linhas de produtos, *software* configurável, interfaces emergentes e contratos. Os estudantes foram avisados acerca do experimento e que seu desempenho no experimento não afetaria sua nota na disciplina. Para motivar os estudantes, a participação no experimento dispensava uma lição de casa.

Os estudantes foram divididos de forma aleatória em quatro grupos, conforme anteriormente mostrado na Tabela 5.1.

5.2.6 Tarefas

O experimento foi realizado utilizando-se quatro tarefas, sendo duas por cada linha de produtos. Cada linha de produtos possuía uma tarefa relacionada a apenas uma *feature*, e outra relacionada a duas *features*. Assim, as tarefas executadas no experimento puderam ser divididas em dois conjuntos:

Conjunto 1: tarefas relacionadas a apenas uma *feature*; e

Conjunto 2: tarefas relacionadas a duas *features*.

Nas tarefas apresentadas, os participantes tinham como ponto de manutenção uma variável não inicializada, e, para concluí-la, deveriam identificar os intervalos de valores—com o auxílio de uma calculadora—para os quais a inicialização da variável não ocasionaria erros no software, algo similar ao exemplo apresentado no Capítulo 4.

Para prover um melhor entendimento de como eram as tarefas, a primeira tarefa do *Bomber*, pertencente ao **Conjunto 1**, é descrita a seguir.

Na primeira tarefa do *Bomber*, os participantes deveriam determinar os valores válidos para inicializar a variável `GAME_DEBRIS` na seguinte atribuição:

```
FORCE_ANG = CRATER_SIZE * GAME_DEBRIS
```

Aos participantes que deveriam executar essa tarefa com as interfaces emergentes com contratos, foi mostrada a seguinte interface para o ponto de manutenção:

"Feature *CRATER* **requires** `FORCE_ANG` $\geq$ `0 && FORCE_ANG` $\leq$ `360 * Common.FIXED`."

Ao reorganizar a interface emergente com contratos, tem-se:

$$0 \leq \texttt{FORCE\_ANG} \leq 360 * \texttt{Common.FIXED}$$

Assim, ao substituir na interface emergente com contratos o valor de `Common.FIXED` é obtido o intervalo de valores válidos para a variável `FORCE_ANG`:

$$0 \leq \texttt{FORCE\_ANG} \leq 360 * 1024$$

$$0 \leq \texttt{FORCE\_ANG} \leq 368640$$

A partir dessa informação e do valor da constante `CRATER_SIZE` é possível identificar quais valores de `GAME_DEBRIS` são válidos:

$$0 \leq \texttt{CRATER\_SIZE} * \texttt{GAME\_DEBRIS} \leq 368640$$

$$0 \leq 30 * \texttt{GAME\_DEBRIS} \leq 368640$$

$$0 \leq \texttt{GAME\_DEBRIS} \leq 368640/30$$

$$0 \leq \texttt{GAME\_DEBRIS} \leq 12288$$

Para os participantes que executaram a tarefa com as interface emergentes, a interface apresentada para o ponto de manutenção foi a seguinte:

"Feature `CRATER` **requires** `FORCE_ANG`"

Note que essa interface não traz nenhuma informação a respeito do intervalo de valores válidos, fazendo com que os participantes tivessem que identificar no próprio código as restrições, que eram representadas no código fonte através de instruções de lançamento de exceções, para, só então, identificar corretamente os valores de inicialização da variável `GAME_DEBRIS`.

As demais tarefas foram semelhantes a este modelo apresentado.

As tarefas foram apresentados aos participantes em dois *Eclipses* adaptados, sendo cada eclipse possuidor de um dos cenários. Os *Eclipses* possuíam, pré-instalados, um *plug-in*, com a finalidade de gravar o tempo de execução das tarefas e da quantidade de respostas incorretas submetidas pelos participantes.

Para cada cenário, foram criados 2 documentos, um para as interfaces emergentes com contratos e outro para as interfaces emergentes, que descreviam as tarefas e um passo-a-passo do uso do *plug-in*. Esses documentos foram distribuídos entre os participantes de

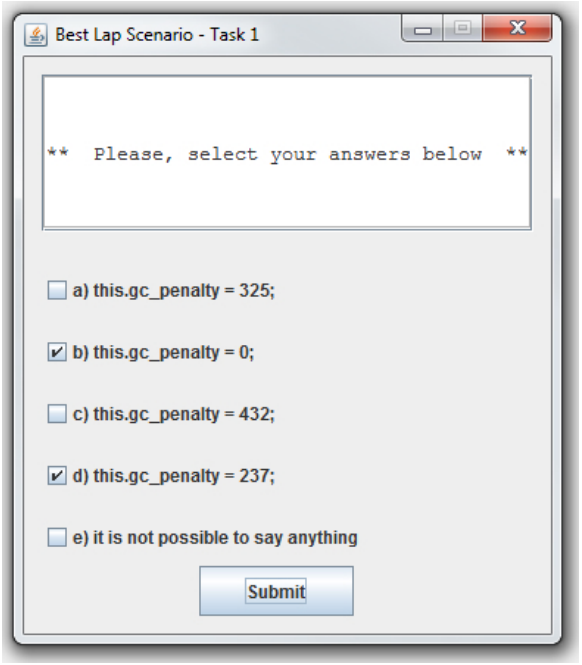


Figura 5.3: *Pop-up* de inserção de respostas.

acordo com os grupos especificados na Tabela 5.1. Assim, por exemplo, os participantes do grupo B receberam a descrição das tarefas do *Best Lap* com a informação das interfaces emergentes, mas receberam a descrição das tarefas do *Bomber* com a informação das interfaces emergentes com contratos.

