Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

Testes de Design: Uma Abordagem Baseada em Testes para Verificação Automática de Conformidade Estrutural entre Implementação e Regras de Design

João Arthur Brunet Monteiro

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Dalton Serey (Orientador)
Jorge Abrantes (Orientador)

Campina Grande, Paraíba, Brasil

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG

M775t Monteiro, João Arthur Brunet.

Testes de design: uma abordagem baseada em testes para verificação automática de conformidade estrutural entre implementação e regras de design / João Arthur Brunet Monteiro. — Campina Grande, 2010.

91f. : il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.

Referências.

Orientador: Profº. Drº. Dalton Serey, Profº. Drº. Jorge Abrantes.

1. Engenharia de Software. 2. Verificação de Software. 3. Software – Design – Testes. 4. Verificação de Conformidade. I. Título.

CDU – 004.41(043)

"TESTES DE DESIGN: UMA ABORDAGEM BASEADA EM TESTES PARA VERIFICAÇÃO AUTOMÁTICA DE CONFORMIDADE ESTRUTURAL ENTRE IMPLEMENTAÇÃO E REGRAS DE DESIGN"

JOÃO ARTHUR BRUNET MONTEIRO

DISSERTAÇÃO APROVADA EM 26.07.2010

DALTON DARIO SEREY GUERRERO, D.Sc
Orientador(a)

JORGE CESAR ABRANTES DE FIGUEIREDO, D.Sc
Orientador(a)

TIAGO LIMA MASSONI, Dr.
Examinador(a)

MARCO TULIO DE OLIVEIRA VALENTE, Dr.
Examinador(a)

CAMPINA GRANDE - PB

Resumo

Assegurar que um programa está de acordo com sua especificação é um elemento chave na garantia de qualidade de software. Embora haja amplo suporte ferramental para checar se uma implementação está funcionalmente de acordo com seus requisitos, checar se ela está em conformidade com regras de design ainda é uma atividade que por muitas vezes é executada manualmente. Neste trabalho propomos uma técnica que visa automatizar a checagem de conformidade entre regras de design de baixo-nível e implementação. A técnica proposta, intitulada testes de design, permite a verificação de programas baseada em testes. O objetivo da técnica é checar se os programadores estão seguindo as regras de design previamente especificadas. Regras são especificadas como testes, daí o nome *testes de design*. De fato, testes de design são JUnit test cases com uma semântica diferente de testes funcionais. Testes funcionais checam se o software se comporta como esperado quando estimulado por determinadas entradas, ao passo que testes de design checam se o software está sendo construído da maneira esperada. Para dar suporte à abordagem de testes de design, foi desenvolvida uma biblioteca chamada DesignWizard (http://www.designwizard.org). Uma avaliação levando em consideração a usabilidade e a escalabilidade do DesignWizard foi efetuada com o objetivo de mostrar sua viabilidade na checagem de conformidade de grandes projetos. Os resultados do experimento de escalabilidade apontam na direção de que a eficiência não é um problema para a ferramenta e que, de acordo com o experimento de usabilidade, o suporte da ferramenta à escrita de testes de design cumpre com as expectativas dos desenvolvedores.

Abstract

Assuring that a program conforms to its specification is a key concern in software quality assurance. Although there is substantial tool support to check whether an implementation complies to its functional requirements, checking whether it conforms to its design remains as an almost completely manual activity. We propose the concept of design tests, which are test-like programs that automatically check whether an implementation conforms to a specific design rule. Design rules are implemented directly in the target programming language in the form of tests. As a proof of concept, we present DesignWizard, an library developed to support design tests for Java programs as JUnit test cases. We have performed an evaluation to show that DesignWizard's API is easy to use and the tool scales as software grows. To achieve this, we have conducted an experiment to assess the usability of DesignWizard's API by analysing eleven developers on the activity of composing five design tests using DesignWizard's API. Besides that, we have measured the time and memory consumption that DesignWizard takes to apply static analysis on several projects sizes varying between 0.125MB and 46MB. The results of our study show that the time and memory consumption of static analysis performed by DesignWizard has a linear form as the size of the application grows. Besides that, the experiment conducted with the programmers leads us to conclude that DesignWizard's API is easy to use in the sense that it meets to programmers expectations.

Agradecimentos

À Liu, meu melhor samba.

À minha família.

Aos meus amigos.

Aos meus orientadores, Dalton Serey e Jorge Abrantes.

À Aninha.

Aos brasileiros, que pagaram por minha educação.

Aos participantes do experimento executado para avaliação deste trabalho.

Este trabalho foi [parcialmente] apoiado pela CPM Braxis e pela FINEP, através do projeto Design Checker.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contextualização

O tamanho e a complexidade dos sistemas de software nos dias atuais estão entre as principais razões para haver divergências entre o design desejado e a implementação do software. Essas divergências denotam que os desenvolvedores não seguiram as regras de design especificadas ou a documentação das regras está desatualizada. Vários fatores podem ser responsáveis por esse cenário, dentre eles, destacamos:

- falta de documentação das regras de design;
- falta de técnicas que apoiem a verificação de conformidade entre documentação de design e o código implementado.

A documentação do design desejado tem papel fundamental para a construção do software. Embora sua importância seja amplamente divulgada [Som06; FL02], o que acontece de fato é que pouco esforço é alocado para esse aspecto [LSF03]. Na maioria do casos, inicialmente o software é pequeno o suficiente para que o processo de desenvolvimento seja iniciado sem a documentação do design, justificando sua ausência [LM08]. Mesmo quando há um reconhecimento de que modelar e documentar o design é importante para o processo de desenvolvimento, a equipe constrói diagramas que pouco refletem a respeito da implementação real do software [Sin98]. Ao longo do desenvolvimento, à medida que o software necessita de mudanças para atender a novos requisitos, aumenta a possibilidade da

introdução de desvios da especificação. Nesses casos, manter a documentação do design atualizada é uma atividade laboriosa, pois em geral, é executada manualmente.

Esse problema torna-se ainda mais evidente quando é levada em consideração a alta rotatividade entre os membros da equipe de desenvolvimento. Nesse caso, como não há documentação confiável, novos membros tendem a implementar funcionalidades sem conhecer/respeitar as decisões que foram tomadas, mas que não foram documentadas apropriadamente.

A construção do software tende a evoluir mais rápido que a documentação. Nos raros casos em que a implementação reflete a documentação do design, é difícil manter a sincronia entre esses dois artefatos. Estudos empíricos mostram que a atividade de atualização da documentação é comumente negligenciada [LSF03]. Ainda que a documentação esteja atualizada, devido a pressões de entrega em prazos curtos, muitas vezes os desenvolvedores não consultam a documentação do design para verificar se o código adicionado viola ou não suas restrições. Como consequência desse cenário, a implementação do software tende a não seguir as regras de design previamente estabelecidas.

Muitos problemas surgem com as divergências entre design e implementação. Este cenário pode levar o software a um estado de degeneração, tornando cada vez mais difícil o seu reúso, bem como sua manutenção, evolução e entendimento [Par94]. Frederick Brooks [Bro95] cunhou o termo Integridade Conceitual para expressar a uniformidade do entendimento que a equipe possui sobre o software. Embora não seja um termo fisicamente capturável, violações de regras de design podem indicar o comprometimento da integridade conceitual do projeto. Portanto, à medida que a implementação se distancia das regras de design especificadas, pode-se admitir que se reduz a integridade conceitual.

Nesse contexto, verificação de software é de suma importância. Tipicamente, é uma das atividades primordiais aplicadas no processo de qualificação do software. Seu objetivo é assegurar que o mesmo está sendo construído corretamente, ou seja, que a implementação está em conformidade com as suas especificações [TWI05]. Existem várias técnicas para verificação de software. Em particular, a verificação de conformidade entre regras de design e a implementação apresenta-se como uma das atividades mais utilizadas.

O estado-da-arte em verificação de conformidade é avançado. Várias abordagens foram propostas para dar apoio a essa atividade [ACN02; MNS95; SSC96; vDGB05]. Por exemplo,

ArchJava [ACN02] é uma extensão da linguagem Java desenvolvida com intuito de garantir integridade de comunicação em um sistema. Integridade de comunicação é uma propriedade relacionada à comunicação entre componentes. Essa propriedade estabelece que os componentes da implementação somente se comunicam com os componentes aos quais estão diretamente conectados na descrição arquitetural. A descrição das regras é feitas através de conceitos como *portas, componentes e conectores* adicionados à linguagem Java. ArchJava é uma alternativa para checagem de integridade de comunicação. No entanto, a especificação das regras requer o aprendizado de novos conceitos inseridos à linguagem, o que aumenta sua curva de aprendizado. Além disso, o espectro de regras de design checáveis através do ArchJava se resume à integridade de comunicação.

Outra abordagem proposta para verificação de conformidade é o uso de modelos de reflexão [MNS95]. Nessa proposta, são especificados manualmente um modelo de alto-nível do software e seu mapeamento em relação a entidades do código-fonte. A partir dessas informações, uma ferramenta compara o modelo especificado e a implementação do sistema.

Apesar dos resultados promissores provenientes do estado-da-arte, a prática ainda deixa a desejar. Embora alguns trabalhos acadêmicos tenham sido propostos para verificação de conformidade entre design e implementação, a técnica mais utilizada é inspeção manual de código, proposta por Fagan [Fag76]. Esse cenário denota uma distância significativa entre o estado da arte e o estado da prática em verificação de conformidade. Parte dessa distância é causada pela dificuldade imposta pelas abordagens na fase de especificação das regras de design.

A inspeção de código é um tipo de processo manual de revisão de software. Seu objetivo é identificar faltas e desvios de especificação na implementação do mesmo [IEE98]. Tipicamente, uma equipe de desenvolvedores experientes é alocada para efetuar tal atividade. Durante a inspeção, o time de revisão checa o código e a documentação em relação a uma lista de interesses. Essa lista de interesses abrange inúmeros aspectos, tais como, corretude de algoritmos, padrões de codificação, possíveis gargalos de desempenho, qualidade dos comentários etc. Em particular, um dos objetivos da inspeção é verificar se o código produzido está de acordo com o design idealizado, isto é, verificar sua conformidade.

Várias empresas e centros de pesquisa (e. g., HP, IBM, AT&T, Bell-Northern Research) adotam e recomendam o uso de inspeção de código para melhorar a qualidade e a pro-

dutividade de seus respectivos processos de software adotados. Estudos dirigidos por tais organizações foram efetuados para evidenciar os benefícios oriundos da inspeção de código. Por exemplo, Grady e Slack [GS94] reportaram um estudo efetuado na HP, onde um terço dos custos na produção de software são provenientes de retrabalho. Nesse caso, segundo os autores, o uso de inspeção para encontrar desvios de especificação reduz 60% destes custos. Essa redução, para uma empresa do porte da HP, pode traduzir-se em $105 milhões por ano.

No entanto, inspeção manual de software tona-se muito custosa à medida que o software cresce. Para ser efetivo, esse mecanismo necessita de profissionais experientes e qualificados, uma vez que aspectos qualitativos de design requerem um nível de *expertise* avançado para serem detectados. Essa demanda por profissionais qualificados aumenta o custo do processo de inspeção, em virtude deste processo requerer planejamento e reuniões que consomem tempo relativamente grande [Gra97; Rus91]. Além disso, entendemos que, por ser efetuada manualmente, a inspeção do código é extremamente laboriosa e, à medida que o sistema cresce, não escala. Um último, mas não menos importante, aspecto a ser levado em consideração em se tratando do processo de verificação manual de conformidade, é o fato desse mecanismo estar sujeito a erros, uma vez que ele é executado manualmente.

Portanto, a falta de ferramentas de apoio ao processo de inspeção de software para a checagem automática entre implementação e documentação de design é o problema que motiva este trabalho. Ainda, a falta de abordagens que facilitem a especificação de regras de design também será endereçada por este trabalho.

1.2 A Solução Proposta

Para solucionar os problemas supracitados, propomos uma técnica baseada em testes para dar apoio às tarefas de verificação de conformidade. Em particular, o objetivo é automatizar parte do processo de inspeção de código. A abordagem contempla um mecanismo de especificação de regras de design e uma técnica automática de checagem de violações dessas regras na implementação do software.

As regras de design são especificadas na forma de testes, os quais denominamos testes de design [BGF09]. Portanto, cumprem o papel de documentação executável do design. Além disso, são escritas na própria linguagem de programação da aplicação a ser analisada. Fator

este que catalisa a adoção da abordagem, uma vez que não será necessário aprender uma nova linguagem para a especificação das regras, reduzindo assim a sua curva de aprendizado e permitindo que toda a equipe entenda e possa contribuir com a escrita das regras.

Como prova de conceito, desenvolvemos um analisador estático, intitulado DesignWizard, que dá suporte à escrita de testes de design para a linguagem Java. Esse analisador é responsável por extrair informações sobre a estrutura do projeto sob verificação e oferecer uma API de consulta a essas informações que servirá de base para a escrita dos testes de design.

1.3 Avaliação

Três estudos foram realizados para avaliar a abordagem. Esses estudos foram conduzidos para avaliar os seguintes aspectos: A aplicabilidade da abordagem, no que diz respeito à escrita de testes de design, a escalabilidade da estratégia de extração das informações, no tocante ao tempo de extração e consumo de memória e, por último, a eficácia da abordagem na detecção de violações e sua inclusão no processo de desenvolvimento.

Para avaliar a aplicabilidade da abordagem de testes de design, foi conduzido um experimento em que onze desenvolvedores foram orientados a escrever cinco testes utilizando a API do DesignWizard. O experimento consistiu em utilizar o protocolo `Think Aloud` [Cla04] para obtenção de dados a respeito das expectativas, estratégias, dificuldades e objetivos dos desenvolvedores ao utilizarem a API do DesignWizard para a implementação dos testes dos design requeridos. Os resultados desse experimento apontam que os desenvolvedores não tiveram dificuldades em utilizar a API e a consideram, na maioria dos casos, muito fácil de ser manipulada.

Em relação à escalabilidade, conduzimos um experimento para avaliar o desempenho da estratégia de extração de informações adotada. Para tanto, executamos um experimento com o DesignWizard. O objetivo desse experimento foi medir o tempo de extração de informações dos projetos sob verificação e o consumo de memória da ferramenta quando submetida a várias aplicações de tamanhos diferentes. Os resultados apontam para um crescimento linear tanto do tempo de extração quanto do consumo de memória à medida que cresce o tamanho do software sob análise. Os resultados dessa avaliação estão descritos com mais

detalhes no Capítulo 4.

Por último, realizamos dois estudos de caso para avaliar a eficácia da abordagem na detecção de violações e sua inclusão no processo de desenvolvimento.

1.4 Resultados e Contribuições

Em termos concretos, as contribuições deste trabalho são:

- Proposta da abordagem de verificação de conformidade baseada em testes de design. Essa proposta foi publicada no International Conference On Software Engineering (ICSE, 2009) [BGF09]. A abordagem de testes de design obteve o primeiro lugar na competição *Jazoon Rookie* [oJT09], realizada na Suiça. Essa competição avalia trabalhos realizados na linguagem Java e os elenca de acordo com sua relevância e aplicabilidade. Após uma disputa entre dezenas de trabalhos, a abordagem de testes de design obteve o primeiro lugar no concurso.

- Desenvolvimento de uma ferramenta que dá suporte à abordagem de testes de design. A ferramenta, intitulada DesignWizard, está disponível sob licença Lesser General Public License (LGPL) no endereço `http://www.designwizard.org`.

- Aplicação da abordagem de testes de design na verificação de conformidade entre diagramas de classe especificados em UML e implementação [PBR08]. Essa aplicação foi publicada em forma de artigo no Symposium on Applied Computing (SAC, 2008).

- Aplicação da ferramenta DesignWizard no contexto de uma técnica para análise de impacto de mudanças em código-fonte [HGF+08]. O resultado dessa aplicação foi publicado em forma de artigo no International Conference on Computer and Information Science (ICIS, 2008).

- Implantação da ferramenta DesignWizard no processo de desenvolvimento de uma das fábricas de software da CPM Braxis. A ferramenta foi utilizada pela empresa CPM Braxis [CPM10] com intuito aumentar o controle de qualidade de seu processo de desenvolvimento. O DesignWizard foi utilizado no contexto de projetos reais e ajudou

a organização a progredir do nível CMMI 3 para o nível 5, segundo foi declarado pela própria empresa.

1.5 Estrutura do Documento

Essa dissertação está organizada da seguinte maneira. No próximo capítulo são descritos os assuntos teóricos necessários para o entendimento dessa dissertação. Inicialmente, uma discussão sobre Design de Software e Verificação de Software é conduzida. Verificação de Software é a abordagem apresentada como uma maneira de checar se a implementação está de acordo com a especificação de design. Tipicamente, essa especificação é descrita através de regras de design. Por isso, esse assunto também é discutido nesse capítulo. Por último, o capítulo de fundamentação versa sobre a avaliação de usabilidade de APIs. No Capítulo 3, apresentamos a abordagem de testes de design e seu suporte ferramental. Logo após, apresentamos a metodologia utilizada para avaliar o trabalho, bem como os resultados oriundos dessa avaliação. No Capítulo 5 relacionamos e comparamos a abordagem de teste de design com outras abordagens de verificação de conformidade. Por fim, no Capítulo 6, encerramos essa dissertação com algumas discussões a respeito das contribuições deste trabalho.

Capítulo 2

Fundamentação Teórica

Neste capítulo estão presentes os tópicos necessários para o entendimento desta dissertação. A técnica descrita neste documento está inserida no contexto de verificação de conformidade estrutural entre regras de design de baixo nível e implementação. Portanto, faz-se necessária a condução de uma discussão a respeito de verificação de conformidade, apresentando suas definições e diferentes estratégias. Além disso, é necessário dissertar sobre design de software, regras de design e seus níveis. Desse modo, será possível delimitar o escopo em que se insere nossa abordagem, bem como adquirir o embasamento teórico necessário para a leitura dos próximos capítulos.

2.1 Design de Software

Projetar o sistema a ser construído é uma atividade vital para a garantia de qualidade e sucesso na construção de software. No livro intitulado *Software Design* [Bud03], o autor destaca vários benefícios oriundos dessa atividade. Por exemplo, design permite uma avaliação da solução antes dela ser implementada. Isto é possível de ser efetuado pela existência de modelos da solução gerados durante o planejamento. Outro fator importante é que esses modelos gerados durante o processo de design servem como documentação da solução a ser implementada. Ao longo do desenvolvimento da solução, esta documentação é extremamente importante para a verificação de conformidade entre o modelo proposto da solução e a implementação. Por último, mas não menos importante, os modelos gerados pelo processo de design facilitam a comunicação entre a equipe que irá desenvolver o produto. Estes mod-

elos descrevem as principais características da solução, como devem ser implementada e os atores envolvidos nessa implementação.