Após a leitura do documento que descrevia as tarefas, os participantes deviam acionar o botão de *play* do *plug-in*, para iniciar a contagem do tempo. Em seguida, quando o participante acreditasse que tinha concluído a tarefa, deveria acionar o botão *stop*. Assim, o *plug-in* parava de contar o tempo e apresentava um *pop-up* com um conjunto de alternativas para que o participante selecionasse suas respostas entre cinco alternativas, como mostrado na Figura 5.3, quais acreditava estar dentro do intervalo de valores válidos. Os participantes tinham liberdade para marcar quantas alternativas quisessem. Contudo, apenas duas delas estavam corretas, o que não foi dito para os participantes. Em seguida, a resposta do participante era comparada com o gabarito pré-cadastrado para cada tarefa. A resposta apenas era considerada correta quando as duas alternativas válidas eram marcadas exclusivamente e simultaneamente.

Ao informar uma resposta incorreta, o *plug-in* voltava a contar o tempo, incrementando um contador de contagem do número de erros do participante na tarefa, dando a oportunidade de o participante continuar tentando concluir a tarefa, até atingir o acerto, ou, de outro modo, estourar o tempo limite de 60 minutos. Durante a execução do experimento, vale frisar que nenhum dos participantes estourou tal limite.

Ao inserir a resposta correta, o tempo parava de forma permanente, sendo salvo juntamente com a quantidade de erros cometidos em um arquivo. Essas medições foram

realizadas de forma individual para cada tarefa.

Gravou-se, também, a tela dos participantes durante a execução do experimento, o que serviu para observar o comportamento deles durante a execução das tarefas.

Destaca-se ainda que o experimento só foi realizado após os participantes executarem tarefas de aquecimento, em um sistema criado com o intuito de ilustrar o experimento, sem que tais resultados fossem considerados ao final.

5.3 Execução

O experimento foi realizado num laboratório de informática em Dezembro de 2013 em duas etapas realizadas no mesmo dia, pois não haviam máquinas suficientes para todos os participantes. Apesar da apresentação da ideia geral do experimento, os participantes não tiveram conhecimento das hipóteses.

Após as tarefas de aquecimento, os participantes puderam executar as tarefas de fato, e, ao final, cada um respondeu ao questionário, onde foram perguntados sobre a experiência pessoal acerca do tema programação.

Na execução das tarefas, existiram algumas falhas de execução dos participantes, como o esquecimento em dar início à gravação do tempo, o que foi passível de correção por meio do uso de gravações de tela.

Para alguns participantes, no entanto, a gravação de tela não foi realizada corretamente, por isso foram excluídos do experimento, evitando a invalidação dos resultados.

5.4 Resultados e discussão

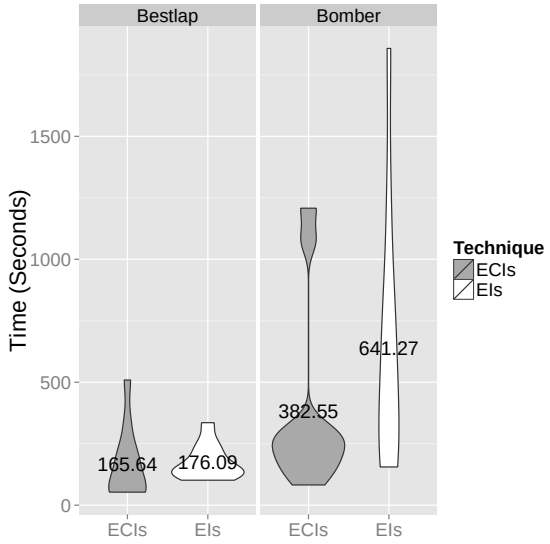
Nesta seção, são apresentados os dados coletados por meio do experimento realizado e, em seguida, são discutidos seus significados e resultados.