Segundo Taylor e Van der Hoek, é possível focar em dois aspectos quando leva-se em consideração a palavra design. Design, como um produto e design como um processo[TvdH07]. Como produto, o termo design significa uma representação do software. Um documento ou outra forma de descrever aspectos importantes sobre o produto a ser desenvolvido. Como processo, design denota o ato de construir essa representação.

No contexto desse trabalho, referenciamos design como sendo a representação do software a ser construído. A partir dessa visão, é preciso destacar que essa representação deve descrever diversos aspectos. Entre esses aspectos, estão:

- A estrutura do sistema, incluindo o relacionamento e hierarquia entre seus módulos;
- A forma como os módulos estão agrupados;
- Os algoritmos a serem utilizados;

Dessa forma, entende-se por design a descrição dos elementos estruturais do software e como esses elementos se relacionam entre si obedecendo determinadas restrições. Essa descrição pode ser realizada em níveis diferentes de abstração. De acordo com o Guia para o Corpo de Conhecimento da Engenharia de Software [BDA+99], design de software consiste em dois níveis de atividades: o design de alto nível e o design de baixo nível. Essas duas categorias diferem-se entre si pela diferença do nível de abstração da representação da solução.

No livro Software Design [Bud03], Budgen afirma que o design de alto nível, também referenciado como design arquitetural, está relacionado a aspectos mais gerais da solução adotada. Esse nível se preocupa em como o software será estruturado e como os componentes de alto nível se relacionam. Um exemplo de preocupação do nível arquitetural é a estratégia adotada para a solução do problema. Por exemplo, é nesse nível que está descrito se o software será desenvolvido utilizando técnicas de sistemas distribuídos ou não. O design de alto nível está fortemente conectado aos requisitos não-funcionais, tais como, escalabilidade, desempenho, segurança etc.

Por outro lado, design de baixo nível, também descrito como design detalhado, descreve a estrutura interna dos componentes especificados no design arquitetural. Nesse nível, dentre

vários aspectos, é descrito o modo como as abstrações são implementadas e a comunicação entre os componentes de baixo nível do software.

Uma preocupação característica do design detalhado é descrever aspectos do design no nível de pacotes, classes, métodos, atributos etc. Os relacionamentos entre essas entidades e as restrições impostas a essa comunicação também são especificadas nesse nível.

Design é, em ambos os níveis, importante para o sucesso do desenvolvimento de software. Por esse motivo, é preciso que as regras especificadas durante essa etapa sejam cumpridas na fase de implementação. Esse é o assunto abordado na Seção 2.2.1.

2.2 Verificação de Software

O Guia para o Corpo de Conhecimento da Engenharia de Software [BDA+99] define verificação de software como sendo uma atividade que tem por objetivo assegurar que o software está sendo construído da maneira correta. Ao contrário de validação de software, onde o interesse está focado em assegurar que o software construído está funcionalmente de acordo com o que o cliente deseja (o software correto está sendo construído), verificação está intrinsecamente ligada a como o software está sendo construído e se está de acordo com suas especificações (o software está sendo construído corretamente).

Técnicas para verificação de software compreendem abordagens estáticas e dinâmicas. De acordo com o Corpo de Conhecimento em Engenharia de Software (SWEBOK), abordagens estáticas de verificação de conformidade envolvem a análise de artefatos do projeto sem a necessidade de executá-los. Por exemplo, através da documentação, da extração de informações do código-fonte utilizando análise estática entre outras atividades. Por outro lado, abordagens dinâmicas fazem uso das informações coletadas durante a execução do software para verificar determinadas propriedades.

A atividade de verificação de software é fundamental para a garantia da qualidade de software. Ainda que não esteja diretamente ligada a requisitos funcionais, assegurar que o software está sendo construído corretamente é um processo importante para a construção do software correto, uma vez que seus componentes internamente seguem suas especificações e deverão integrar-se para o cumprimento de um determinado requisito.

2.2.1 Verificação de Conformidade

Verificação de conformidade é um dos tipos de verificação de software. Várias atividades foram propostas para aplicar o processo de verificação de conformidade, isto é, checar se a implementação segue as decisões previamente especificadas. Em linhas gerais, a Figura 2.1 ilustra essa atividade.

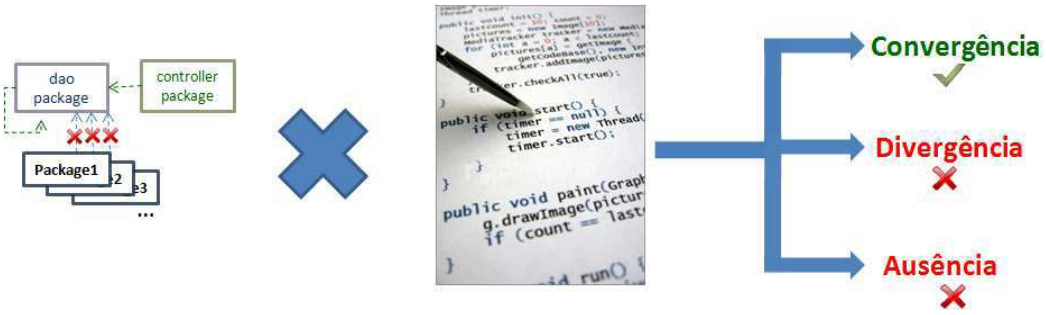


Figura 2.1: Visão geral do processo de verificação de conformidade.

Segundo Knodel e Popesco, o processo consiste em confrontar uma especificação com a sua implementação e obter, como resultado dessa comparação, três categorias bem definidas onde as relações entre os módulos se encaixam [KP07]:

- Convergência;
- Divergência;
- Ausência.

Convergência é a existência de uma relação que foi implementada conforme especificada. Esse cenário indica que a implementação segue o design planejado. Por exemplo:

A classe `Carro` *deve implementar a interface* `Veículo`.

Para haver convergência em relação a essa regra, deve haver no código fonte da aplicação uma classe `Carro` que implementa uma interface `Veículo`.

Divergência diz respeito à existência de uma relação entre dois componentes que não é permitida. Divergência indica que a implementação não obedece as regras de design previamente especificadas. Por exemplo,

Não é permitido que entidades do pacote P se comuniquem com o pacote Q.

Neste caso, se entidades do pacote P se comunicarem com entidades do pacote Q, será identificada uma divergência. Ou seja, uma violação de um aspecto de design previamente estabelecido.

Por último, ausência diz respeito à relação entre dois componentes que foi planejada, porém não implementada. Ou seja, assim como na divergência, a implementação não está seguindo as regras de design. Por exemplo,

O método `A.run()` *da classe* `A` *deve obrigatoriamente invocar o método* `B.start()` *da classe* `B`.

Se na implementação o método `A.run()` não invocar o método `B.start()`, haverá a ausência de um aspecto de design planejado.

A identificação dessas três categorias pode ser feita manualmente ou não. Por exemplo, inspeção manual de código, proposta por Fagan [Fag76], consiste em analisar o código manualmente a fim de encontrar desvios de especificação. Existem outros tantos tipos de verificação de conformidade manual. Por exemplo, em metodologia ágeis, a prática de programação em pares é utilizada para aumentar a confiança de que o código sendo desenvolvido não viola restrições previamente especificadas.

Em contraste com inspeção manual de código e outras atividades manuais de verificação, existem muitas abordagens que visam automatizar parte ou totalmente esse processo. Murphy et al. [MNS95], por exemplo, propuseram uma abordagem intitulada modelos de reflexão que visa manter a sincronia entre modelos de alto-nível e o código-fonte de um sistema. Nessa abordagem, a especificação do modelo de referência e seu mapeamento para o código são feitos manualmente. A automação está na fase de extração do modelo realmente implementado, que é feito atráves de análise estática. Com posse do modelo de referência e da implementação é possível destacar divergências entre o que foi previamente especificado e o que foi implementado de fato.

ArchJava [ACN02] é outro exemplo de abordagem automática para verificação de conformidade. Trata-se de uma extensão da linguagem Java para descrição de restrições arquiteturais e posterior verificação de conformidade entre a implementação e essas restrições previamente especificadas. Essa extensão adiciona conceitos de portas e conexões para que

o projetista descreva os componentes e as interações entre esses componentes usando a linguagem. A ideia central do trabalho é checar a integridade de comunicação entre dois componentes do sistema. Integridade de comunicação é uma propriedade que estabelece que componentes devem se comunicar apenas se eles estiverem conectados na descrição da arquitetura. Ao contrário da abordagem de modelos de reflexão, ArchJava captura as violações em tempo de execução, isto é, dinamicamente.

Muitos trabalhos foram propostos para verificação automática de conformidade entre design e implementação. No Capítulo 5 esses trabalhos são discutidos com maiores detalhes. A análise desses trabalhos levou Knodel e Popesco a identificarem três principais tipos de abordagens para verificação de conformidade [KP07]:

- Modelos de Reflexão;
- Regras de Conformidade entre Relações;
- Regras de Acesso a Componentes.

Modelos de Reflexão, como dito anteriormente, comparam dois modelos de um mesmo software. Tipicamente, aborda-se a comparação entre o modelo arquitetural e o código-fonte. Essa comparação requer o mapeamento manual entre os dois modelos. Através desse mapeamento é possível identificar divergências, ausências e convergências entre as duas representações.

Regras de Conformidade entre Relações está relacionada a especificação de regras permissivas e proibitivas no que diz respeito à comunicação entre os componentes de um sistema. Essa abordagem assemelha-se a Modelos de Reflexão no sentido de que identificam os mesmos defeitos. No entanto, o mapeamento entre os dois modelos é feito de maneira automática na abordagem de Regras de Conformidade entre Relações.

Por último, Regras de Acesso a Componentes permitem a especificação de portas para conexão de um componente. Essa técnica permite o encapsulamento de informações em um nível não suportado pela linguagem de programação. ArchJava [ACN02], por exemplo, estende a linguagem Java para adicionar o conceito de portas e conexões aos componentes. Regras de Acesso a Componentes assemelham-se a Regras de Conformidade entre Relações no que diz respeito a especificação de restrições de comunicação. No entanto, Regras de

Acesso a Componentes faz menção a um único componente. Ou seja, especificam as portas de entrada e saída de um componentes e, portanto, não são flexíveis o bastante para a especificação de uma regra tal como:

Apenas o componente `A` *pode acessar o componente* `B` *através do método* `A.m()`.

A técnica proposta nesse documento assemelha-se a estratégia de **Regras de Conformidade entre Relações** por não requerer o mapeamento manual e por ser flexível a ponto de possibilitar a expressão de regras entre dois componentes específicos. A adoção dessa estratégia nos permitiu flexibilizar a construção de regras de design. Desse modo, as regras não precisam ser gerais.

2.3 Regras de Design

No livro "Software Architecture in Practice" [BCK03], os autores descrevem como é composta a estrutura de um software levando em consideração três aspectos: Módulos, Componentes e Conectores e Alocação. O entendimento desses aspectos é importante para a delimitação do escopo das regras de baixo nível endereçadas pela abordagem de testes de design descrita nesse documento. Por esse motivo, faz-se necessária uma discussão sobre esses aspectos.

Módulos Nessa visão, os elementos do software são vistos como módulos. Módulos representam entidades presentes na implementação (código-fonte). O foco dessa visão está na estrutura do software, e não no comportamento durante a sua execução. O interesse é identificar o modo como os módulos estão relacionados levando em consideração relações de herança, implementação e uso.

Componentes e Conectores O foco dessa visão está no comportamento observado durante a execução do software. Questões relativas a como os objetos são armazenados em tempo de execução, quais partes do sistema são replicáveis e como os dados trafegam no sistema são exemplos de objetos de interesse dessa visão.

Alocação Estruturas de alocação referem-se a relação entre elementos do software e um ou mais ambientes externos em que o software é criado e executado. Por exemplo, qual elemento do software é executado em determinado processador ou em que arquivo está cada armazenado cada elemento do software.

Dentre esses aspectos, as regras de design de baixo nível endereçadas pela abordagem proposta neste documento estão relacionadas aos Módulos. Primeiramente, porque as visões de Componentes e Conectores e Alocação são baseadas em aspectos dinâmicos do software. Como nossa abordagem faz uso de análise estática, ou seja, em tempo de compilação, esses aspectos não são capturados. Por essa razão, iremos discorrer uma discussão mais detalhada sobre a divisão de um software em estrutura modular e como essa estrutura está relacionada as regras de design de baixo-nível. Estruturas baseadas em módulos podem ser:

Decomposição Refere-se ao caso em que o relacionamento entre entidades do código é instanciado através de uma relação "é submódulo de". Essa relação indica que uma entidade é composta por um conjunto de outras entidades, ou seja, pode ser decomposta em outras entidades. Por exemplo, um pacote é composto por um conjunto de classes e interfaces.

Uso Refere-se ao caso em que o relacionamento entre entidades do código é instanciado através de uma relação "usa". Neste caso, para projetos que são implementados em linguagens orientadas a objeto, essa relação se dá através de chamadas de métodos e acesso a atributos. Por exemplo, uma classe `A` usa uma classe `B` se um de seus métodos acessa atributos de `B` ou invoca métodos de `B`. É importante destacar que essa relação é transferida através da decomposição. Por exemplo, se uma classe `A` de um pacote `P` usa uma classe `B` de um pacote `Q`, o pacote `P` usa o pacote `Q`.

Camadas Quando a relação de uso é cuidadosamente controlada e há uma separação de preocupações, surgem no sistema camadas. Essas camadas são porções do código da aplicação que lidam com um funcionalidade específica. Organizar a aplicação em uma estrutura de camadas impõe que uma camada N apenas use serviços da camada N-1. De fato, essa estrutura é um combinação de regras Decomposição e Uso, pois uma camada nada mais é que um conjunto de entidades da aplicação, o que pode ser feito atráves da relação de decomposição. Já a relação interação entre camadas é facilmente mapeada para a relação de

uso.

Generalização Refere-se ao caso em que o relacionamento entre entidades do código é instanciado através de uma relação "estende". Usualmente, essa relação se dá entre duas classes presentes no código. Por exemplo, a classe `LinkedHashSet` estende a classe `HashSet`.

No contexto de nossa abordagem, por ser baseada em análise estática, o software é modelado tendo como norte a visão de Módulos. Ainda, o escopo das regras de design de baixo nível está delimitado pelos quatro aspectos acima. Mais especificamente, com testes de design, é possível construir regras de design que referenciem Pacotes, Classes, Métodos e Atributos e contemplam os aspectos supracitados. Isto é, as relações entre essas entidades estão no escopo de Decomposição, Uso, Camadas e Generalização. É importante destacar que é possível construir regras de baixo nível que seja baseada em mais de um aspecto. Por exemplo:

A Classe `CalculadoraCientifica` deve estender `Calculadora` e conter o método `multiplica()`, que não pode invocar o método `soma()`.

Como podemos notar, essa regra é baseada em **Generalização** (`CalculadoraCientifica` deve estender `Calculadora`), **Decomposição** (`CalculadoraCientifica` deve conter o método `multiplica()`) e **Uso** (o método `multiplica()` não pode invocar o método `soma()`).

2.4 Avaliação de Usabilidade de API

O trabalho descrito nessa dissertação exige uma avaliação que explore aspectos de usabilidade de interfaces programáveis de aplicações (API). Para tanto, foi realizado um experimento baseado em estudos realizados pela Microsoft [Cla04; Cla05; Cla03] e que foram reproduzidos por outros pesquisadores [SC07b; BMS+08; SM08; SC07a; ESM07a] de reconhecida competência nessa área. Em termos gerais, esses trabalhos avaliam a usabilidade de uma API baseados na ideia de que a comparação entre o que os desenvolvedores esperam e o que a API provê é que se faz interessante observar na avaliação da usabilidade de uma API [Cla04].

Nesse contexto, para que a leitura e o entendimento do Capítulo 4 sejam mais facilmente alcançados, é necessária a discussão do protocolo utilizado para observar e avaliar o comportamento dos programadores diante da API.

O protocolo *Think Aloud* [Lew82] é o padrão utilizado para coletar dados durante testes de usabilidade de interfaces de sistemas em geral. Esse protocolo consiste em observar participantes durante a execução de um experimento. Os participantes, além de observados por câmeras e gravadores de áudio, são orientados a verbalizarem suas expectativas, estratégias, dificuldades e objetivos. A partir dessa verbalização, o condutor do experimento é capaz de coletar dados que evidenciam dificuldades e facilidades encontradas pelo desenvolvedor durante a execução de uma determinada tarefa na interface sendo avaliada.

O uso do protocolo *Think Aloud* já está consolidado como metodologia para avaliar usabilidade de interfaces, sobretudo para interfaces gráficas. No entanto, em se tratando de interfaces programáveis, o ambiente difere em vários aspectos do ambiente gráfico. Por este motivo, pesquisadores conduziram estudos para adequar o protocolo *Think Aloud* para o ambiente de interfaces programáveis [Cla04]. Embora seja recente essa adequação, a comunidade científica vem consolidando o seu uso para a avaliação de APIs. Como prova disso, vários trabalhos [SC07b; ESM07b; ESM07a; SC07a; SM08] relatam o sucesso da aplicação do procolo *Think Aloud* para APIs. Por exemplo, Ellis et al. [ESM07a] utilizaram o protocolo para comparar a criação de objetos utilizando o padrão Factory e utilizando construtores. Outro exemplo a ser destacado, é o estudo realizado por Stylos e Clarke [SC07a] para avaliar o impacto de construtores com parâmetro na usabilidade da API.

O *Think Aloud Protocol For APIs*, por ser baseado no protocolo *Think Aloud*, conduz os participantes de um experimento a verbalizarem suas expectativas, estratégias, dificuldades e objetivos. No caso da avaliação de APIs, os desenvolvedores participantes do experimento devem externar, por exemplo, a dificuldade em encontrar um determinado método, ou a facilidade encontrada para mapear um conceito do domínio do problema para uma abstração presente no código.

Uma etapa fundamental da extensão do protocolo *Think Aloud* para APIs consiste em orientar os desenvolvedores a escreverem as tarefas primeiramente em pseudocódigo e, em um segundo momento, transcreverem o pseudocódigo para código real utilizando a API sob avaliação. Esse cenário é uma inversão do que comumente é utilizado. Normalmente, os

estudos de usabilidade primeiro orientam os participantes a utilizarem a ferramenta sobre avaliação para então conseguirem extrair dados sobre o que os desenvolvedores pensam a respeito da interface. Na inversão, primeiro é mapeado o modelo mental do desenvolvedor no que diz respeito às suas expectativas em relação a API e é feita uma análise de quão próxima está a API das expectativas dos desenvolvedores.

2.5 Considerações Finais

Tendo como viés o contexto de verificação de conformidade e todo o conhecimento supracitado, testes de design, a abordagem descrita neste documento, tem como base uma estratégia de **verificação estática** de conformidade entre **regras de design de baixo-nível** e implementação. Essa verificação estática assemelha-se à estratégia de **Regras de Conformidade entre Relações** por não requerer o mapeamento manual e por ser flexível a ponto de possibilitar a expressão de regras entre dois componentes específicos. Ainda, testes de design especificam regras de design de baixo nível que são baseadas em uma **visão modular** da aplicação, focando nos aspectos de **Decomposição, Uso, Camadas e Generalização.**

Capítulo 3

Testes de Design

Neste trabalho apresentamos uma abordagem baseada em testes que visa automatizar o processo de verificação de conformidade estrutural entre regras de design de baixo nível e implementação. O ponto principal da abordagem é a construção de um teste que, ao invés de checar se a funcionalidade do código está de acordo com a especificação, checa se sua estrutura está em conformidade com regras de design previamente especificadas. A esses testes damos o nome de testes de design [BGF09].

É importante destacar que teste de design é um tipo de teste que se baseia em propriedades estruturais do código para checar se uma regra especificada programaticamente está sendo violada ou não pela implementação. Enquanto testes funcionais checam se uma implementação está funcionalmente de acordo com sua especificação, isto é, se o resultado de uma execução desta implementação para uma determinada entrada gera a resposta esperada, testes de design checam se a implementação está de acordo com regras estruturais pré-estabelecidas. Em suma, ao contrário de testes funcionais que checam **o que** o software faz, testes de design checam **como** o software está sendo implementado.

3.1 Conceito de Testes de Design

Testes de design são testes automáticos para verificação de conformidade entre regras de design de baixo nível e implementação. Um teste de design é a especificação executável de uma regra de design de baixo nível. Três características fundamentais de testes de design são:

- testes de design são especificados na mesma linguagem de programação da aplicação sob verificação;
- testes de design são executados automaticamente;
- testes de design checam como o software está sendo construído.

O primeiro aspecto está relacionado à facilidade de se especificar um teste de design. Já o segundo aspecto, relaciona-se ao fato de a etapa de verificação ser automatizada. Por último, é destacado o aspecto que diferencia testes de design de testes funcionais. Isto é, ao invés de checar o que o software faz, testes de design checam como o software está sendo construído.

Testes de design podem descrever vários aspectos de design desejáveis na estrutura do código. Tipicamente, a maioria dos projetos de software define regras de design para controlar sua estrutura. Essas regras são guias e/ou contratos que devem ser seguidos pelos desenvolvedores durante a fase de implementação para que seja mantida a integridade conceitual da equipe em relação ao projeto. Um exemplo comumente encontrado de regra de design é a restrição de comunicação entre determinados componentes do sistema para evitar acoplamentos desnecessários. Uma instância clássica dessa restrição é a aplicação do padrão arquitetural MVC, que visa desacoplar a camada de apresentação da camada de lógica de negócio de uma aplicação. Em uma versão simplificada, para fins didáticos, teríamos a seguinte restrição de design estabelecida pelo padrão MVC:

- As entidades do pacote `apresentacao` não podem ter acesso direto às entidades do pacote `dados`.

Esse é um caso simples de uma regra de design de baixo-nível que pode ser especificada e checada facilmente utilizando testes de design. O Pseudocódigo 1 implementa essa regra de design na forma de um teste.

PseudoCódigo 1 Teste de design para restrição de comunicação entre dois pacotes.

```
1 apresentacao = getPacote("apresentacao")
2 dados = getPacote("dados")
3 pacotesAcessados = apresentacao.getPacotesAcessados()
4 assertFalse ( pacotesAcessados.contains(dados) )
```

A ideia consiste em obter informações a respeito da estrutura do código e checar se essa estrutura possui uma determinada propriedade. Para esse exemplo, o teste consiste em obter os pacotes que são acessados pelo pacote `apresentacao` (linha 3) e verificar se esse conjunto não contém o pacote `dados` (linha 4).

Suponhamos que o código de uma Calculadora eletrônica tenha sido escrito conforme o Pseudocódigo 2. Com a execução do teste de design implementado pelo Pseudocódigo 1 é possível verificar automaticamente se a implementação da Calculadora eletrônica segue ou não a regra especificada pelo teste.

PseudoCódigo 2 Trecho da implementação de uma Calculadora Eletrônica.

```
 1 package apresentacao;
 2 class InterfaceGrafica {
 3   soma() {
 4     Dados.getValores();
 5     Main.setSoma(valor);
 6   }

 7 package dados;
 8 class Dados {
 9   setSoma(int) {
10      ...
11   }
12 }

13 package main;
14 class Main {
15   soma() {
16     Dados.setSoma(valor)
17   }
18 }
```

Com intuito de esclarecer o funcionamento de um teste de design, vamos apresentar como se dá sua interpretação passo a passo. A Tabela 3.1 ilustra essa execução, destacando os valores que as variáveis assumem à medida que o código do teste é executado.

Na primeira linha, o teste busca uma referência ao pacote `apresentacao` da Calculadora eletrônica. Na linha seguinte, o mesmo é feito para o pacote `dados`. Na terceira linha, existe uma chamada ao método `getPacotesAcessados`, que irá retornar todos os pacotes que são acessados por entidades do pacote de apresentação. Os pacotes retornados são `main` e `dados`, uma vez que há chamadas (linhas 4 e 5 do Pseudocódigo 2) para entidades desses pacotes dentro do pacote `apresentacao`.

Tabela 3.1: Execução passo a passo do teste de design implementado pelo Pseudocódigo 1.

Linha executada	**apresentacao**	**dados**	**pacotesAcessados**
Linha 1	apresentacao	null	null
Linha 2	apresentacao	dados	null
Linha 3	apresentacao	dados	{main, dados}

Na fase de asserção (linha 4), o teste checa se o conjunto `pacotesAcessados` não contém o pacote `dados`. Nesse momento, o teste irá falhar, pois como é possível verificar na Tabela 3.1, o conjunto `pacotesAcessados` contém, além do pacote `main`, o pacote `dados`. A Figura 3.1 é uma representação visual da implementação da Calculadora eletrônica expressa pelo Pseudocódigo 2. O destaque em vermelho indica a relação indesejada de design que é identificada pelo teste.

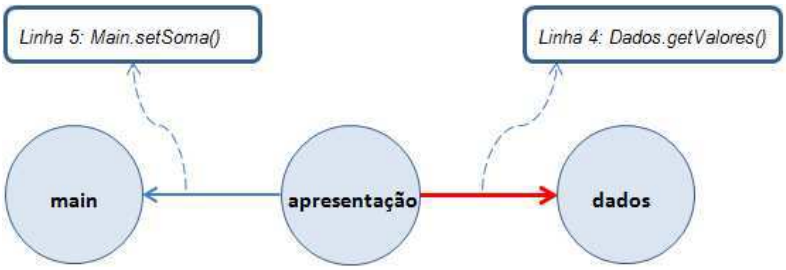


Figura 3.1: Representação visual para o Pseudocódigo 2.

Para estar de acordo com a regra de design estabelecida, a chamada `Dados.getValores()` deve ser retirada do código da Calculadora eletrônica. Assim, seria observado um cenário em que o conjunto `pacotesAcessados` contém apenas o pacote `main`. Como consequência desse cenário, a fase de asserção revelaria que a implementação está em conformidade com o teste especificado.

Como é possível perceber, um teste de design baseia-se na estrutura do código a ser analisado (`getPacote()`, `getPacotesAcessados()` - linhas 1, 2 e 3). Para ter acesso a tais informações, a abordagem proposta deve fazer uso de um analisador estático que tem por objetivo extrair informações a respeito da estrutura da aplicação sob análise e prover uma API para que essas informações sejam recuperadas. Além disso, por se tratar de um teste, teste de design possui asserções (e. g. `assertFalse pacotesAcessados.contains(dados)`). Nesse caso, a abordagem requer o uso de um *framework* de testes que irá prover as rotinas de asserção e um meio automatizado para execução e publicação dos resultados dos testes. Esses dois componentes, analisador estático e *framework* de testes, são discutidos com mais detalhes na Seção 3.4.

3.2 Escrita de Testes de Design

Uma regra de design pode ser expressa por uma fórmula em lógica de primeira ordem sobre os domínios expressos por um modelo de design. Isso significa dizer que um teste de design é um programa que especifica uma regra e verifica se, dada uma implementação como entrada, a regra é satisfeita ou não. Os domínios que são expressos pelo modelo de design referem-se às entidades e relacionamentos presentes no código. Uma vez que testes de design especificam regras estruturais, é preciso que esses testes sejam escritos referenciando essas entidades e relacionamentos.

Em nossa abordagem, as regras são escritas na linguagem Java. Por esse motivo, precisamos estabelecer um metamodelo de design que capturasse aspectos fundamentais da estrutura de um programa escrito nessa linguagem. A Figura 3.2 ilustra o metamodelo de design idealizado. Como pode ser visto, existem sete entidades: *Atributo*, *Método*, *Tipo*, *Classe*, *Interface*, *Enum* e *Pacote*. Há também relações definidas para essas entidades[1]. Por exemplo, um pacote pode conter uma classe e, por consequência, uma classe é declarada em um pacote. Esse metamodelo é utilizado para servir como base na atividade de modelagem da estrutura do código a ser verificado. Através dessa modelagem, é possível checar se a implementação está de acordo com regras de design que se baseiam em propriedades estru-

[1] A explicação detalhada sobre todas as relações entre as entidades está disponível no Apêndice A.

turais da aplicação sob verificação. Por exemplo, é possível checar se uma classe pertence ou não a um determinado pacote.

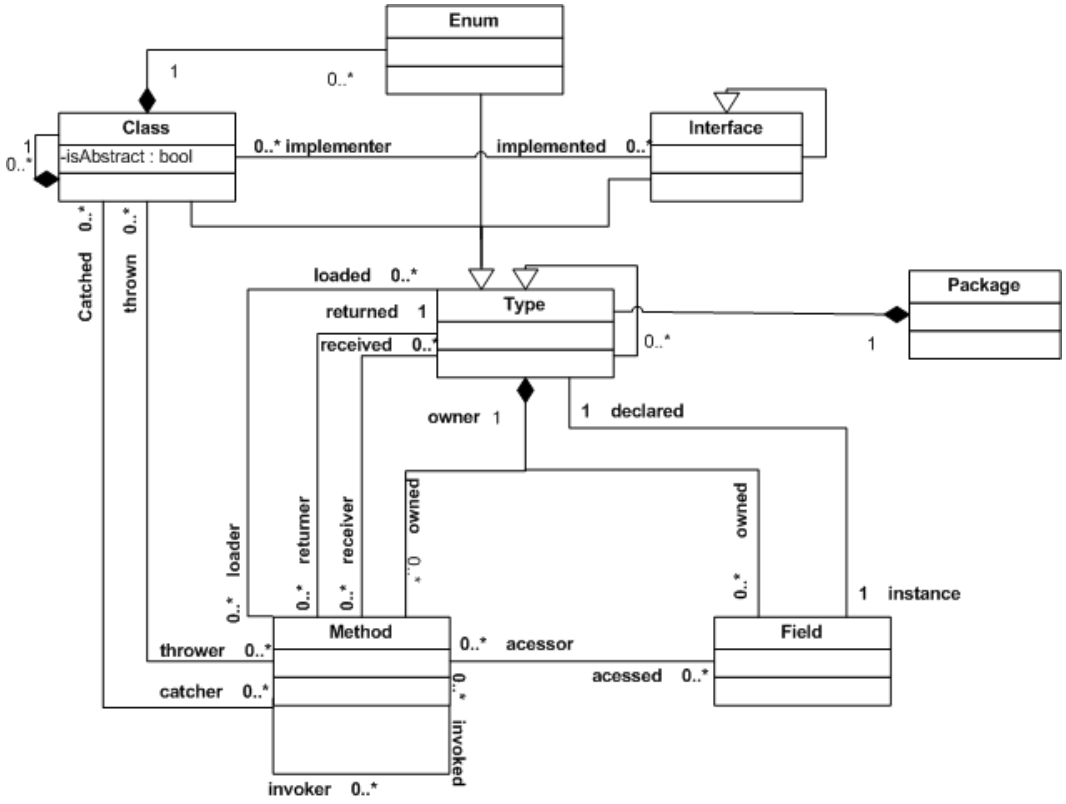


Figura 3.2: Metamodelo de Design para a Linguagem Java.

Por se tratar de uma fórmula em lógica de primeira ordem, regras de design podem conter: quantificadores, operadores lógicos e predicados sobre os domínios expressos pelo metamodelo de design ilustrado pela Figura 3.2. Dado que testes de design são programas, a própria linguagem de programação é utilizada para a descrição de quantificadores e operadores lógicos. Por outro lado, predicados podem ser construídos com auxílio de uma API que disponibilize uma maneira de recuperar relacionamentos entre as entidades do código. Podemos caracterizar essa API como uma forma de navegar por uma estrutura baseada no metamodelo de design e recuperar as informações da aplicação sob verificação.

Para tanto, essa API deve conter os seguintes serviços:

- getAtributo(“String nomeDoAtributo”) : Atributo
- getMetodo(“String nomeDoMetodo”) : Metodo

- getClasse(“String nomeDaClasse”) : Classe
- getInterface(“String nomeDaInterface”) : Interface
- getEnum(“String nomeDoEnum”) : Enum
- getPacote(“String nomeDoPacote”) : Pacote

Esses são os métodos necessários para ter acesso às entidades do código. Através do retorno desses métodos (Atributo, Metodo, Classe, Pacote, Interface e Enum) é que se tem acesso às relações entre as entidades. Por exemplo, para obter as classes que são declaradas dentro de um pacote, a abstração `Pacote` deve conter o seguinte método:

- getClasses() : Set<Classe>

Esse método retorna o conjunto de classes que pertencem a um determinado pacote. Note que, embora não seja explícito na assinatura do método, ele retorna todas as classes que fazem parte de uma relação do tipo:

Pacote *contém* Classe.

Além de obter informações sobre a aplicação, testes de design são escritos utilizando operadores lógicos e quantificadores, uma vez que regras de design tipicamente baseiam-se nessas estruturas. Por exemplo:

Toda classe presente no pacote `command` deve estender a classe `Command`.

Em termos de lógica de primeira ordem, essa regra é especificada da seguinte maneira:

$$\forall c \; contem(command, c) \Rightarrow herda(c, Command)$$

Para essa regra de design, o teste deve ser especificado baseado no quantificador universal (para-todo). Como testes de design são escritos na mesma linguagem de programação da aplicação sob verificação, a própria estrutura da linguagem é utilizada para o uso de quantificadores. O Pseudocódigo 3 é a implementação do teste de design para a regra acima.

Analisando o Pseudocódigo 3 é possível notar que o método `getClasses()` (linha 2) da abstração `Pacote` foi utilizado para ter acesso ao conjunto de classes que são declaradas

PseudoCódigo 3 Exemplo de teste de design com quantificador universal.

```
1 pacoteCommand = getPacote("command")
2 Set classes = pacoteCommand.getClasses()
3 FOR classe in classes DO
4   assertTrue classe.extends(getClasse("Command"))
5 DONE
```

no pacote `command`. Além disso, é possível verificar que o comando de iteração `FOR` (linha 3) foi utilizado para expressar o quantificador universal. Isto é, para a especificação do teste não é necessário adicionar nenhum conceito à linguagem de programação. Esse é um ponto chave na especificação de regras através de testes de design.

Além do quantificador universal, regras de design comumente baseiam-se no quantificador existencial. Por exemplo:

Algum dos métodos da classe `Command` deve se chamar `execute()` ou `wait()`.

PseudoCódigo 4 Exemplo de teste de design com quantificador existencial.

```
1 classeCommand = getClasse("Command")
2 assertTrue ( classeCommand.contemMetodo("execute()") ||
    classeCommand.contemMetodo("wait()") )
```

O Pseudocódigo 4 é a implementação do teste de design para a regra acima. Nesse caso, o método `contemMetodo()` está sendo utilizado para verificar a existência do método `execute()`. Novamente, a própria linguagem é utilizada para especificação do quantificador, pois não há inserção de comandos ou sintaxe adicional, nem adição de conceitos à linguagem, para que a escrita do teste seja realizada.

É possível também notar que, para fazer asserções a respeito da estrutura do código, testes de design utilizam rotinas assertivas e operadores lógicos. Por exemplo, a linha 2 do Pseudocódigo 4 evidencia o uso de uma rotina do tipo `assertTrue`, combinada ao uso do operador lógico “ou”. Os operadores lógicos utilizados na especificação do teste são provenientes da linguagem de programação. Portanto, regras de design que necessitam

da utilização de operadores lógicos são especificadas utilizando os próprios operadores da linguagem de programação sendo utilizada.

Por último, as rotinas de asserção (e. g., `assertTrue`) devem ser utilizadas através de *frameworks* de testes. Por exemplo, se a escrita da regra for realizada na linguagem Java, o *framework* JUnit se adequa facilmente à abordagem, pois possui rotinas que permitem validar expressões *booleanas*.

Em resumo, a escrita de testes de design baseia-se em dois componentes: uma API para prover informação a respeito da aplicação sob verificação e um *framework* de testes para prover rotinas de asserção. Testes de design também são escritos utilizando quantificadores e operadores lógicos. No entanto, essas estruturas são recursos provenientes da própria linguagem de programação utilizada para compor os testes.

3.3 Processo de Verificação de Conformidade Utilizando Testes de Design

Nesta seção apresentamos o modo como é conduzida a verificação de propriedades estruturais através de testes de design. Além disso, apresentamos uma visão geral das atividades envolvidas no processo de verificação.

Quando um teste de design é executado, um grafo baseado no metamodelo ilustrado na Figura 3.2 é construído para modelar a estrutura da aplicação sob verificação. Por exemplo, para o código Java implementado pelo Código 1, é construído o grafo ilustrado pela Figura 3.3. Nele, é possível observar dez entidades que compõem o design do Código 1: `numbers`, `Integer`, `Number`, `Comparable`, `String`, `stringValue`, `Float`, `floatValue()`, `Converter` e `convert()`. As relações entre essas entidades são ilustradas pelas arestas. Por exemplo, as arestas entre `Integer` e `Number` indicam que a classe `Integer` estende a classe `Number` e implementa a interface `Comparable`. Além disso, a classe `Integer` possui um método `floatValue()` que, em sua implementação, invoca o método `convert()` da classe `Converter`.