5.4.1 Análises dos dados

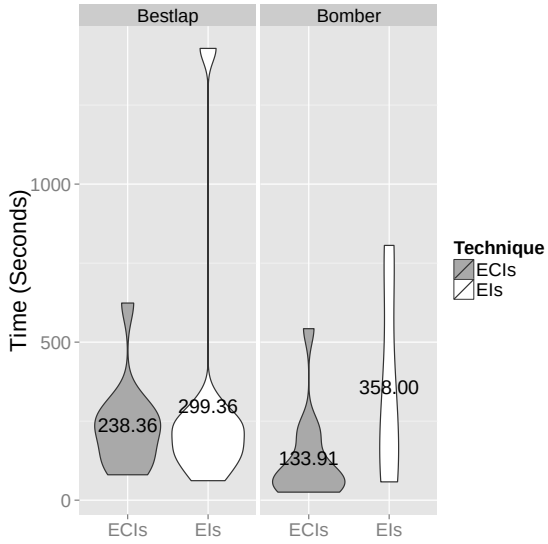
A seguir, são avaliadas as hipóteses levantadas na subseção 5.2.1. A discussão é iniciada avaliando-se a hipótese **H1** e, em seguida, a hipótese **H2**.

Para melhor discussão, optou-se por analisar cada conjunto de tarefas individualmente. Primeiro, realizou-se a análise dos resultados obtidos a partir das tarefas do **Conjunto 1** e depois os resultados das tarefas do **Conjunto 2**.

H1: Interfaces emergentes com contratos aceleram o processo de compreensão, quando comparadas às interfaces emergentes.



(a) Tempo de resposta das tarefas do **Conjunto 1**.



(b) Tempo de resposta das tarefas do **Conjunto 2**.

Figura 5.4: *Violin plots* do tempo de resposta das tarefas dos **Conjuntos 1 e 2**.

A Figura 5.4(a) apresenta as médias de tempo que os participantes necessitaram para concluir as tarefas do **Conjunto 1**, separadas por linha de produto e técnica. No geral, a execução das tarefas com o auxílio das interfaces emergentes com contratos foi, aproximadamente, 1.5 vezes mais rápida, quando comparada com as interfaces emergentes convencionais. Porém, no *Bomber*, existiram diferenças maiores do que no *Best Lap*. Esses resultados levam a acreditar que não apenas a técnica interferiu nos resultados, mas que a linha de produtos também exerceu efeito sobre o tempo de resposta, indicando que as diferenças entre o uso das interfaces com e sem contratos foram mais visíveis nesse cenário.

A Figura 5.4(b), mostra as médias de tempo da execução das tarefas do **Conjunto 2**. Nota-se que as interfaces emergentes com contratos obtiveram, novamente, melhores resultados quando comparadas com as interfaces emergentes convencionais, o que foi possível tornar a execução das tarefas até, aproximadamente, 1.8 vezes mais rápida. Contudo, assim como nas tarefas do **Conjunto 1**, os resultados do **Conjunto 2** também sugerem que a linha de produtos exerceu efeito sobre o tempo de resposta.

Na avaliação das hipóteses, foram conduzidos testes de análise de variância (ANOVA) 2X2 com as duas técnicas (interfaces emergentes com e sem contratos) e as duas linhas de produtos (*Best Lap* e *Bomber*), seguindo a convenção de que um fator é significante quando $p\text{-}value < 0.05$ [Box et al. 2005]. Os resultados da ANOVA do **Conjunto 1** podem ser observados na Figura 5.5(a) e, a partir deles, conclui-se que existe um efeito significativo da linha de produtos, mas não para a técnica. Dessa forma, pode-se concluir que os dados mostram que a técnica não alterou o tempo de resposta. Algo diferente

Factor	df	F	p-value
Técnica	1	1.761	**0.191**
Linha de produtos	1	11.309	0.002
Interface x Linha de produtos	1	1.498	0.228

(a) ANOVA das tarefas do **Conjunto 1**.

Factor	df	F	p-value
Técnica	1	3.176	**0.082**
Linha de produtos	1	0.082	0.776
Interface x Linha de produtos	1	1.039	0.314

(b) ANOVA das tarefas do **Conjunto 2**.

Figura 5.5: ANOVA das tarefas dos **Conjuntos 1 e 2**.

quando analisou-se a linha de produtos. Além disso, os resultados da ANOVA também mostraram que a aplicação de uma determinada técnica em uma das linhas de produtos não apresentou melhores resultados, ao contrário do que a simples análise das médias do tempo de respostas sugeriam.

Quanto as tarefas do **Conjunto 2**, o teste não revelou diferenças significativas, e isso se aplica à técnica, à linha de produtos e combinação de linha de produtos e técnica. Consequentemente, os resultados levaram à rejeição da hipótese alternativa $H1_1$, e confirmação da hipótese nula $H1_0$.

A seguir, será realizada a verificação da hipótese **H2**, onde o número de erros cometidos pelos participantes foram o objeto de estudo.

H2: Interfaces emergentes com contratos diminuem o número de erros cometidos no processo de compreensão, quando comparadas às interfaces emergentes.