Quando o teste é executado e o grafo é montado, a verificação é realizada percorrendo esse grafo e analisando suas entidades e relações. O ato de percorrer o grafo é feito através dos métodos da API que fornece informações a respeito da entidades. Por exemplo, para

Código 1 Exemplo de código escrito em Java.

```
1 package numbers;
2 public class Integer extends Number implements Comparable {
3   private String stringValue;
4   public Float floatValue() {
5      return Converter.convert("1");
6   }
7 }
```

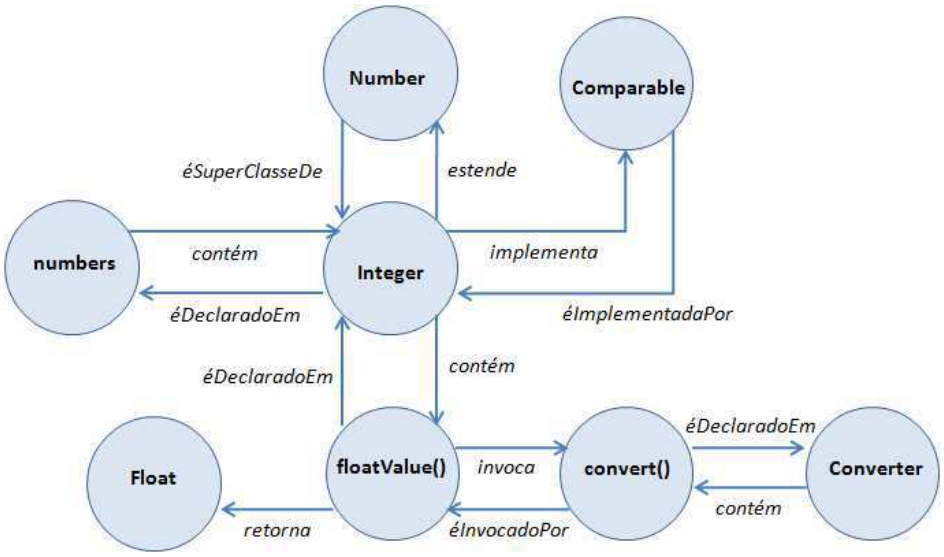


Figura 3.3: Grafo construído durante a execução de um teste de design.

verificar se o método `floatValue()` usa o método `convert()`, o teste de design seria especificado pelo Pseudocódigo 5.

PseudoCódigo 5 Exemplo de teste de design.

```
1 metodoFloat = getMetodo("floatValue()")
2 metodoConvert = getMetodo("convert()")
3 Set chamadosPorFloat = metodoFloat.getMetodosInvocados()
4 assertTrue chamadosPorFloat.contains(metodoConvert)
```

Nesse teste, o método utilizado para navegar no grafo e recuperar os métodos invocados por `floatValue()` é o método `getMetodosInvocados()`. Para o Código 1, isso significa dizer que o conjunto será formado pelo método `convert()`, como é destacado pela Figura 3.4.

Por fim, a asserção é feita verificando a presença dessa entidade no conjunto `chamadosPorFloat` (linha 4). Nesse momento, o teste revelaria a convergência entre a implementação e a especificação da regra.

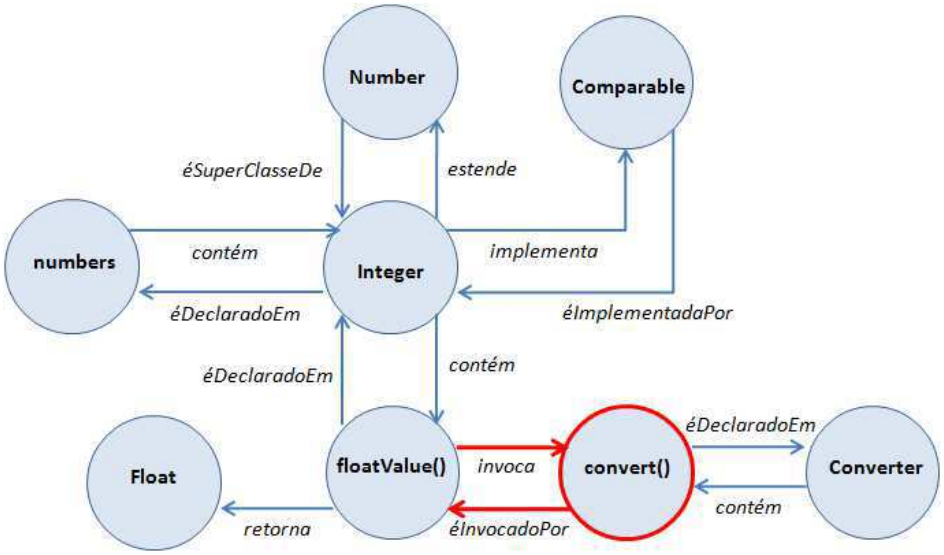


Figura 3.4: Destaque das entidades recuperadas pela API.

A escrita e execução de um teste de design estão inseridos no contexto de um processo que idealizamos para endereçar a verificação de propriedades estruturais. A Figura 3.5 ilustra uma visão geral do processo de testar o design de uma aplicação utilizando testes de design.

Esse processo é dividido em três fases: A fase de concepção das regras design, a fase de especificação dos testes e a fase de checagem, em que os testes são executados.

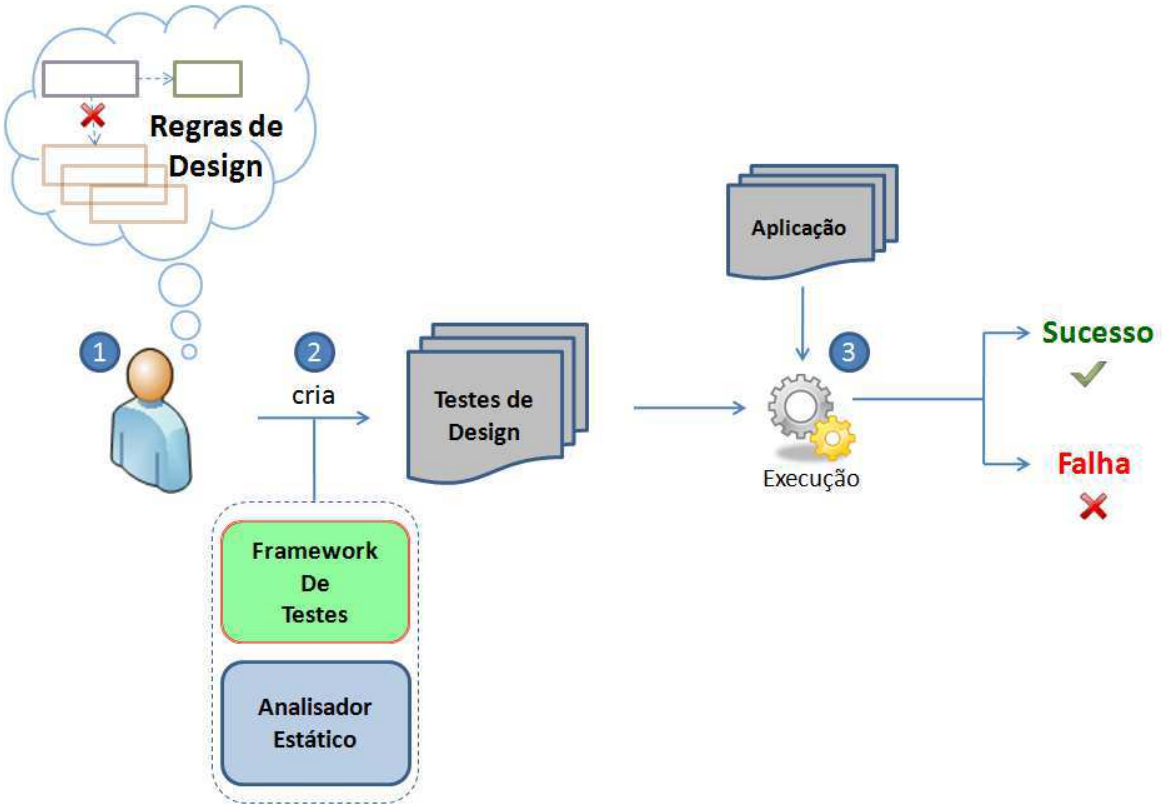


Figura 3.5: Visão Geral do Processo de Teste de Design.

Na primeira fase, o projetista do sistema pensa nas regras de design que devem ser obedecidas durante a implementação do software. A concepção dessas regras de design pode ser resultado de reuniões entre o projetista e a equipe de desenvolvimento, pode ser aspectos de design que um projetista experiente deseja que esteja na implementação etc. As regras concebidas nessa fase expressam como as entidades do sistema devem estar organizadas internamente. O ponto importante a se destacar é que não importa como as regras de design concebidas são documentas (diagramas, documentos etc), elas irão ser implementadas em forma de testes executáveis, mais precisamente, testes de design.

Seguindo o fluxo da Figura 3.5, na segunda fase, o projetista irá implementar as regras discutidas em forma de testes de design, que se comportam como especificações executáveis das regras. Para tal atividade, é preciso fazer uso dos dois componentes presentes

na Figura 3.5, um analisador estático que disponibilize uma API de acesso às informações da aplicação sob verificação e um *framework* de testes. O resultado dessa fase é o teste de design que especifica as regras a serem observadas na implementação do software.

Uma vez construídos os testes, na fase seguinte ocorre a execução dos mesmos. Nessa fase, os testes recebem como entrada a aplicação a ser a analisada e confrontam sua estrutura com as regras de design especificadas. O resultado da execução dos testes de design possui uma semântica diferente da execução de um teste funcional. Uma falha na asserção indica que a implementação não está em conformidade com as regas de design especificadas no teste. Por outro lado, uma execução bem sucedida denota que as regras especificadas são observadas na implementação.

Na próxima seção apresentamos o suporte ferramental construído para dar suporte à escrita e execução de testes de design.

3.4 Ferramental de Suporte a Testes de Design

De acordo com a Figura 3.5, para dar suporte a abordagem de testes de design, são necessários dois componentes: um **analisador estático** e um ***framework* de testes**. O analisador estático é responsável por:

- extrair e modelar informações sobre a estrutura do código;
- expor uma API com métodos de consulta sobre as entidades do código.

Já o *framework* de testes é responsável por:

- prover rotinas de asserção;
- prover uma infra-estrutura para execução automática dos testes;
- reportar os resultados da execução do teste.

O analisador estático é o componente que recebe a aplicação a ser verificada e extrai informações a respeito de sua estrutura. Essas informações devem ser modeladas em um grafo em que as entidades são os vértices e as arestas são as relações entre as entidades. Por

fim, o analisador estático deve prover uma API com métodos que expõem as informações extraídas e modeladas. A Figura 3.6 ilustra esse processo.

Nesta seção, apresentamos o DesignWizard [2] - o analisador estático desenvolvido para dar suporte a testes de design para a linguagem Java. Em linhas gerais, DesignWizard é um analisador estático que que extrai informações de *bytecode* Java, modela essas informações em um conjunto de entidades e suas respectivas relações e provê uma API de consulta a esse modelo.

Figura 3.6: Visão geral sobre o papel de um analisador estático.

A extração de informações desempenhada pelo DesignWizard é realizada através de um arquivo *Jar* da aplicação sendo verificada. Para tanto, o DesignWizard aplica análise estática utilizando o ASM [BLC02] - um manipulador de bytecode Java. A partir de então, as entidades e relações extraídas são modeladas em um grafo baseado no metamodelo de design ilustrado na Figura 3.2.

Com relação ao *framework de testes*, em nosso suporte ferramental utilizamos o *framework JUnit* [GB99]. Essa escolha baseou-se no fato de o JUnit ser um *framework* bastante difundido na comunidade.

Desta forma, em nossa implementação da abordagem, o design/programador irá utilizar a API do DesignWizard e o framework JUnit para especificação e execução dos testes de design. O testes serão construídos utilizando a API do DesignWizard para especificar as propriedades estruturais, enquanto que o framework JUnit será utilizado para a escrita das asserções e para a execução dos testes.

Com o intuito de demonstrar como se dá a escrita de testes de design utilizando o DesignWizard e o JUnit, vamos apresentar um exemplo clássico de regra estabelecida para controlar

[2] DesignWizard é um projeto *OpenSource* e é distribuído sob licença LGPL, podendo ser adquirido no site www.designwizard.org.

a estrutura do código durante sua evolução. Essa regra estabelece que um projeto não deve conter dependência cíclica entre seus pacotes. Isto é, se dados dois pacotes, eles não dependem um do outro mutuamente. Assegurar que a aplicação não contém dependência cíclica é de extrema importância para sua evolução, uma vez que dependência cíclica dificulta o entendimento, a modularização e a modificabilidade do software. Essa regra pode ser expressa através da seguinte fórmula lógica:

$$\forall p, q \; acessa(p, q) \Rightarrow \sim acessa(q, p)$$

A regra define que não pode haver quaisquer dois pares de pacotes que dependem mutuamente um do outro. O Pseudocódigo 6 é um exemplo de código que viola essa regra, pois há dependência cíclica entre os pacotes `apresentacao` e `dados`. Essa dependência é ilustrada pela Figura 3.7.

PseudoCódigo 6 Implementação que viola a regra de dependência cíclica.

```
 1 package apresentacao;
 2 class InterfaceGrafica {
 3   printSoma() {
 4     Dados.calculaSoma();
 5     print resultado
 6   }
 7   recebeDados() {
 8     // recebe dados de entrada
 9   }

10 package dados;
11 class Dados {
12   calculaSoma() {
13     // calcula
14   }
15   validaEntrada() {
16     InterfaceGrafica.recebeDados();
17     // valida dados
18   }
19 }
```

Figura 3.7: Dependência cíclica entre pacotes.

O Código 2 é a implementação em Java, utilizando DesignWizard e JUnit, de um teste de design que verifica a existência de dependência cíclica entre pacotes de uma aplicação Java. O primeiro ponto a ser destacado é que a classe `DesignTest` estende a classe `TestCase` do JUnit (linha 1). De fato, um teste de design é um caso de teste JUnit que checa propriedades estruturais do código.

Código 2 Teste de design para dependência cíclica entre pacotes utilizando o DesignWizard e JUnit.

```
1   public class DesignTest extends TestCase {
2      public void testCyclicDependences() throws IOException {
3         DesignWizard dw = new DesignWizard("project.jar");
4         Collection<PackageNode> allPackages = dw.getAllPackages();
5         for (PackageNode packageNode : allPackages) {
6            Set<PackageNode> callersPackages = packageNode.
               getCallersPackages();
7            for (PackageNode caller : callersPackages) {
8               assertFalse(caller.getCallersPackages().
                  contains(packageNode));
9            }
10        }
11     }
12  }
```

A extração das informações a respeito da estrutura do código é executada na criação de um objeto da classe DesignWizard (linha 3) que recebe como parâmetro o local onde se encontra o arquivo JAR do projeto a ser verificado. A partir deste momento, o DesignWizard extrai e modela as informações de forma transparente ao projetista que está construindo o

teste.

A API provê vários métodos para consulta de informações a respeito das entidades modeladas. Alguns desses métodos [3] podem ser vistos no Código 2. São eles:

- `getAllPackages()`: retorna todos os pacotes de um dado projeto;
- `getCallersPackages()`: retorna todos os pacotes que acessam determinado pacote.

Por fim, o teste de design utiliza a rotina de asserção do JUnit (`assertFalse`) para assegurar que os pacotes que acessam um determinado pacote não são acessados por ele.

Uma vez especificado o teste, basta apenas executá-lo de maneira automática, haja vista que casos de teste JUnit são automaticamente executáveis. A Figura 3.8 ilustra o resultado da execução do teste de design especificado pelo Código 2. Como pode ser observado, utilizamos a mesma infra-estrutura do JUnit para executar e reportar os resultados de maneira automática. No caso demostrado pela Figura 3.8, a aplicação sob verificação não obedece a regra de design especificada pelo teste, uma vez que o *framework* JUnit apresentou uma barra vermelha após a execução do teste de design. Para testes que asseguram que a implementação está de acordo com a regra de design especificada, uma barra verde é apresentada para ilustrar o sucesso na verificação.

A API do DesignWizard provê diversos métodos para que as informações sobre as entidades (Pacotes, Enums, Interfaces, Classes, Métodos e Atributos) e seus relacionamentos sejam facilmente adquiridas. No próximo capítulo, apresentaremos a avaliação conduzida no sentido de averiguar a usabilidade da API no que diz repeito à presença de métodos que satisfazem às expectativas dos programadores na atividade de construção de testes de design.