Em relação a quantidade de erros cometidos pelos participantes do experimento, a Tabela 5.2 apresenta a quantidade de erros em ambos os conjuntos. Nas tarefas do **Conjunto 1** a diferença na quantidade de erros foi mais expressiva no *Bomber*. Do total de erros cometidos em ambas as linhas de produtos, os erros cometidos com as interfaces emergentes com contratos representaram aproximadamente 33% dos erros, sugerindo que a utilização das interfaces emergentes com contratos, em comparação com as interfaces emergentes convencionais, proporcionaram a diminuição do número de erros nas tarefas desse conjunto.

Nas tarefas do **Conjunto 2**, o *Bomber* apresentou novamente as maiores diferenças. Do total de erros cometidos, os erros cometidos com as interfaces emergentes com contratos representaram aproximadamente 27%, sugerindo, assim, que as interfaces emergentes com contratos apresentaram resultados ainda mais expressivos em tarefas que exigiram a verificação de mais de uma *feature*.

No total, foram cometidos 121 erros na execução das tarefas dos dois conjuntos, dos quais, 85 (70%) foram cometidos com interfaces emergentes e 36 (30%) com interfaces

(a) Tarefas do **Conjunto 1**

Técnica	Linha de produtos		Soma	%
	Best Lap	*Bomber*		
EI	3	35	38	67%
ECI	6	13	19	33%
Soma	9	48	57	100%

(b) Tarefas do **Conjunto 2**

Técnica	Linha de produtos		Soma	%
	Best Lap	*Bomber*		
EI	24	23	47	73%
ECI	16	1	17	27%
Soma	40	24	64	100%

Tabela 5.2: Número de erros por técnica e linha de produtos.

emergentes com contratos.

A partir desses resultados é possível notar uma redução significativa da quantidade de erros cometidos com o uso das interfaces emergentes com contratos.

Contudo, ainda é necessária a aplicação de testes para confirmar a existência de alguma diminuição no número de erros provenientes do uso de interfaces emergentes com contratos. Para isso, foi realizado o teste exato de Fisher (*Fisher's exact test*) [Anderson and Finn 1996]. Os resultados desse teste estão presentes na Tabela 5.3. O teste indica que para as tarefas do **Conjunto 1** e do **Conjunto 2** existem diferenças significativas entre as técnicas. Por isso, conclui-se que interfaces emergentes com contratos auxiliaram os participantes a evitarem erros.

Dessa forma, os resultados levaram à rejeição da hipótese nula $H2_0$ e, em contrapartida, a confirmação da hipótese alternativa $H2_1$.

Conjunto	Z value	p-value
1	1.978	**0.048**
2	3.207	**0.001**

Tabela 5.3: Resultados do teste exato de Fisher para o número de erros.

5.4.2 Discussão

Tomando-se por base a análise dos resultados obtidos e explanados na Seção 5.4.1, pode-se responder aos questionamentos que foram levantados anteriormente.

Q1: *As Interfaces emergentes com contratos diminuem o tempo necessário para o entendimento do código pelos desenvolvedores?*

Nas tarefas do **Conjunto 1** não foram encontradas diferenças estatisticamente significativas, quando esteve sob análise a técnica utilizada. Contudo, o mesmo não se pode afirmar para a linha de produtos. Isso pode ter ocorrido pelo *Bomber*, já que este apresentou-se um pouco mais complexo, em virtude da profundidade da chamada de método que estava envolvida na tarefa, ao passo que a tarefa do *Bomber* possuía profundidade 2, a do *Best Lap* possuía profundidade 1.

Quando foram analisadas as gravações dos participantes, notou-se que, para chegar na primeira chamada de método, eles levaram tempo semelhante em ambas as linhas de produtos, mas alguns participantes tinham dificuldades em perceber que havia mais um nível no *Bomber*.

Para as tarefas do **Conjunto 2**, foram encontradas diferenças estatisticamente pouco significativas entre o tempo de resposta a favor das interfaces emergentes com contratos.

Em média, as interfaces emergentes com contratos proporcionaram aos participantes que as tarefas fossem executadas mais rapidamente, em comparação as interfaces emergentes, e isso em todas as tarefas, principalmente naquelas em que foram utilizadas duas *features*.

Uma explicação para tal fato pode ser a dificuldade que os participantes tiveram quando usaram interfaces emergentes para relacionar as limitações impostas pelas duas *features*, pois, para os participantes com interfaces emergentes com contratos tal relação era mais clara. Parece plausível, portanto, que com um maior número de *features* envolvidas, os benefícios das interfaces emergentes com contratos se tornaram mais aparentes, o que necessita de estudos mais profundos para melhor avaliação.

Estatisticamente, não foi possível rejeitar a hipótese nula $H1_0$. Contudo, a execução das tarefas do **Conjunto 1** com as interfaces emergentes com contratos foi 50% mais rápida do que o tempo que os participantes levaram para concluir as mesmas tarefas com interfaces emergentes. No **Conjunto 2**, esse resultado foi ainda mais expressivo, mostrando que os participantes que usaram interfaces emergentes com contratos foram 80% mais rápidos do que os participantes que usaram interfaces emergentes.