3.5 Discussão Sobre Testes de Design

Através da abordagem de teste de design é possível especificar regras de design na mesma linguagem de programação usada para escrever o código funcional. Esse fator pode se tornar extremamente atrativo para a adoção da abordagem, uma vez que não será necessário aprender uma nova linguagem para a especificação das regras, reduzindo assim a sua curva de

[3] A documentação completa da API está disponível em www.designwizard.org

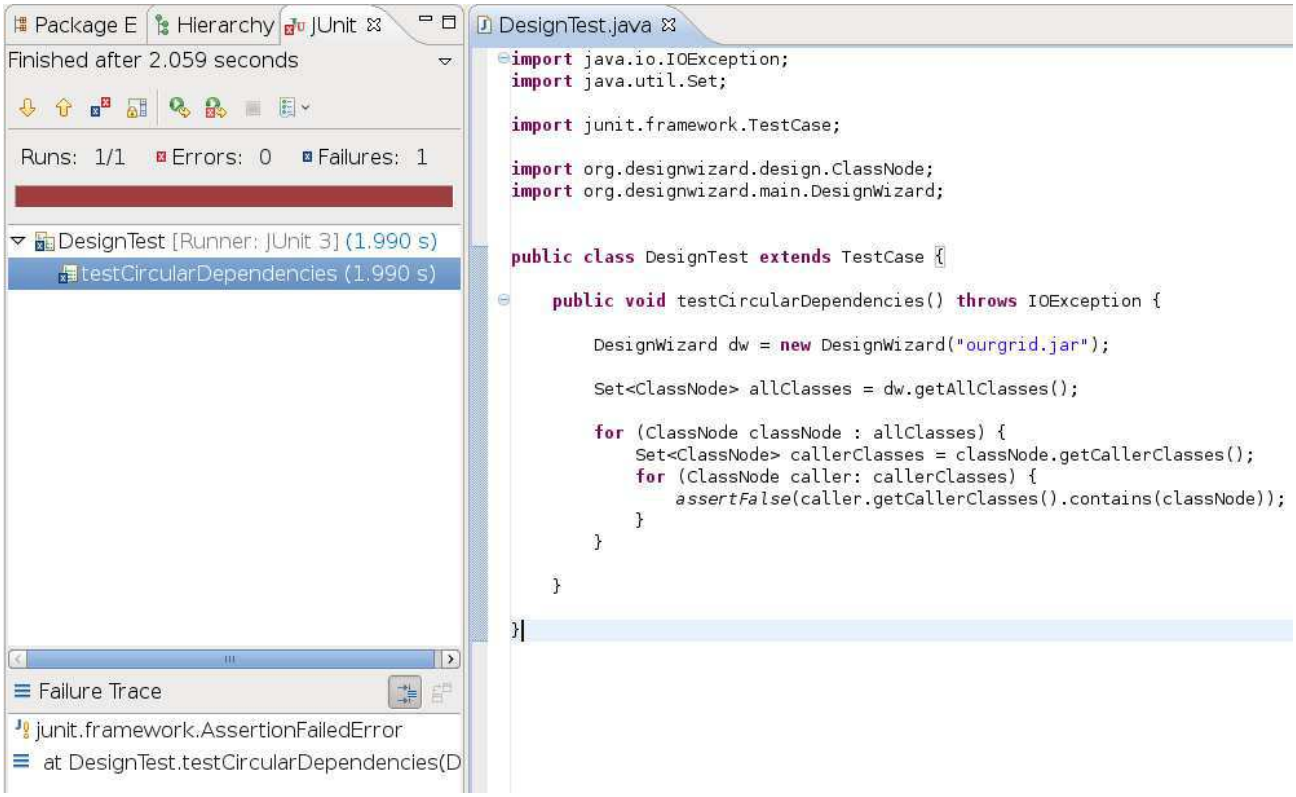


Figura 3.8: Execução de um teste de design.

aprendizado e permitindo que toda a equipe entenda e contribua com as regras de design implementadas. Ainda, testes de design são executados automaticamente. Uma vez escritos, o custo de se efetuar a verificação é mínimo.

Em se tratando de processos de desenvolvimento que adotam testes para garantir a qualidade do código desenvolvido, a abordagem de teste de design pode ser incluída sem acarretar grandes impactos na condução desses processos. Por se tratar de testes que são executados automaticamente, os testes de design podem ser adicionados à suite de testes funcionais. Assim como os programadores verificam frequentemente se a implementação está funcionalmente correta, com testes de design as violações de regras de design são identificadas no momento em que são adicionadas. Além disso, se houver gerenciamento de *commits* no sentido de prevenir inclusão de código no repositório sem que ele esteja passando nos testes, ainda que os testes funcionais apontem para o funcionamento correto do software, é possível assegurar que o seu *design* não está sendo modificado sem levar em consideração as regras de

design. Com isto, é mais fácil manter a integridade conceitual entre os membros da equipe, uma vez que eles possuem um mecanismo automático para checar se as mudanças por eles implementadas violam o design previamente especificado.

Capítulo 4

Avaliação

Neste capítulo é descrita a metodologia utilizada para avaliar o conceito de testes de design, bem com os resultados provenientes dessa avaliação. Para isso, foram realizados três estudos: i) o de usabilidade do conceito de testes de design, através do experimento para avaliar a usabilidade da API do DesignWizard; ii) o da escalabilidade do DesignWizard, em termos de tempo e memória necessários para extração de informações de projetos de dimensões realistas e; iii) dois estudos de caso para avaliar a eficácia da abordagem na detecção de violações e sua inclusão no processo de desenvolvimento.

4.1 Usabilidade da API do DesignWizard

Com intuito de avaliar a usabilidade da API do DesignWizard, foi conduzido um experimento baseado nos estudos realizados pela Microsoft [Cla04; Cla05; Cla03] e que foram reproduzidos por outros pesquisadores [SC07b; BMS+08; SM08; SC07a; ESM07a] de reconhecida competência nessa área.

Na próxima subseção, será descrita a metodologia adotada para a avaliação da usabilidade da API do DesignWizard. O objetivo foi executar um experimento baseado no conhecimento difundido pelos estudos supracitados. Este experimento visou demonstrar que a API do DesignWizard é intuitiva e simples de ser utilizada.

4.1.1 Metodologia

Visão Geral

Segundo Steven Clarke, a comparação entre o que os desenvolvedores esperam e o que a API provê é o que se faz interessante observar na avaliação da usabilidade de uma API [Cla04]. Seguindo tal linha de raciocínio, a metodologia utilizada consistiu em fornecer testes de design realistas para que desenvolvedores os implementassem, inicialmente, em pseudocódigo. O pseudocódigo, nesse contexto, representa o que os desenvolvedores esperam que exista na API para que eles possam implementar os testes de design. Após essa primeira fase, foi indicado aos desenvolvedores que implementassem os mesmos testes de design utilizando o DesignWizard e o JUnit. Todas as tarefas foram distribuídas na forma de um projeto no Eclipse. Os participantes tinham acesso à Internet e à documentação da API do DesignWizard. Durante essa etapa, foi utilizado o protocolo *Think Aloud For APIs* [Cla04] para coleta de dados. Esse protocolo consiste em orientar os desenvolvedores para que verbalizem suas expectativas, estratégias, dificuldades e objetivos ao utilizarem uma determinada API.

Os desenvolvedores foram monitorados por uma câmera de vídeo e tiveram seus discursos gravados através do uso de um microfone. Ainda, foi utilizado um software para captura de tela do computador. Por fim, anotações a respeito do comportamento dos desenvolvedores foram feitas pelo condutor do experimento. Todas essas ferramentas foram utilizadas com o objetivo de coletar uma quantidade extensa de dados relevantes à produção de resultados coerentes.

Ao final do experimento, os participantes foram orientados a responder um questionário[1] que também foi utilizado como fonte de dados para a análise.

Fases

Detalhadamente, a metodologia foi dividida em seis fases:

1. Definição do Projeto a Ser Utilizado no Experimento;

[1] Esse questionário foi aplicado via *web* e está disponível no endereço eletrônico `http://spreadsheets.google.com/viewform?hl=pt_BR&formkey=dDEyQ0p4Y1J3X2hsZXVhdk1tMGN4U1E6MA..` . Além disso, é apresentado no Apêndice B dessa dissertação.

2. Escolha dos Participantes;

3. Definição dos Testes de Design a Serem Escritos;

4. Palestra Sobre Testes de Design;

5. Execução do Experimento;

6. Análise dos Dados;

Fase 1: Definição do Projeto a ser Utilizado no Experimento. O software escolhido para ser utilizado no experimento foi o OurGrid [CBA+06]. OurGrid é um *middleware* para um ambiente de grade computacional aberta, escrito em sua totalidade na linguagem Java. A escolha deste software é decorrente de vários fatores, tais como, o fato de sua equipe de desenvolvimento estar localizada no mesmo laboratório em que este trabalho foi desenvolvido, facilitando o contato com os projetistas e desenvolvedores do projeto. Além disso, trata-se de um sistema em produção desde 2007 que, em sua versão atual, possui 111,790 linhas de código. Um último fator, porém não menos importante, é o fato do OurGrid possuir regras de design bem definidas e que não são checadas de forma automática por falta de mecanismos viáveis que o façam.

Fase 2: Escolha dos Participantes. Foram selecionados onze desenvolvedores para participarem do experimento. A experiência mínima de um ano com programação Java foi utilizada como ponto de corte deste conjunto. Essa estratégia visou impedir a participação de desenvolvedores inexperientes. Neste caso, seria complicado determinar se o participante estava com dificuldade em utilizar a API do DesignWizard ou estava com dificuldade na própria linguagem Java. Esta pré-seleção resultou em um conjunto de desenvolvedores que possuem experiência com Java variando entre 2 e 7 anos, com média igual a 4 anos.

Dentre os desenvolvedores selecionados, cinco estão cursando a graduação em Ciência da Computação pela Universidade Federal de Campina Grande. O restante (seis alunos) estão cursando a Pós-graduação em Ciência da Computação do Departamento de Sistemas e Computação (DSC), lotado na mesma universidade. A idade dos participantes varia entre 20 e 27 anos, com média de 22 anos.

Por fim, é importante destacar que nenhum dos usuários havia escrito testes de design. Portanto, não conheciam a API do DesignWizard.

Fase 3: Definição dos Testes de Design a Serem Escritos. Ao todo, foram selecionados cinco testes de design para serem implementados pelos participantes. Todos os testes de design propostos são relacionados ao projeto OurGrid. Para essa escolha, o líder do projeto foi consultado para que sugerisse regras de design que fossem relevantes no contexto do projeto. Portanto, é importante destacar que os testes de design surgiram de necessidades reais do projeto OurGrid e foram sugestionadas pelo seu próprio líder. Dentre as várias regras de design a serem checadas, cinco foram selecionadas para que os seguintes critérios fossem atendidos:

- Explorar diferentes tipos de regras de design de baixo nível;
- Explorar diferentes abstrações da API do DesignWizard.

As regras selecionadas estão detalhadas abaixo:

Regra 1: Dao-Controller

Somente o pacote `org.ourgrid.peer.controller` pode acessar o pacote `org.ourgrid.peer.dao`.

Essa regra é considerada a mais simples e diz respeito à comunicação entre dois pacotes existentes no projeto OurGrid. Apesar de sua simplicidade, tem um papel fundamental para o projeto, uma vez que restringe o acesso aos objetos de dados.

Regra 2: Dependência Cíclica

Não deve haver dependência cíclica entre pacotes.

Assim como a regra anterior, essa regra está relacionada a comunicação de pacotes. No entanto, ao contrário da Regra 1, que referenciava dois pacotes específicos do projeto, é preciso que o desenvolvedor seja capaz de referenciar todos os pacotes existentes. Evitar que haja dependência cíclica entre projetos implica em uma melhor modularização de seus componentes, o que facilita o entendimento e manutenção dos mesmos.

Regra 3: Pacote Common

Toda classe do pacote `org.ourgrid.common` deve ser utilizada por pelo menos 2 pacotes diferentes.

Essa regra foi adicionada por explorar a abstração *Classe*, além de explorar a relação de continência entre um pacote e várias classes. Segundo o líder do projeto OurGrid, essa regra se faz necessária para que todas as classes presentes no pacote `org.ourgrid.common` sejam realmente comuns ao projeto.

Regra 4: Classe SchedulerCommandExecutor

Toda classe do pacote `org.ourgrid.broker.commands.executors` deve estender a classe `org.ourgrid.broker.commands.executors.SchedulerCommandExecutor`

Além de explorar os aspectos da regra anterior (regra 3), essa regra faz menção a relação de herança entre duas classes, visando assegurar que dentro do pacote `org.ourgrid.broker.commands.executors` só existam classes que sejam executoras de comandos, ou seja, herdem da classe `org.ourgrid.broker.commands.executors.SchedulerCommandExecutor`.

Regra 5: HashCode e Equals

Toda classe que explicitamente declarar o método `hashCode(int)` deve também declarar explicitamente o método `equals(java.lang.Object)`

Além de explorar a relação de continência entre uma classe e um método, essa regra faz menção a um aspecto particular da API do DesignWizard, que é distinguir métodos herdados dos métodos declarados.

Fase 4: Palestra Sobre Testes de Design. Antes da execução do experimento, ministramos para cada desenvolvedor uma palestra com duração máxima de 30 minutos sobre o conceito de teste de design. É importante ressaltar que, ao longo da palestra, não foram

mencionados ou apresentados métodos presentes na API do DesignWizard. Tal ação poderia compremeter a validade do experimento, uma vez que os desenvolvedores poderiam ter conhecimento prévio sobre a API.

Fase 5: Execução do Experimento. O experimento foi executado no Laboratório de Sistemas Distribuídos (LSD) da Universidade Federal de Campina Grande (UFCG). Cada participante, individualmente, escreveu os testes de design estabelecidos. Em um primeiro momento, essa escrita foi realizada através de pseudocódigo. Após essa etapa, os desenvolvedores tiveram que escrever os mesmos testes utilizando a API do DesignWizard e o Junit.

Foi utilizado o protocolo *Think Aloud* para coletar dados a respeito das expectativas dos participantes em relação à API do DesignWizard. Além disso, os partipantes foram submetidos a um questionário que abordava questões relacionadas à usabilidade da API.

Fase 6: Análise dos Resultados. Após o fim da execução dos experimentos e da coleta de dados, quantificamos os resultados para então iniciar a fase de análise dos mesmos. Essa análise foi dividida em três fases:

1. Comparação entre os pseudocódigos e a API do DesignWizard;
2. Análise dos dados provenientes do protocolo *Think Aloud*;
3. Análise das respostas do questionário.

Comparação entre os pseudocódigos e a API do DesignWizard.

Essa comparação foi realizada com intuito de verificar se a API atendia às expectativas dos desenvolvedores. Isso foi feito através da análise da similaridade entre os métodos presentes no pseudocódigo e a API do DesignWizard.

Análise dos dados provenientes das observações.

Nessa fase, a avaliação do áudio e das anotaçoes feitas durante o protocolo *Think Aloud* se mostraram de suma importância para uma análise embasada dos resultados. Em relação aos vídeos de captura de tela e do comportamento dos programadores, poucos dados foram

avaliados. Na análise de uma interface gráfica é extremamente importante essa fonte de dados. No entanto, quando se trata de APIs, as outras fontes se mostram mais precisas.

Análise das respostas do questionário.

Para analisar os dados coletados através do questionário, foi utilizado como guia o *Framework* de Dimensões Cognitivas, proposto por Green e Petre [GP96] e adaptado por Clarke e Becker [Cla05].

O *Framework* de Dimensões Cognitivas apresenta 12 dimensões que individualmente e coletivamente causam impacto no aprendizado de uma API e na maneira como os desenvolvedores a manipulam. As questões incluídas no questionário o foram por explorar algumas dimensões deste *framework*. Em particular, foram escolhidas três dimensões para serem exploradas a partir das respostas dos participantes: *API Viscosity*, *Role Expressiveness* e *Work-Step Unit*. Essas dimensões foram escolhidas por levarem em consideração aspectos mais relacionados ao cliente da API, ou seja, os desenvolvedores.

API Viscosity é a dimensão que está relacionada à dificuldade encontrada por um desenvolvedor ao modificar um código escrito em uma determinada API. Por outro lado, a dimensão *Role Expressiveness* descreve quão fácil é entender um código escrito na API. Ou seja, se a API é expressiva ou não. A última dimensão selecionada, *Work-Step Unit*, versa sobre o esforço que um desenvolvedor precisa empregar para cumprir determinada tarefa.

É importante destacar que as perguntas utilizadas no questionário seguiram sugestões do trabalho de Clarke [Cla04], onde o pesquisador descreve questões que devem ser utilizadas para explorar as dimensões do *framework* de dimensões cognitivas citadas anteriormente.

4.1.2 Resultados

Os resultados do experimento estão descritos em duas partes. Inicialmente, serão apresentados os números e observações referentes a cada teste de design proposto para os participantes. A outra parte consiste em reportar os resultados tendo como viés o *framework* de dimensões cognitivas.

Resultados da regra Dao-Controller

Para a implementação dessa regra, era esperado que os desenvolvedores seguissem a estratégia de adquirir uma referência ao objeto que provê informações do pacote `org.ourgrid.peer.dao` para que pudessem obter todos os chamadores deste pacote através de uma método `getCallersPackages()`. O último passo seria assegurar que esse conjunto de chamadores possui um único elemento, o pacote `org.ourgrid.peer.controller`.

A Tabela 4.1 detalha os resultados da comparação entre os pseudocódigos e a API do DesignWizard. Essa comparação quantifica as expectativas dos desenvolvedores e descreve se a API do DesignWizard cumpre com essas expectativas ou não.

Tabela 4.1: Resultados da regra Dao-Controller

Método Esperado	**Ocorrências no Pseudocódigo**	**Presente na API**
getPackage(String nomeDoPacote)	9	Sim
getCallers()	8	Sim
getAllClasses()	2	Sim
getAllPackages()	1	Sim
getName()	1	Sim
new PackageNode(String nomeDoPacote)	1	Não
getAccessingPackages(pacote)	1	Não

Análise. Como pode ser visto na Tabela 4.1, a análise do pseudocódigo revelou que nove entre os onze desenvolvedores esperavam que houvesse um método `getPackage(String nomeDoPacote)` (ou algo muito semelhante) na API do DesignWizard que retornasse uma abstração que provê informações do pacote com o nome especificado. Um dos desenvolvedores, esperava que houvesse um método `getAllPackages()` que retornasse o conjunto de todos os pacotes presentes no código para que ele, posteriormente, pudesse iterar sobre esse conjunto e comparar através de um método `getName()`. Por fim, um único desenvolvedor esperava criar o objeto do tipo Pacote através de um construtor passando como parâmetro o nome do pacote.

De todas as alternativas adotadas pelos desenvolvedores, apenas a última não está pre-

sente na API do DesignWizard. A API não fornece construtores para que sejam criadas abstrações do código extraído. Ao invés disso, a API fornece métodos que já retornam essas abstrações prontas e com as informações a respeito das entidades, por exemplo, `getPackage(String name)` e `getClass(String name)` etc.

Para ter referência aos pacotes chamadores, oito entre os onze desenvolvedores esperavam que houvesse um método `getCallers()` na abstração retornada pelo método `getPackage(String nomeDoPacote)`. Esse método deveria retornar uma coleção contendo os pacotes chamadores. Outros dois desenvolvedores utilizaram a estratégia de obter todas as classes de um pacote para então obter seus chamadores. Apenas um desenvolvedor esperava que uma entidade central tivesse um método para consultar a respeito dos chamadores de um determinado pacote (e g. `DesignWizard.getAccessingPackages(pacote)`).

O Código 3 é um exemplo de pseudocódigo produzido por um dos desenvolvedores. É possível notar que, comparando com o Código 4, escrito pelo mesmo desenvolvedor, o pseudocódigo assemelha-se muito à implementação real. De fato, as mudanças estão na nomenclatura utilizada para as abstrações e na coleção retornada pelo método `getCallersPackages()`, diferenças essas que não se mostram obstáculo no uso da API.