Q2: *As Interfaces emergentes com contratos melhoram o entendimento do código pelos desenvolvedores?*

Ao analisar o quesito quantidade de erros, as interfaces emergentes com contratos apresentaram vantagens, especialmente nas tarefas com duas *features*. Os participantes que usaram interfaces emergentes com contratos cometeram 30% do total de erros das tarefas do **Conjunto 1 e 2**, o que proporcionou a rejeição da hipótese nula $H2_0$ e confirmação da hipótese alternativa $H2_1$.

Um fato interessante percebido por meio das gravações dos participantes que usavam interfaces emergentes, é que estes apenas identificavam a restrição presente em uma das *features*, mas não verificavam a outra *feature*. No entanto, com o uso das interfaces emergentes com contratos esse tipo de erro é minimizado, uma vez que as restrições das *features* envolvidas estão explicitadas na própria interface emergente através dos contratos.

5.5 Ameaças à validade

Nesta seção são apresentados fatores que podem ter influenciado nos resultados obtidos neste trabalho, divididos em ameaças internas e externas. Entre as ameaças internas estão a definição das tarefas e avaliação dos dados. As ameaças externas compreendem a escolha dos participantes e a linguagem de programação usada.

5.5.1 Ameaças internas

Como o *Bomber* teve suas *features* criadas artificialmente, pôde-se escolher trechos de códigos e funcionalidades passíveis de aplicação dos contratos que necessitavam da realização de cálculos para se chegar aos intervalos de valores considerados válidos. Isso pode ter acarretado na necessidade de realização, pelos participantes, de alguns cálculos mais complexos, favorecendo as interfaces emergentes com contratos. Reduziu-se tal risco ao serem selecionados intervalos simples que precisavam apenas de algumas constantes, espalhadas pelo próprio código. A utilização de uma linha de produtos real, *Best Lap*, também auxiliou a minimizar a ameaça. As profundidades das chamadas de métodos também afetaram o desempenho dos participantes, o que foi perceptível nas tarefas com apenas uma *feature* do *Bomber*.

Por fim, em virtude da medição incorreta do tempo de execução de alguns participantes e a impossibilidade de recuperação desse tempo através das gravações, foram necessárias as exclusões de três participantes durante a análise, restando vinte e dois participantes. Mesmo com tais exclusões, não houve impacto nos dados, já que se manteve um número balanceado de participantes entre as linhas de produtos e interfaces.

5.5.2 Ameaças externas

As principais ameaças externas foram: os estudantes e o uso da linguagem de programação Java. Quanto ao uso de estudantes, para minimizar a ameaça, foram escolhidos aqueles que pertenciam a um curso de linhas de produtos de software, os quais possuíam contato com conteúdo teórico e prático. Mesmo sem haver o fator experiência profissional em softwares com variabilidade, existia a experiência básica. No quesito linguagem de programação, a linguagem Java é utilizada na implementação de linhas de

produtos [Alves et al. 2005b, Alves et al. 2005a]. Apesar das tarefas serem consideradas simples e de rápida execução, a utilização dos softwares reais minimiza um pouco a ficção das tarefas.

5.6 Conclusão do experimento

Em geral, as interfaces emergentes com contratos possuem potencial para melhorar a manutenção das linhas de produtos de software, mesmo que apenas uma das hipóteses tenha sido aceita. Ao final da execução do experimento, alguns participantes relataram verbalmente que as interfaces emergentes com contratos ajudaram a concluir as tarefas. É possível que, em linhas de produtos reais com mais de duas *features* envolvidas nas tarefas, as interfaces emergentes com contratos tragam maiores benefícios, uma vez que ajudam o desenvolvedor a relacionar a informação semântica de diversas *features*, tornando-a uma técnica importante para dar suporte ao desenvolvedor.

É necessário, no entanto, maior aprofundamento para compreender como as interfaces emergentes com contratos afetam as tarefas de manutenção da linha de produtos. Especialmente em linhas de produtos maiores e mais realistas.

Capítulo 6

Trabalhos Relacionados

Neste capítulo são apresentados os trabalhos relacionados.

6.1 Interfaces gráficas para linhas de produtos

A ferramenta *Colored IDE* (CIDE), foi desenvolvida para diminuir, através de cores e da *virtual separation of concerns* [Kästner 2010], a poluição no código fonte causada pelo uso das diretivas de pré-processamento. No lugar das diretivas de pré-processamento, são atribuídas cores que distinguem as diferentes *features*, e a ferramenta ainda permite que as *features* que não sejam de interesse do desenvolvedor sejam "escondidas", melhorando, assim, a legibilidade e o entendimento do código [Kästner et al. 2008]. Contudo, não visualizar o código de uma *feature* pode ser perigoso uma vez que aumenta ainda mais a dificuldade para que o desenvolvedor perceba dependências entre *features*. Por isso, é possível que as interfaces emergentes com contratos seja uma forma de complementar a *virtual separation of concerns*, permitindo que o desenvolvedor esconda as *features* que não são de seu interesse e ainda possua informações semânticas sobre as dependências existentes.

6.2 Interfaces para linhas de produtos

As *features* de uma linha de produtos podem ser implementadas através de programação orientada a aspectos, mas devido ao problema da fragilidade dos *pointcuts* [Störzer and Koppen 2004], pesquisadores propuseram a definição de interfaces entre os aspectos e as classes. A proposta das *Crosscutting Programming Interfaces* (XPIs) [Griswold et al. 2006] visa justamente desacoplar os aspectos das classes. Já o *Open Modules* [Aldrich 2005] propõe o uso de interfaces para explicitar os *join points* das classes. Essas abordagens usam instruções especificas das linguagens para estabelecer as interfaces e, por isso, os desenvolvedores são os responsáveis por escrevê-las e mantê-las.

Ao contrário dessas abordagens, as interfaces emergentes e as interfaces emergentes com contratos fornecem uma forma de inferir as interfaces sob demanda, independente da linguagem ou técnica sem a necessidade de que o desenvolvedor escreva e mantenha as interfaces.

6.3 Construção de especificações

As especificações são construídas baseadas apenas na implementação ou baseadas na implementação e outras especificações. As interfaces emergentes com contratos são estabelecidas a partir do uso de especificações baseadas na implementação e outras especificações—*weakest precondition* e JML—. Contudo, estabelecer especificações para um programa pode se tornar uma tarefa exaustiva, trabalhosa e passível de erros. Por isso, uma alternativa é o uso de especificações baseadas apenas na implementação [Flanagan and Leino 2001].

6.4 Contratos para linhas de produtos

Muitos pesquisadores propuseram o uso de contratos para especificar linhas de produtos. Os contratos são usados para realizar verificações no software como, por exemplo: análise estática [Scholz et al. 2011], prova de teoremas [Bruns et al. 2011, Damiani et al. 2012, Thüm et al. 2012] e *runtime assertion checking* [Thüm et al. 2013]. Além disso, os contratos podem facilitar o desenvolvimento em uma linha de produtos [Thüm et al. 2013] e o desenvolvimento entre linhas de produtos [Schröter et al. 2013], servindo como uma interface comportamental entre os artefatos de implementação das linhas de produtos de *software*.

Capítulo 7

Conclusão e Trabalhos Futuros

Este trabalho apresentou as interfaces emergentes com contratos, geradas a partir da utilização do *Design by Contracts* e aplicação da *Weakest Precondition*. As interfaces emergentes com contratos são uma forma de tentar amenizar a baixa expressividade das interfaces emergentes e, consequentemente, diminuir os problemas que essa baixa expressividade ocasiona.

Para investigar as melhorias propostas advindas do uso das interfaces emergentes com contratos, quando em comparação com as interfaces emergentes, foi realizado um experimento controlado com 22 participantes. Os resultados desse experimento não mostraram a existência de melhorias estaticamente significativas na redução do tempo necessário no processo de compreensão de programas, mas foram observadas melhorias de até 80% na velocidade em que as tarefas de compreensão foram concluídas. Quando o objeto de análise foi a quantidade de erros cometidos no processo de compreensão, os resultados indicaram que existem melhorias estatisticamente significativas na diminuição da quantidade de erros cometidos pelos participantes provenientes do uso das interfaces emergentes com contratos. Do total de erros cometidos no experimento, cerca de 30% foram cometidos com a utilização das interfaces emergentes com contratos, ao passo que os 70% restantes foram cometidos com a utilização das interfaces emergentes convencionais.

Além disso, quando o desenvolvedor precisou racionalizar a respeito de mais de uma *feature* ao mesmo tempo, os benefícios das interfaces emergentes com contratos se tornaram mais aparentes.

Reforçando as afirmações feitas acima, alguns participantes relataram que as interfaces emergentes com contratos os ajudaram a concluir as tarefas de maneira mais eficaz.

Por fim, ao observar os resultados, é possível concluir que, com um maior número de *features* relacionadas a uma tarefa de manutenção, as interfaces emergentes com contratos gerem maiores benefícios aos desenvolvedores, uma vez que os dados mostraram que existem dificuldades por parte dos desenvolvedores em relacionar as restrições existentes em diferentes *features*.

7.1 Trabalhos futuros

Nesse contexto, foram apresentados apenas os conceitos iniciais das interfaces emergentes com contratos, através de exemplos simples, e uma avaliação de eventuais melhorias provenientes de seu uso. Desse modo, ao se analisar o conhecimento adquirido pela execução do experimento e de seus resultados, pode-se concluir que alguns dos trabalhos futuros são:

- Estender as interfaces emergentes com contratos para instruções de código mais complexas como, por exemplo, instruções `if`, `for` e `while`. Assim, as interfaces emergentes com contratos estarão se aproximando das tarefas de manutenções realizadas em projetos reais.
- Realizar a replicação do experimento mas, dessa vez, com tarefas que estejam relacionadas com um código mais complexo e um maior número de *features*, uma vez que os resultados sugerem que as interfaces emergentes com contratos são mais úteis em cenários maiores.
- Implementar uma ferramenta que faça uso das interfaces emergentes com contratos para calcular as interfaces para o desenvolvedor. Uma vez que as interfaces emergentes já possuem a ferramenta *Emergo* [Ribeiro et al. 2012], e as interfaces emergentes com contratos são uma extensão das interfaces emergentes convencionais, é natural pensar em estender o *Emergo* para que realize o cálculo das interfaces emergentes com contratos.