Código 3 Teste de design para regra Dao-Controller (Pseudocódigo).

```
1 Package daoPack = dw.getpackage(org.ourgrid.peer.dao);
2 Package controllerPack = dw.getpackage(org.ourgrid.peer.controller);

3 List daoCallers = daoPack.getPackCallers();

4 assertEquals(1, daoCallers.size());
5 assertEquals(controllerPack, daoCallers.get(0));
```

Como era de se esperar, ao transcrever o pseudocódigo para implementação utilizando DesignWizard, os desenvolvedores não tiveram grandes dificuldades. Sobretudo porque, na maioria dos casos, até a nomenclatura utilizada no pseudocódigo foi a mesma utilizada para a API. Até mesmo o desenvolvedor que adotou a estratégia de criar uma abstração para Pacote, encontrou a alternativa existente facilmente.

Código 4 Teste de design para regra Dao-Controller.

```
1 PackageNode daoPack = dw.getPackage("org.ourgrid.peer.dao");
2 PackageNode controllerPack = dw.getPackage("org.ourgrid.peer.controller");

3 Set<PackageNode> daoCallers = daoPack.getCallerPackages();

4 assertEquals(1, daoCallers.size());
5 assertEquals(controllerPack, daoCallers.iterator().next() );
```

Para esta regra, todos os desenvolvedores concluíram com sucesso a escrita do teste de design utilizando o DesignWizard e o JUnit.

Por fim, durante o experimento foi possível notar que os desenvolvedores utilizaram os métodos `equals(Object)` e `contains(Object)` com naturalidade para comparar objetos da API e verificar a existência de objetos em conjuntos. Essa observação suporta a ideia de que utilizar a mesma linguagem para descrever as regras de design é um fator que implica na usabilidade da técnica.

Resultados da regra de Dependência Cíclica

A estratégia esperada para a resolução desse problema é simples, basta obter a referência para todos os pacotes do código e assegurar que não há, entre quaisquer dois pares de pacotes, dependência cíclica. Isto é, os dois dependem mutuamente um do outro. Os resultados para essa regra estão na Tabela 4.2.

Tabela 4.2: Resultados da regra de Dependência Cíclica

Método Esperado	**Ocorrências no Pseudocódigo**	**Presente na API**
getAllPackages()	10	Sim
getCallers()	6	Sim
calls(package)	4	Não
getPacks(String regex)	1	Não
depends(package)	1	Não

Análise. A partir da análise do pseudocódigo dos desenvolvedores foi possível notar que dez entre os onze desenvolvedores utilizaram no pseudocódigo o método

`getAllPackages()`, ou nomenclatura muito semelhante, por exemplo, `getPacks()` e `getPackages()`. Para esse cenário, as expectativas dos desenvolvedores foram atendidas pelo método `getAllPackages()` presente na API do DesignWizard. Apenas um dos desenvolvedores esperava que houvesse na API um método `getPkgs(String regex)` que recebesse como parâmetro uma expressão regular definindo o conjunto de pacote que deveria ser retornado, assim como ilustra o Código 5.

Código 5 Sugestão de método para a API do DesignWizard.

```
1 getPkgs("org.myproject.*")
```

Neste caso, embora o método não esteja presente na API, o desenvolvedor encontrou facilmente a alternativa, mas ressaltou que preferiria o método indicado no pseudocódigo.

Outro ponto a ser destacado é o fato de que quatro desenvolvedores esperavam que houvesse o método `calls(package)` na API. Além disso, um desenvolvedor utilizou em seu pseudocodigo a nomenclatura `depends(package)` para verificar se um pacote usa outro pacote. Nesse caso, embora esses métodos não estejam presentes na API do DesignWizard, a alternativa `getCallers()` foi adotada pelos desenvolvedores na implementação real. Através da protocolo *Think Aloud* foi possível identificar que os desenvolvedores não tiveram problemas em encontrar a alternativa para a ausência dos métodos esperados.

Todos os participantes concluíram a especificação desse teste de design com sucesso.

Resultados da regra do pacote Common

Esta regra difere das anteriores por requisitar o uso de uma abstração para as classes presentes no código, e não somente pacotes. Neste caso, é necessário obter um conjunto de classes pertencentes ao pacote `org.ourgrid.common`. Portanto, era de se esperar que os desenvolvedores fizessem menção ao método `getAllClasses()` presente na API do DesignWizard, que retornaria uma coleção contendo as classes pertencentes a um determinado pacote. O resultado da comparação entre os pseudocódigos e a API estão inseridos na Tabela 4.3.

Tabela 4.3: Resultados da regra do pacote Common

Método Esperado	**Ocorrências no Pseudocódigo**	**Presente na API**
getCallers()	11	Sim
getAllClasses()	9	Sim
getPackage()	8	Sim
getClasses(String pacote)	2	Não
new PackageNode(String nomeDoPacote)	1	Não

Análise. De fato, nove entre os onze desenvolvedores fizeram menção ao método `getAllClasses()` em seus pseudocódigos e não tiveram dificuldades ao implementar o teste de design utilizando o DesignWizard, visto que existe esse método na API. Novamente, através da comparação entre o pseudocódigo e o código dos desenvolvedores, ficou evidenciado o fato de que a nomenclatura utilizada na API do DesignWizard aproxima-se daquele adotada na resolução dos problemas.

O restante dos desenvolvedores esperavam que uma entidade central fosse responsável por obter essa informação, tal como é ilustrado no pseudocódigo (Pseudocódigo 7).

PseudoCódigo 7 Exemplo de método para obter classes de um pacote.

```
1 DesignWizard.getClasses(pacote)
```

Apesar do fato de não existir tal método na API do DesignWizard, foi possível notar que esses dois desenvolvedores conseguiram encontrar a alternativa para a resolução desse problema facilmente e, portanto, a utilizaram na implementação da regra com o DesignWizard.

Novamente, todas as implementações dos participantes do experimento foram realizadas corretamente.

Resultados da regra da classe SchedulerCommandExecutor

A estratégia para o desenvolvimento desse teste de design consiste em iterar sobre o conjunto de entidades do pacote `org.ourgrid.broker.commands.executors` e verificar se as classes existentes herdam da classe `SchedulerCommandExecutor`. Portanto, é preciso antes dessa verificação, assegurar dois aspectos: i) que a entidade é realmente uma

classe e ii) que a classe não é a própria `SchedulerCommandExecutor`.

A primeira condição deve ser verificada pois o método `getAllClasses()` da classe `PackageNode` retorna todas as classes e interfaces de um determinado pacote, pois uma entidade `ClassNode` é a representação de uma classe ou uma interface do sistema. A API foi desenvolvida dessa maneira para seguir o modelo da biblioteca padrão de Java que provê informações a respeito do código.

Os resultados provenientes da comparação entre os pseudocódigos e a API estão presentes na Tabela 4.4.

Tabela 4.4: Resultados da regra da classe SchedulerCommandExecutor

Método Esperado	**Ocorrências no Pseudocódigo**	**Presente na API**
getClasses()	10	Sim
getPackage(String package)	9	Sim
extends(Class class)	8	Não
getClass(String className)	7	Sim
isInterface()	3	Sim
getSuperClass(Class class)	3	Sim
isClass()	3	Sim
getName()	2	Sim
getSuperClass()	2	Não
isSubClass()	1	Sim
getClasses(PackageNode pack)	1	Não
new PackageNode(String nome)	1	Não

Análise. Foi possível notar através da análise dos códigos que seis entre onze desenvolvedores esperavam que o método `getAllClasses()` retornasse apenas objetos que representassem classes de um determinado pacote, ao invés de interfaces também. Por este motivo, esses seis desenvolvedores cometeram erro ao utilizar a API. Embora uma simples consulta a API esclarecesse a dúvida, ela não foi consultada por nenhum desses desenvolvedores.

Outra observação importante constada diz respeito a verificação de herança entre classes. Duas alternativas diferentes foram adotas pelos desenvolvedores: Em uma al-

ternativa, três dentre os onze desenvolvedores esperavam a existência de um método `getSuperClass()`. O restante, oito desenvolvedores, utilizaram a alternativa de um método *booleano* `extends(Class class)`. Dessas duas alternativas, a API do DesignWizard contempla somente a primeira. No entanto, os desenvolvedores que adotaram a segunda alternativa no pseudocódigo não tiveram dificuldades para utilizar a primeira alternativa.

Os resultados dessa regra sugestionam que a implementação do método `getAllClasses()` deve ser modificada para retornar apenas as classes existentes em um determinado pacote. Ainda, é preciso adicionar um método `getAllInterfaces()` para que as interfaces sejam retornadas separadamente das classes. Por fim, foi sugestionado por dois dos desenvolvedores a adição do método booleano `extends(Class c)` para verificar a relação de herança entre duas classes.

Resultados da regra do HashCode e Equals

Este teste de design requer que os desenvolvedores utilizem um método específico da API do DesignWizard que faz menção a uma classe declarar explicitamente, e não somente herdar, um determinado método. O método a ser utilizado pelos desenvolvedores seria o `getDeclaredMethod(String signature)`. A Tabela 4.5 apresenta os resultados para essa regra.

Análise. A análise do pseudocódigo revelou a ausência de um método importante na API. Quatro entre os onze desenvolvedores esperavam que houvesse um método `containsMethod(String name, List params)` na abstração de uma classe. No entanto, a API do DesignWizard provê os métodos `getAllMethods()` e `getDeclaredMethod(String signature)` que podem ser utilizados para verificar a existência de um determinado método em uma classe. Pela análise do código e através do protocolo *Think Aloud*, foi possível detectar que dentre esses quatro desenvolvedores apenas um não conseguiu encontrar a alternativa oferecida pela API.

Dentre os onze desenvolvedores, cinco não especificaram o teste de design corretamente. Dentre esses cinco que erraram o teste, quatro o fizeram por não ler a documentação (Javadoc) da API, pois utilizaram o método correto para a ação

Tabela 4.5: Resultados da regra da do HashCode e Equals

Método Esperado	**Ocorrências no Pseudocódigo**	**Presente na API**
getAllClasses()	11	Sim
declares(String signature)	4	Sim
containsMethod(String name, List params)	4	Sim
getMethod(String signature)	2	Sim
getAllMethods()	2	Sim
new Method(String signature)	1	Não
getDeclareMethod(String signature)	1	Sim
getName()	1	Sim
getReturnType()	1	Sim
getParameters()	1	Sim

(`getDeclaredMethod(String signature)`), mas a passagem de parâmetro foi equivocada. Por exemplo, um trecho de código (Código 6) retirado da implementação de um dos desenvolvedores utiliza o método `getDeclaredMethod(String signature)` de maneira equivocada. Ao invés de passar como parâmetro toda a assinatura do método, como orienta a documentação, o desenvolvedor passou como parâmetro apenas o nome do método.

Código 6 Exemplo de código de um dos desenvolvedores.

```
1 MethodNode declaredMethod = aClass.getDeclaredMethod("hashCode");
```

Neste caso, os resultados sugeriram a criação do método `containsMethod(String name, List params)` na API do DesignWizard. Este método permite verificar a existência de um método com nome e parâmetros especificados.

Resultados do Questionário

Os dados coletados através do questionário submetido aos desenvolvedores serão analisados nessa seção. Essa análise levará em consideração o framework de dimensões cognitivas,

através do qual aspectos da usabilidade da API serão abordados.

Em relação a dimensão *API Viscosity*, o questionário continha a seguinte questão:

- Se você precisasse mudar o código você teria: Muito esforço, esforço moderado ou pouco esforço?

Neste caso **100% dos participantes** responderam que teriam pouco esforço caso tivessem que modificar o código implementado. Embora o esforço de uma mudança dependa da própria mudança a ser efetuada, foi observado durante o experimento que os desenvolvedores, por muitas vezes, refatoravam o código para uma solução mais simples do que a implementada anteriormente. Em alguns casos, a mudança de estratégia na resolução do problema evidenciou que a API não se fazia um obstáculo para essa mudança.

Com relação a dimensão *Role Expressiveness*, foram efetuadas duas perguntas:

- Quando você lê o código feito, quão fácil é saber o que cada parte faz? (muito fácil, fácil, razoável, difícil ou muito difícil)
- Quão fácil é saber qual classe e método usar para determinada subtarefa? (muito fácil, fácil, razoável, difícil ou muito difícil)

Os histogramas 4.1(a) e 4.1(b) ilustram as respostas dos desenvolvedores para as questões supracitadas. Em ambos os casos, os resultados indicam que a API é intuitiva e explicativa, haja vista que, em sua grande maioria, todas as respostas variam entre "muito fácil" e "fácil", obtendo apenas um resposta "razoável". A partir do questionário e dos relatos do protocolo *Think Aloud*, identificou-se que sete entre os onze desenvolvedores indicaram que a nomenclatura dos métodos e abstrações é intuitiva, reforçando ainda mais os indícios de uma boa usabilidade da API. Um dos desenvolvedores elogiou o fato da API do DesignWizard ser bem semelhante à API de Java para extração de informações do código-fonte utilizando técnicas de reflexão.

Levando em consideração a dimensão *Work-Step Unit* do *framework* de dimensões cognitivas, a seguinte questão foi levantada:

- Em sua opinião, a quantidade de código necessária para compor os testes é grande, justa ou pequena?

O histograma 4.1(c) quantifica as respostas dos participantes. A grande maioria das respostas (dez entre onze) variam entre "Justa" e "Pequena", indicando que os desenvolvedores acreditam que compor testes de design não é uma tarefa que requer demasiado esforço.

Adicionalmente a resposta dessa pergunta, para explorar melhor a dimensão *Work-Step Unit*, foi requisitado aos participantes que justificassem a resposta à pergunta anterior. Em relação ao único desenvolvedor que acreditou ser grande a quantidade de código necessária para a escrita dos testes, foi relatado que ele gostaria de ter escrito menos código Java para que fosse possível checar as regras de design. Dentre os outros depoimentos, um em particular chama a atenção. O desenvolvedor ressaltou que acha a quantidade de código escrita semelhante aos testes funcionais que ele costuma implementar e que por isso, considera justa a quantidade de código necessária para compor os testes de design. Ainda, sete dentre os onze desenvolvedores, ressaltaram que as abstrações presentes na API do DesignWizard facilitam em muito a escrita dos testes.

Por fim, foi adicionada uma questão que diz respeito à facilidade de transcrever o pseudocódigo escrito para uma implementação concreta utilizando o DesignWizard:

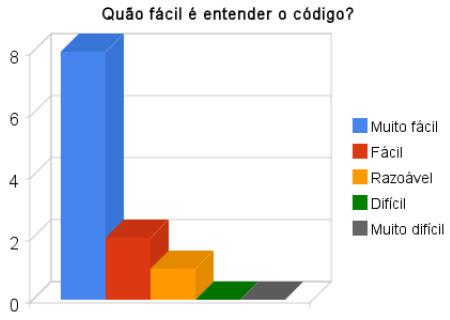
- Quão fácil foi traduzir o pseudocódigo para código utilizando o DesignWizard?

Embora não esteja relacionada a nenhuma dimensão do *framework* de dimensões cognitivas, esta pergunta foi adicionada porque baseia-se na ideia de que, se os desenvolvedores considerarem fácil a tradução do pseudocódigo para a implementação de fato, a API não é um obstáculo na adoção da abordagem, uma vez que a escrita de testes de design é considerada uma tarefa fácil de ser efetuada. O histograma 4.1(d) mostra as respostas dos desenvolvedores em relação a essa pergunta.

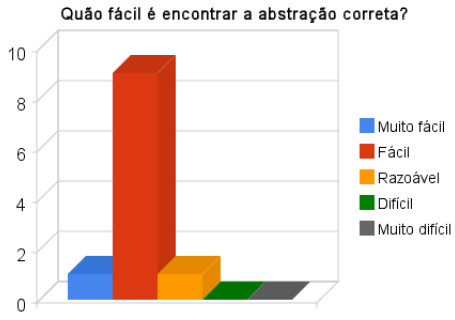
Todas as respostas variam de "Muito Fácil" a "Fácil", evidenciando que, para esse conjunto de desenvolvedores, a API do DesignWizard aproxima-se do pseudocódigo escrito por eles. Outro dado importante extraído do protocolo *Think Aloud* que reforça essa proximidade é que todos os desenvolvedores, sem exceção, comentaram a respeito da semelhança entre o pseudocódigo escrito e a implementação concreta.

Um último, porém não menos importante, quesito que se mostra relevante para a avaliação de uma API, é a sua documentação. Em relação a este aspecto, foi requerido aos participantes do experimento que avaliassem a documentação API do DesignWizard. A

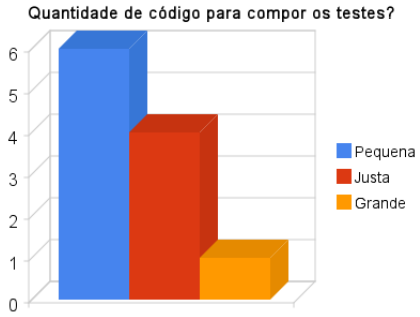
Figura 4.1(e) sumariza os dados extraídos do questionário. Podemos inferir através desses resultados e dos dados extraídos do protocolo *Think Aloud* que a documentação da API foi bem avaliada pelos desenvolvedores.



(a) Facilidade de mapear código para a sua função.



(b) Facilidade para mapear entidades do código para determinadas tarefas.

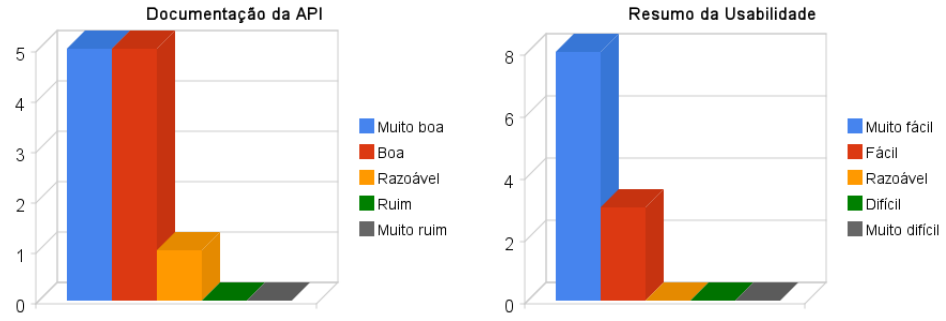


(c) Quantidade de código necessária para compor os testes.

(d) Facilidade de traduzir pseudocódigo em implementação real.

4.1.3 Limitações do Estudo

Do ponto de vista de pesquisa, é preciso evidenciar que os resultados obtidos através do experimento apenas dão indícios de que a API é fácil de ser manipulada. Por se tratar de um experimento com pouca validade estatística, não é possível garantir que na maioria dos casos os desenvolvedores terão facilidade ao utilizar a API do DesignWizard para construir testes de design, haja vista a existência de vários fatores que podem comprometer as observações feitas.



(e) Documentação da API do DesignWizard. (f) Usabilidade da API do DesignWizard.

Portanto, não é possível generalizar os resultados aqui obtidos. De fato, o objetivo deste experimento é relatar a experiência de alguns desenvolvedores ao utilizar a API do DesignWizard e evidenciar que, de um modo geral, a API desempenhou um bom papel no que diz respeito a sua usabilidade, ou seja, no atendimento às expectativas dos desenvolvedores.

4.1.4 Conclusões

Nesse capítulo foi apresentada a avaliação da usabilidade da API do DesignWizard. Essa avaliação seguiu uma metodologia adotada por trabalhos anteriores na área. De um modo geral, como pode ser visto na Figura 4.1(f), os desenvolvedores não tiveram dificuldades em utilizar a API e a consideram, na maioria dos casos, muito fácil de ser manipulada.

A avaliação conduzida foi de fato extremamente importante para que fosse possível identificar em casos reais a facilidade que os desenvolvedores possuem ao implementar testes de design. Ainda, a avaliação teve uma contribuição muito relevante no que diz respeito ao amadurecimento da API do DesignWizard. Os defeitos e sugestões que se evidenciaram durante a avaliação da API contribuíram para a sua evolução. É importante destacar que essa evolução é fruto da experiência de diferentes programadores utilizando a API do DesignWizard.

4.2 Escalabilidade

Para avaliar a escalabilidade do DesignWizard, conduzimos um experimento para medir o tempo que a ferramenta leva para, através de análise estática, extrair fatos de aplicações Java.

Como objeto de estudo do experimento, utilizamos oito arquivos Jar [2] de aplicações Java com tamanho variando entre 0.1MB (*External Tools Plugin*) e 46MB (*Biblioteca Padrão de Java*), que foram escolhidos para representar pequenos e grandes projetos, respectivamente. A Tabela 4.6 apresenta os dados das aplicações escolhidas.

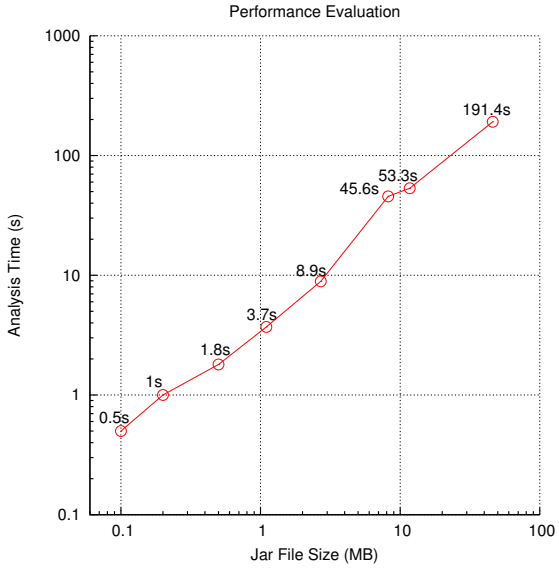
Tabela 4.6: Aplicações utilizadas para a medição do tempo de extração.

Aplicação	**Tamanho (MB)**	**Número de Relações Extraídas**
External Tools Plugin	0.1	4458
Subversion Plugin	0.2	9115
Text Editor Plugin	0.5	21702
Team UI Plugin	1.1	37070
FindBugs	2.7	107261
JDT Plugin	8.2	324750
Azureus	11.7	414320
Biblioteca Padrão de Java	46	1278409

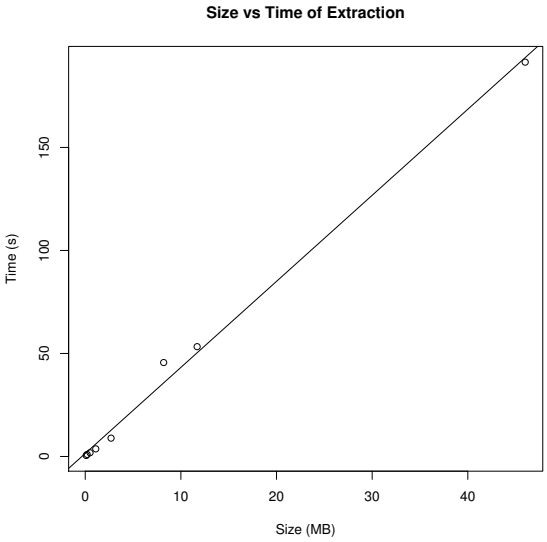
A coluna *Número de Relações Extraídas* é referente à quantidade de relações que o DesignWizard extrai de cada Jar. Inicialmente, no planejamento do experimento, esse era o critério para a escolha dos projetos a serem extraídos. No entanto, isso dificultaria a procura desses projetos, pois para saber o número de relações é preciso executar o DesignWizard. Calculamos o coeficiente de correlação (R) entre o tamanho de um arquivo Jar e a quantidade de relações. Para o conjunto de dados presentes na Tabela 4.6, o coeficiente de correlação apontou para uma relação forte entre o tamanho de um arquivo Jar e o número de relações (*R=0.9914*). Este cálculo nos permitiu usar o tamanho dos arquivos Jar como parâmetro de entrada no experimento.

A Tabela 4.7 ilustra os dados extraídos do experimento com intervalos de confiança de 95% (N=30). Para fins de melhor visualização, a Figura 4.1(g) resume esses dados em escala logarítmica, mostrando o tempo (em segundos) que o DesignWizard leva para concluir a análise estática à medida que o tamanho do software cresce. Através da análise do gráfico, podemos observar o crescimento linear do tempo de extração em relação ao tamanho da aplicação extraída. Para ratificar essa observação, calculamos a regressão linear dos dados

[2] Foram escolhidos arquivos Jar que continham apenas arquivos .class em seu conteúdo.



(g) Desempenho do DesignWizard em relação ao tamanho do projeto.



(h) Regressão linear

e o valor do coeficiente de determinação $R^2 = 0.9946$. A Figura 4.1(h) ilustra o gráfico da regressão linear em conjunto com os resultados do experimento.

Ao analisar a Figura 4.1(g) é possível perceber que o DesignWizard escala e que eficiência no tempo de extração não é um problema para a ferramenta. Um ponto importante a ser destacado no gráfico é que para a extração de fatos do *software* FindBugs, o DesignWizard levou 8.9 segundos. Este ponto do gráfico é importante, pois o tamanho do FindBugs é representativo, uma vez que é igual a média dos dez software mais adquiridos através do SourceForge. É importante ressaltar que esse número é aceitável, visto que é preciso aplicar a análise estática uma só vez para uma bateria de testes de design. Mesmo o tempo de 191.4 segundos para a extração da biblioteca padrão de Java é aceitável, haja vista o tamanho do projeto.

O experimento foi executado em uma máquina com processador Intel(R) Core(TM)2 Duo 2.33GHz, 1 GB de memória RAM disponível e operando sobre a plataforma linux com distribuição Debian e kernel 2.6.28-11-vserver.

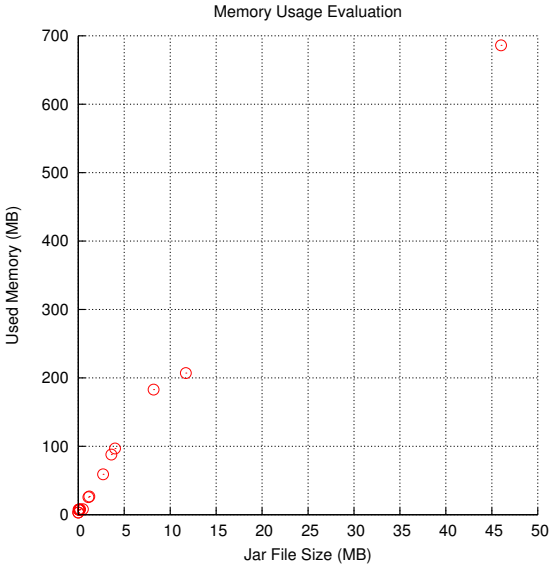
Tabela 4.7: Dados sobre o desempenho do DesignWizard em relação ao tamanho do projeto a ser extraído.

Aplicação	**Tempo (s)**
External Tools Plugin	0.5 ± 0.025
Subversion Plugin	1 ± 0.072
Text Editor Plugin	1.8 ± 0.055
Team UI Plugin	3.7 ± 0.066
FindBugs	8.9 ± 0.192
JDT Plugin	45.6 ± 0.536
Azureus	53.3 ± 0.343
Biblioteca Padrão de Java	191.4 ± 0.986

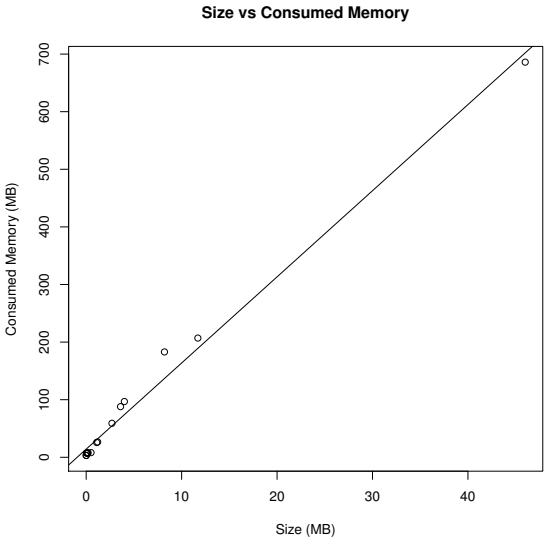
4.3 Avaliação do Consumo de Memória

Após a extração dos fatos, a estratégia adotada no desenvolvimento do DesignWizard é de manter todo o grafo extraído na memória principal para que o acesso às informações através da API seja otimizado. No entanto, essa estratégia pode se tornar um gargalo na utilização de memória. Para avaliar tal aspecto, conduzimos um experimento com os mesmos arquivos Jar utilizados no experimento de escalabilidade, detalhados na Tabela 4.6. A Figura 4.1(i) ilustra o consumo de memória da ferramenta em relação ao tamanho da aplicação sob verificação, enquanto que a Figura 4.1(j) ilustra o gráfico da regressão linear (coeficiente de determinação $R^2 = 0.9892$) em conjunto com os resultados do experimento.

Como é possível perceber, o crescimento do consumo de memória é linear. Para o JAR correspondente a toda a biblioteca Java, o DesignWizard consumiu um quantidade razoável de memória. Foi preciso alterar os parâmetros de inicialização da JVM para que a extração fosse concluída com sucesso. Esse é um artifício comumente utilizado para a execução de aplicações Java de grande porte. Todavia, se esse aspecto for visto como um gargalo, esse problema é facilmente endereçado em futuras implementações do DesignWizard. Várias alternativas podem ser utilizadas para tal objetivo. Uma delas é a utilização de um banco de dados para o armazenamento dos fatos extraídos. Dessa maneira, a memória principal não seria um gargalo para a extração de grandes projetos.



(i) Consumo de memória do DesignWizard em relação ao tamanho do projeto.



(j) Regressão linear

Nome	**Descrição**	**LOCs**	**# Classes**
OurGrid	Middleware para Grid P2P	111,790	683
OurBackup	Ferramenta para backup P2P	133,974	642

Tabela 4.8: Projetos selecionados para o estudo de caso.

4.4 Estudos de Caso

Esta seção relata os dois estudos de caso conduzidos com o intuito de avaliar a utilização de testes de design na detecção de violações de regras em projetos reais. Como objetos de estudo, foram utilizados o Ourgrid e o OurBackup [Oli07], cujos detalhes estão dispostos na Tabela 4.4. OurGrid é um *middleware* para um ambiente de grade computacional aberta, escrito em sua totalidade na linguagem Java. Ourbackup é um sistema entre-pares que tem por objetivo promover a cooperação entre peers para backup de informações. A escolha desses software é decorrente de vários fatores, tais como, o fato de suas equipes de desenvolvimento estarem localizadas no mesmo laboratório em que este trabalho foi desenvolvido, facilitando o contato com os projetistas e desenvolvedores dos projetos. Um último fator, porém não menos importante, é o fato desses projetos possuirem regras de design bem definidas e que não são checadas de forma automática por falta de mecanismos viáveis que o façam.

Não obstante ao fato desses projetos serem desenvolvidos na mesma universidade em que desenvolvemos nossa abordagem, é importante destacar que nossa equipe não faz parte da equipe de desenvolvimento desses projetos.

O primeiro estudo caso foi realizado tendo como objeto de estudo o OurGrid. Posteriormente, realizamos o estudo de caso com o projeto OurBackup. Em ambos, a metodologia utilizada consistiu em 4 etapas assim definidas:

- Identificação das regras a serem checadas;
- Implementação das regras em forma de testes de design;
- Execução dos testes de design;
- Análise das violações.

A identificação da regra a ser aplicada no OurGrid foi realizada através de uma entrevista com o gerente do projeto. Portanto, a regra escolhida para ser implementada e checada através teste de design foi fruto de uma necessidade real dentro do contexto do projeto.

A regra escolhida diz respeito à restrição de comunicação entre os pacotes `dao` e `controller`. Especificamente, a regra estabelece a seguinte restrição:

> Somente o pacote `org.ourgrid.peer.controller` pode acessar o pacote `org.ourgrid.peer.dao`.

Essa regra, apesar de simples, tem um papel fundamental para o projeto, uma vez que restringe o acesso aos objetos de dados. Na prática, essa regra era checada manualmente pelo gerente do projeto OurGrid. E por esse motivo, segundo ele, não era constantemente checada. Esse cenário suporta a ideia de que há necessidade de um mecanismo automático de verificação de conformidade entre regras de design e implementação.

Uma vez identificada a regra e a necessidade de se checá-la periodicamente, a etapa de escrita dos testes de deisgn foi realizada por nossa equipe. O Código 7 é a implementação da regra como um teste de design. Essa etapa não foi realizada por membros do projeto OurGrid porque o intuito desse estudo não é avaliar a escrita de testes de design e sim a sua eficácia na detecção de violações. Detalhes sobre avaliação da facilidade de escrita de testes de design estão relatados na Seção 4.1 deste capítulo.

Código 7 Teste de design para regra Dao-Controller.

```
1 PackageNode daoPack = dw.getPackage("org.ourgrid.peer.dao");
2 PackageNode controllerPack = dw.getPackage("org.ourgrid.peer.controller");

3 Set<PackageNode> daoCallers = daoPack.getCallerPackages();

4 assertEquals(1, daoCallers.size());
5 assertEquals(controllerPack, daoCallers.iterator().next() );
```

A etapa de execução da regra de design revelou 10 violações. A análise dessas violações foi efetuada em conjunto com o gerente da equipe de desenvolvimento do OurGrid. Segundo o gerente, algumas das violações eram esperadas. Por exemplo, entidades do pacote de testes necessitam ter acesso direto ao pacote `dao` para facilitar a escrita dos testes de unidade. Esse caso revelou que a regra foi concebida de maneira equivocada, uma vez que não se adicionou essa restrição. Esse cenário é um resultado muito importante da execução de testes de design, pois leva a equipe a levar em consideração o porquê das violações. Esse cenário faz com que o design esteja sob constante revisão.

Embora algumas violações tivessem sido consideradas como aceitáveis pelo gerente do projeto Ourgrid, outras violações identificadas pela execução do teste foram consideradas inaceitáveis. Por exemplo, foi identificado pela execução do teste que a classe `WorksTable`, que é parte do código responsável pela interface gráfica da aplicação, acessava o classes do pacote `dao` diretamente. Seguir essa regra, segundo o gerente do projeto, é muito importante para manter a qualidade do código, uma vez que promove a separação entre dados e apresentação.

A aplicação do teste de design no projeto OurGrid revelou aspectos importantes da abordagem. Inicialmente, evidenciou-se o fato de que testes de design são eficazes no que diz respeito a detecção de violações de regras de design na implementação. Um outro aspecto muito importante evidenciado pela execução do estudo de caso foi o fato de que o resultado da execução do teste de design traz importantes discussões a respeito do design da aplicação sendo desenvolvida.

Por último, é importante destacar que, em projetos como o OurGrid, que utilizam testes para garantia de qualidade da aplicação, a abordagem de testes de design se adequa facil-

mente. No OurGrid, testes funcionais são executados automaticamente sempre que um *commit* é realizado. Caso os testes revelem falhas no software, o desenvolvedor não está habilitado a submeter o código desenvolvido. Diante desse cenário, se os testes de design forem adicionados à suite de testes funcionais, ainda que o código do desenvolvedor esteja funcionalmente correto, ele também precisará estar de acordo com as regras de design estabelecidas. Essa situação acarreta em um cenário em que o design está sob constante avaliação, trazendo os benefícios advindos de revisões periódicas de design.

O outro estudo de caso realizado teve como objeto de estudo o projeto OurBackup. Como dito anteriormente, a metodologia empregada foi a mesma definida no início dessa seção.

Assim como o projeto OurGrid, o OurBackup também possui regras de design a serem seguidas pelos desenvolvedores, mas que não estão documentadas. Essas regras são fruto de discussões entre a equipe, embora não estejam documentadas, são de conhecimento comum a toda a equipe.

Para a indentificação das regras a serem aplicadas, o líder do projeto foi consultado. Essa primeira etapa resultou em um conjunto de quatro regras a serem desenvolvidas na forma de testes de design. Dentre elas, destacamos a regra que estabelece que a classe `org.ourbackup.server.Server` deve conter apenas métodos públicos, que retornem `void` e não lancem exceção alguma. O Código 8 é o teste de design que implementa essa regra.

Código 8 Design Test code for methods from Server class.

```
 1 public class OurBackupDesignTest extends TestCase {
 2  public void testServerMethodsSignature() {
 3    DesignWizard dw;
 4    dw = new DesignWizard("ourbackup.jar");
 5    ClassNode server = dw.
 6       getClass("org.ourbackup.server.Server");
 7    Set allMethods = server.getAllMethods();
 8    for (MethodNode method : allMethods ) {
 9      assertTrue(m.getVisibility().
10        equals(Modifier.PUBLIC));
11      assertTrue(m.getReturnType().equals("void"));
12      Set exceptions = m.getThrownExceptions();
13      assertTrue(exceptions.isEmpty());
14    }
15  }
16 }
```

A importância dessa regra reside no fato de que clientes externos utilizam os serviços providos pela classe `Server` e deseja-se que esses clientes tenham o mínimo de conhecimento a respeito das outras entidades do código. Por esse motivo é preciso evitar que os cliente não tenham que capturar exceções ou lidar com tipos de retorno vinculados ao código do OurBackup.