7.2 Considerações finais

É necessário frisar que os resultados aqui apresentados foram obtidos a partir de trechos de códigos simplificados. Desta forma, não foi possível realizar uma análise com resultados conclusivos acerca de sistemas maiores e com instruções mais complexas.

Apesar de não ter sido possível confirmar ambas as hipóteses em favor das interfaces emergentes com contratos, os resultados indicam que, de fato, existiram melhorias na velocidade de conclusão das tarefas realizadas e no entendimento do código. Esses resultados são animadores e indicam que o desenvolvimento da ideia das interfaces emergentes com contratos poderá proporcionar, no futuro, resultados melhores.

Além disso, através deste trabalho e com base no experimento realizado, foi possível descobrir que os desenvolvedores possuem dificuldades em relacionar restrições existentes em mais de uma *feature*. Neste ponto, as interfaces emergentes com contratos se mostraram mais úteis. Assim, a execução de um novo experimento em que as tarefas possuam interação entre diferentes *features* poderá confirmar esta hipótese.

Referências Bibliográficas

[Aldrich 2005] Aldrich, J. (2005). Open Modules: Modular Reasoning About Advice. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 144–168, Berlin, Heidelberg. Springer.

[Alves et al. 2005a] Alves, V., Cardim, I., Vital, H., Sampaio, P., Damasceno, A., Borba, P., and Ramalho, G. (2005a). Comparative Analysis of Porting Strategies in J2ME Games. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 123–132, Washington, DC, USA. IEEE.

[Alves et al. 2005b] Alves, V., Matos, P., Cole, L., Borba, P., and Ramalho, G. (2005b). Extracting and Evolving Mobile Games Product Lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 70–81, Berlin, Heidelberg. Springer.

[Anderson and Finn 1996] Anderson, T. and Finn, J. (1996). *The New Statistical Analysis of Data*. Springer, New York, NY, USA.

[Barnett et al. 2011] Barnett, M., Fähndrich, M., Leino, K. R. M., Müller, P., Schulte, W., and Venter, H. (2011). Specification and Verification: The Spec# Experience. *Comm. ACM*, 54:81–91.

[Basili et al. 1994] Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). The goal question metric approach. In *Encyclopedia of Software Engineering*. Wiley.

[Batory 2005] Batory, D. (2005). Feature models, grammars, and propositional formulas. In *Proceedings of the 9th International Conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin, Heidelberg. Springer-Verlag.

[Box et al. 2005] Box, G., Hunter, J., and Hunter, W. (2005). Statistics for experimenters: an introduction to design, data analysis, and model building.

[Bruns et al. 2011] Bruns, D., Klebanov, V., and Schaefer, I. (2011). Verification of Software Product Lines with Delta-Oriented Slicing. In *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*, volume 6528 of *LNCS*, pages 61–75, Berlin, Heidelberg, New York, London. Springer.

[Burdy et al. 2005] Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J., Leavens, G. T., Leino, K. R. M., and Poll, E. (2005). An Overview of JML Tools and Applications. *Int'l J. Software Tools for Technology Transfer (STTT)*, 7(3):212–232.

[Cataldo and Herbsleb 2011] Cataldo, M. and Herbsleb, J. D. (2011). Factors leading to integration failures in global feature-oriented development: An empirical analysis. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 161–170, New York, NY, USA. ACM.

[Cataldo et al. 2009] Cataldo, M., Mockus, A., Roberts, J. A., and Herbsleb, J. D. (2009). Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Trans. Software Engineering (TSE)*, 35(6):864–878.

[Clements and Northrop 2001] Clements, P. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA.

[Damiani et al. 2012] Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E. B., and Yu, I. C. (2012). A Transformational Proof System for Delta-Oriented Programming. In *Proc. Int'l Workshop Formal Methods and Analysis in Software Product Line Engineering (FMSPLE)*, pages 53–60, New York, NY, USA. ACM.

[Dijkstra 1976] Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall PTR, 1st edition.

[Feigenspan et al. 2012] Feigenspan, J., Kästner, C., Liebig, J., Apel, S., and Hanenberg, S. (2012). Measuring Programming Experience. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 73–82, Washington, DC, USA. IEEE.

[Figueiredo et al. 2008] Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., and Dantas, F. (2008). Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 261–270, New York, NY, USA. ACM.

[Flanagan and Leino 2001] Flanagan, C. and Leino, K. R. M. (2001). Houdini, an Annotation Assistant for ESC/Java. In *Proc. Int'l Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity (FME)*, pages 500–517, London, UK. Springer.

[Griswold et al. 2006] Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., and Rajan, H. (2006). Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60.

[Hatcliff et al. 2012] Hatcliff, J., Leavens, G. T., Leino, K. R. M., Müller, P., and Parkinson, M. (2012). Behavioral Interface Specification Languages. *ACM Computing Surveys*, 44(3):16:1–16:58.

[ISO 2011] ISO (2011). *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland.

[Kästner 2010] Kästner, C. (2010). *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg.

[Kästner et al. 2008] Kästner, C., Apel, S., and Kuhlemann, M. (2008). Granularity in Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320, New York, NY, USA. ACM.

[Kästner et al. 2011] Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011). Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824, New York, NY, USA. ACM.

[Kim et al. 2011] Kim, C. H. P., Batory, D., and Khurshid, S. (2011). Reducing Combinatorics in Testing Product Lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 57—68, New York, NY, USA. ACM.

[Leavens et al. 2006] Leavens, G. T., Baker, A. L., and Ruby, C. (2006). Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38.

[Leavens and Cheon 2006] Leavens, G. T. and Cheon, Y. (2006). Design by Contract with JML.

[Liebig et al. 2010] Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. (2010). An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114, Washington, DC, USA. IEEE.

[Meyer 1988] Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition.

[Meyer 1992] Meyer, B. (1992). Applying Design by Contract. *IEEE Computer*, 25(10):40–51.

[Parnas 1972] Parnas, D. L. (1972). On the Criteria to be used in Decomposing Systems into Modules. *Comm. ACM*, 15(12):1053–1058.

[Pohl et al. 2005] Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering : Foundations, Principles and Techniques.* Springer, Berlin, Heidelberg, New York, London.

[Popov 2003] Popov, N. (2003). Verification using weakest precondition strategy. in computer aided verification of information systems – a practical industry oriented approach. In *Proceedings of the Workshop CAVIS'03, 12 th of February, e-Austria Institute.*

[Rebêlo et al. 2009] Rebêlo, H., Lima, R., Cornélio, M., Leavens, G. T., Mota, A., and Oliveira, C. (2009). Optimizing JML Feature Compilation in Ajmlc Using Aspect-Oriented Refactorings. In *Proc. Brazilian Symposium on Programming Languages (SBLP).*

[Ribeiro 2012] Ribeiro, M. (2012). *Emergent Feature Modularization.* PhD thesis, Recife.

[Ribeiro et al. 2014] Ribeiro, M., Borba, P., and Kästner, C. (2014). Feature Maintenance with Emergent Interfaces. In *Proc. Int'l Conf. Software Engineering (ICSE).* ACM. To appear.

[Ribeiro et al. 2010] Ribeiro, M., Pacheco, H., Teixeira, L., and Borba, P. (2010). Emergent Feature Modularization. In *Proc. Int'l Conf. Object-Oriented Programming Systems Languages and Applications Companion (SPLASH)*, pages 11–18, New York, NY, USA. ACM.

[Ribeiro et al. 2011] Ribeiro, M., Queiroz, F., Borba, P., Tolêdo, T., Brabrand, C., and Soares, S. (2011). On the Impact of Feature Dependencies when Maintaining Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 23–32, New York, NY, USA. ACM.

[Ribeiro et al. 2012] Ribeiro, M., Tolêdo, T., Winther, J., Brabrand, C., and Borba, P. (2012). 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. ACM.

[Scholz et al. 2011] Scholz, W., Thüm, T., Apel, S., and Lengauer, C. (2011). Automatic Detection of Feature Interactions using the Java Modeling Language: An Experience Report. In *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*, pages 7:1–7:8, New York, NY, USA. ACM.

[Schröter et al. 2013] Schröter, R., Siegmund, N., and Thüm, T. (2013). Towards Modular Analysis of Multi Product Lines. In *Proc. Int'l Workshop Multi Product Line Engineering (MultiPLE)*, pages 96–99, New York, NY, USA. ACM.

[Störzer and Koppen 2004] Störzer, M. and Koppen, C. (2004). PCDiff: Attacking the Fragile Pointcut Problem. In *Proc. European Interactive Workshop on Aspects in Software (EIWAS)*.

[Thüm et al. 2013] Thüm, T., Apel, S., Zelend, A., Schröter, R., and Möller, B. (2013). Subclack: Feature-Oriented Programming with Behavioral Feature Interfaces. In *Proc. Workshop MechAnisms for SPEcialization, Generalization and inHerItance (MASPEGHI)*, pages 1–8, New York, NY, USA. ACM.

[Thüm et al. 2012] Thüm, T., Schaefer, I., Apel, S., and Hentschel, M. (2012). Family-Based Deductive Verification of Software Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 11–20, New York, NY, USA. ACM.

[Yin et al. 2011] Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., and Bairavasundaram, L. (2011). How Do Fixes Become Bugs? In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 26–36, New York, NY, USA. ACM.