Com a execução desse teste de design foi possível verificar que a implementação da classe `Server` estava de acordo com a regra de design especificada, ou seja, nenhuma violação foi identificada.

Foi possível manualmente observar que a classe `Server` de fato estava de acordo com a regra especificada. Em discussão com o líder do projeto OurBackup, foi identificado que essa regra havia sido violada em determinado momento da evolução do software, mas que, após identificada, o código foi alterado para refletir o design idealizado. O teste de design composto para essa regra foi adicionado à suite de testes do OurBackup, fazendo com que a implementação seja periodicamente checada.

Embora os estudos de caso realizados tenham sido de pequeno porte, as análises e observações realizadas reforçam ainda mais a ideia de que testes de design são facilmente aplicáveis ao processo de desenvolvimento. É nítido que, apenas analisando dois casos específicos, essas observações não podem ser generalizadas. No entanto, através de nossa experiência com a aplicação de testes de design em ambientes reais, percebemos que a abordagem tem um forte apelo ágil, por promover a discussão de design, facilitar sua mudança e desempenhar o papel de documentação executável do design.

Capítulo 5

Trabalhos Relacionados

Este capítulo descreve os trabalhos que estão relacionados a abordagem de testes de design. Esses trabalhos foram selecionados por estarem inseridos no contexto de análise estática e verificação de propriedades estruturais de software. A ideia é descrever cada trabalho relacionado e conduzir uma discussão comparativa levando em consideração a abordagem de testes de design.

5.1 Uma Linguagem Baseada em Aspectos para Checagem de Regras de Design

Limitações no modelo da linguagem AspecJ motivaram Morgan et al. [MDVW07] a desenvolverem uma linguagem de domínio específico, chamada *Program Description Language* (PDL), baseada no paradigma orientado a aspectos, que tem como objetivo checar se regras de *design* estão sendo obedecidas. Com PDL, as regras são checadas estaticamente, isto é, em tempo de compilação.

PDL é semelhante à AspectJ no que diz respeito à sintaxe da linguagem. Todavia, seu modelo de pontos de junção (*joinpoints*) é baseado em entidades da aplicação (e. g., classes, métodos, atributos etc). Este fator faz com que seja possível expressar uma grande quantidade de regras de design que se baseiam nessas entidades. De fato, PDL é capaz de expressar e checar estaticamente diversas regras que AspectJ não é capaz. Um exemplo de regra deste tipo é a Lei de Demeter (*Law of Demeter*), cuja checagem utilizando AspectJ só é possível

ser efetuada dinamicamente.

O Código 5.1 é um exemplo simples de como descrever uma regra utilizando PDL. Neste caso, PDL foi utilizada para identificar a presença de atributos públicos no código, o que é uma prática indesejável no contexto de programação orientada a objetos, uma vez que não favorece o encapsulamento.

Código 9 Exemplo de checagem de regras de design com PDL.

```
1 field(sourceType) && public
2 : "Do not declare visible instance fields"
```

Como podemos observar no Código 5.1, PDL permite referenciar entidades do sistemas de acordo com o seu tipo (`field(sourceType)`), além de suas propriedades estruturais (`public`).

Ao contrário de outras abordagens imperativas como o FindBugs [HP04], FXCop [fxc] e a abordagem proposta neste documento, PDL é uma linguagem declarativa. Este fator é responsável por uma maior concisão na expressão das regras de *design*. Na avaliação conduzida pelos autores, PDL é comparada com FXCop levando em consideração o tamanho (em número de linhas) das regras especificadas e, de acordo com os autores, as regras descritas em PDL são mais concisas. No entanto, pelo mesmo fator, PDL é menos expressiva do que FXCop. Na avaliação da expressividade, PDL foi capaz de expressar 37% das regras checadas por FXCop.

Essa perda de expressividade está relacionada a vários fatores. Dentre eles, a falta de algumas primitivas que podem ser usadas para checagem de importantes aspectos estruturais do código. Por exemplo, com PDL não é possível checar a seguinte regra:

"Não capture exceções genéricas."

Para checar tal regra, é preciso avaliar uma expressão do tipo *catch* e analisar o tipo da exceção capturada. Contudo, PDL não é capaz de efetuar tal atividade, enquanto que essa checagem é possível com testes de *design*.

5.2 Linguagem de Restrição de Dependência

Terra e Valente [TV09] propuseram um sistema de verificação estática de arquitetura semelhante à abordagem de testes de design. A ideia baseia-se na criação de uma linguagem de restrição de dependência para a especificação de dependências aceitáveis e inaceitáveis de acordo com a arquitetura do sistema. Após especificadas, as restrições são checadas automaticamente através de uma ferramenta, intitulada DCL Check, que automaticamente detecta pontos do código fonte que violam as restrições especificadas. Assim como a abordagem de testes de design, a ferramenta DCL Check trata ausência e divergência com a mesma semântica. Isto é, como cenários em que o código não implementa o design planejada.

A linguagem proposta pelos autores objetiva permitir a definição de restrições de dependência entre módulos[1]. Através da Linguagem de Retrição de Dependência, é possível especificar regras permissivas, proibitivas e mandatórias.

Regras permissivas especificam que somente classes de um determinado componente podem depender de classes de outro componente. Por exemplo, o Código 5.2 é um exemplo de especificação de uma regra permissiva utilizando a linguagem de retrição de dependência proposta.

Código 10 Exemplo de especificação de regra permissiva.

```
1 module Helper : foo.bar.helper.*
2 module Exception : foo.bar.exception.*
3 only Helper can-throw Exception
```

Regras proibitivas são descritas de maneira semelhante à regras permissivas. No entanto, utiliza-se a palavra chave *cannot* para denotar que um determinardo módulo não pode depender de outro.

Por último, é possível também descrever restrições de dependência que obrigatoriamente devem estar presente no código da aplicação. A descrição desse tipo de regra faz uso da palavra chave *must*. Por exemplo, o Código 5.2 é um exemplo de regra mandatória. Através de sua análise, é possível concluir que todas as classes do pacote `foo.bar.services` devem implementar a interface `foo.bar.interfaces.Service`.

[1] O conceito de módulo adotado pela abordagem é um conjunto de classes.

Código 11 Exemplo de especificação de regra mandatória.

```
1 module Services : foo.bar.services.*
2 must-implement foo.bar.interfaces.Service
```

Outro ponto semelhante a abordagem de testes de design é o fato de que a linguagem definida pelos autores leva em consideração as relações entre módulos e suas possíveis formas: Decomposição, Uso, Camadas e Generalização. Até onde podemos observar, a diferença neste aspecto é que testes de design leva em consideração a relação *Catch* como forma de dependência entre duas classes. Já a linguagem definida pelos autores desconsidera essa relação. Por outro lado, testes de design não dão suporte à escrita de regras que baseiam-se em anotações no código, atividade suportada pela linguagem de restrição de dependência.

Como é possível notar, na abordagem proposta por Terra e Valente, as regras são especificadas em uma linguagem diferente da aplicação sob verificação. Este aspecto pode influenciar na adoção da abordagem. O que realmente se advoga aqui é que escrever regras de design na mesma linguagem de programação que a aplicação sob verificação é um processo que catalisa adoção de uma abordagem de verificação.

5.3 Bridging the Software Architecture Gap

Lindvall e Muthing [LM08] desenvolveram uma ferramenta, denominada *Software Architecture Visualization and Evaluation* (*SAVE*), que visa conectar a implementação com a arquitetura desejada através da checagem de violações de restrições arquiteturais previamente especificadas.

A ferramenta *SAVE* extrai a partir do código fonte da aplicação um modelo visual da arquitetura presente na implementação e compara este modelo com o especificado pelo usuário. Esta comparação leva em consideração os mapeamentos, feitos manualmente, entre o modelo extraído e o modelo especificado. Violações dos mapeamentos são identificadas e reportadas para posterior investigação e reparação das divergências.

Apesar de os interesses deste trabalho estarem alinhados com a abordagem de testes de *design*, existem diferenças significativas entre as duas abordagens. Dentre essas diferenças, destaca-se o fato de que *SAVE* é uma ferramenta que expõe visualmente os modelos a serem

comparados. Esta característica implica em uma melhora no entendimento da estrutura da aplicação, uma vez que visualização está intrinsecamente relacionada a compreensão de *software*. Todavia, a manutenção desses modelos é um tanto custosa, pois quando uma entidade é adicionada ao código, é preciso atualizar manualmente o modelo que descreve a arquitetura desejada, além dos mapeamentos entre os dois modelos. Por fim, SAVE restringe-se a dar suporte à especificação de restrições de comunicação entre módulos e entidades do sistema. Por outro lado, como foi mostrado nesta dissertação, testes de *design* podem especificar diferentes tipos de regras de *design*, não somente as que referenciam comunicação entre módulos.

5.4 Software Reflexion Models

Software Reflexion Models é uma abordagem proposta por Murphy et al. [MNS95] que visa manter a sincronia entre modelos de alto-nível e o código-fonte de um sistema. Isto é feito através da especificação manual de um modelo de alto-nível e o seu mapeamento em relação as entidades do código-fonte. A partir dessas informações, uma ferramenta computa as divergências entre o modelo especificado e a implementação do sistema.

De fato, é possível checar se a implementação de um sistema está de acordo com regras de *design* utilizando *Software Reflexion Models*. Contudo, essa checagem estaria restrita à regras de *design* que especificam comunicação entre módulos. Além disso, por ser um processo interativo, construir modelos e mapeamentos sempre que o código é alterado torna a checagem laboriosa. Por estes motivos, *Software Reflexion Models* é mais adequada ao contexto de compreensão de programas. A ideia por trás da abordagem é que, a partir das divergências identificadas, o usuário possa aumentar o seu conhecimento a respeito da implementação e reparar essas divergências para que o modelo de alto-nível reflita a implementação do sistema.

5.5 ArchJava

ArchJava [ACN02] é uma extensão da linguagem Java[2] desenvolvida com intuito de garantir integridade de comunicação em um sistema, isto é, garantir que os componentes da implementação somente se comunicam com os componentes aos quais estão diretamente conectados na descrição da arquitetura.

ArchJava provê suporte à descrição de aspectos arquiteturais do sistema através da adição de conceitos como componentes, portas e conexões à linguagem Java. Uma vez especificado parte da arquitetura, *ArchJava* verifica se há sincronia, no que diz respeito a integridade de comunicação, entre essa especificação e a implementação.

Existem diversos aspectos que diferenciam *ArchJava* da abordagem de testes de *design*. Em relação a forma como são descritos os aspectos estruturais a serem checados, ArchJava requer o aprendizado de novos conceitos que são adicionados a linguagem Java. Além disso, o seu uso está condicionado ao uso do compilador da linguagem. Por fim, *ArchJava* se restringe a checar integridade de comunicação, não provendo meios para especificação e checagem de outros importantes aspectos estruturais, atividades que são suportadas por testes de design.

5.6 AspectJ

Regras de *design* muitas vezes são investigadas no domínio de Programação Orientada a Aspectos (POA), uma vez que essas regras frequentemente referenciam diversas entidades (pacotes, classes, atributos etc) presentes em diferentes partes do código, podendo ser então caracterizadas como interesses transversais da aplicação. Desta maneira, algumas abordagens [LLW03; LL00] utilizam AspectJ [KHH+01], uma extensão orientada a aspectos de propósito geral da linguagem Java, para checagem de regras de *design*.

No entanto, por limitações no modelo da linguagem, AspectJ não é adequada para verificação estática de algumas propriedades estruturais, sendo mais aplicável na verificação dinâmica. Por exemplo, Lieberherr et al. mostraram que não é possível utilizar AspecJ com o intuito de checar em tempo de compilação se a lei de Demeter [Lie89] está sendo obe-

[2] www.java.com

decida pela implementação. Neste caso, os autores deste trabalho propuseram extensões na linguagem AspectJ para tornar possível a verificação estática dessa regra de *design*.

Um estudo efetuado por Shomrat e Yehudai [SY02] explorou o uso de POA, através da linguagem AspectJ, para a checagem de regras de design. Nesse estudo, os autores identificaram algumas regras de *design* que não são passíveis de verificação estática utilizando AspectJ:

- Nomeação de entidades
- Relações de herança
- Relações entre classes

Esses tipos de regras baseiam-se em aspectos estruturais do código, por exemplo, relacionamento de herança entre classes. Neste contexto, AspectJ é capaz de possibilitar que uma classe herde de outra classe, todavia, não é capaz de prevenir essa relação. AspectJ não é capaz de checar regras desse tipo em tempo de compilação, sendo mais apropriado o seu uso no domínio de interação entre objetos, ou seja, aspectos dinâmicos da aplicação.

5.7 CodeQuest

CodeQuest [HVdMDV05] é uma abordagem que combina a utilização de um banco de dados com uma linguagem baseada no paradigma lógico para localizar propriedades estruturais no código-fonte. A ideia por trás deste trabalho é utilizar uma linguagem semelhante a PROLOG, denominada DataLog, para a escrita de consultas a um banco de dados para localizar pontos de interesse no código-fonte do programa. Esses pontos de interesse incluem regras de design, *bug patterns*, *bad smells* etc. O Código 5.7 representa uma consulta para apontar a presença de atributos com visibilidade pública e modificáveis. Trata-se de uma situação que deve ser evitada, principalmente na construção de sistemas orientados a objeto, pois expões variáveis à mudanças a partir de qualquer ponto do código.

A abordagem CodeQuest é comparável a este trabalho no que diz respeito a localização de pontos de interesse no código. No entanto, a forma como CodeQuest descreve as consultas é substancialmente diferente de testes de design. Um primeiro aspecto a ser notado é

Código 12 Exemplo de checagem de regras de design com Datalog.

```
1 q1(T,F) :- type(T),child(T,F),field(F),
2         modifier(F,public),
3            not(modifier(F,final))
```

a utilização do paradigma lógico para construção das consultas. Por um lado, este aspecto torna as regras mais concisas quando comparadas ao paradigma imperativo empregado na construção de testes de design. No entanto, como foi demonstrado por outro trabalho relacionado [MDVW07], linguagens declarativas perdem em poder de expressão quando se leva em consideração a variedade de tipos de regras que podem ser especificadas.

5.8 Design Fragments

Fairbanks et al. [FGS06] criaram o termo "fragmento de design" (*Design Fragments*) para designar a forma como um programa deve interagir com um *framework* para alcançar determinado objetivo. Um fragmento de design, segundo os autores, é composto por duas partes. A primeira descreve o contrato (construção de classes, métodos, atributos etc) que o programador deve cumprir para a utilização do *framework*. A segunda descreve quais partes do *framework* irão se comunicar com o código do programador. Esses fragmentos de design podem ser utilizados para diversos fins, incluindo acelerar o entendimento e aprendizagem do programador em relação ao uso do *framework*. Para tal objetivo, os autores construíriam um catalago com onze fragmentos de design relacionados a *frameworks* Java amplamente utilizados por diversos projetos.

No contexto de verificação de conformidade estrutural, os autores deste trabalho indicam que, através de ferramentas de análise estática, é possível checar a conformidade estrutural entre o código e um determinado fragmento de design. Através desse processo, o programador teria ciência das violações de regras de design presentes no seu código. Neste fator reside a classificação desse trabalho como trabalho relacionado a abordagem proposta neste documento. De fato, fragmentos de design também se apresentam como uma forma de expressar regras de design.

A especificação de fragmentos de design é viabilizada através de uma linguagem criada

pelos autores do trabalho. Nessa linguagem, os fragmentos de design são especificados através de documentos XML contendo entidades do código e suas relações. O Código 5.8 mostra um exemplo de especificação de um fragmento de design.

Código 13 Especificação de um fragmento de design

```
1 <class name="RoleThread" provided="no">
2     <implementsInterface
3         name="java.lang.Runnable" />
4     <method name="run" returnValue="void">
5     </method>
6 </class>
```

Embora seja possível a verificação de conformidade estrutural utilizando fragmentos de design, sua aplicação está mais ligada ao aprendizado no uso de *frameworks*. Fragmentos de design são guias que descrevem como um programa deve interagir com o código de um *framework*, bem como os pontos em que se dá essa interação. Por isso a importância da criação de um catálogo de fragmentos de design para *frameworks* Java amplamente utilizados.

5.9 Ferramentas: FindBugs, Jlint, PMD e CheckStyle

O uso de análise estática para checagem estrutural de software tem sido cada vez mais difundido no ambiente acadêmico e empresarial. Esta afirmativa é plenamente suportada pelo grande número de ferramentas que utilizam análise estática para diversos fins, tais como, detectar possíveis faltas no software e checar se a codificação está de acordo com determinado estilo. A popularidade dessas ferramentas também é visível. Por exemplo, *FindBugs* [HP04] tem sido extensivamente (700.000 downloads desde Julho de 2008) utilizada pela comunidade.

FindBugs é uma ferramenta que utiliza análise estática para checar a presença de *bug patterns* no código-fonte de aplicações Java. Segundo Hovermeyer e Pugh, *bug patterns* são trechos de códigos que podem acarretar em erros na aplicação. Tipicamente, são introduzidos através do uso equivocado de *frameworks*, APIs e estruturas da linguagem de programação. Por exemplo, invocar o método `toString()` em um *array* retorna uma *String* representando o *hashcode* do objeto, ao invés de imprimir uma representação do objeto que seja

informativa e fácil de ser entendida por humanos, como recomenda a documentação. Este é um erro comum na linguagem Java e que é facilmente contornado fazendo uso do método `toString(Object[] a)` da classe `Arrays`.

O fato da ferramenta, através de um conjunto de regras pré-definidas, ser capaz de checar uma grande variedade de *bug patterns* automaticamente, contribui para que *FindBugs* seja amplamente utilizada. No entanto, a ferramenta não dá suporte à construção de regras de design, atividade suportada pela abordagem de testes de design. É sabido que várias regras de design são dependentes da aplicação a qual está sendo checada. Por este motivo, se faz necessário um mecanismo para expressar e checar tais regras.

Entre outros aspectos, *FindBugs* é uma ferramenta que difere da abordagem de testes de design por tratar de aspectos de mais baixo nível do código, como possíveis referências nulas e tratamento de *streams*. Neste contexto *JLint* [Art06] e *PMD* [Cop03] também se encaixam na taxonomia de ferramentas para verificação estática de *bug patterns* no código. De fato, essas ferramentas são muito semelhante a *FindBugs* no que diz respeito ao seu funcionamento e propósito.

Ainda no contexto de ferramentas que aplicam análise estática para verificação de propriedades do código, *CheckStyle* [Bur05] utiliza análise estática para verificar se regras de estilo de codificação estão sendo obedecidas durante a implementação. Enquanto ferramentas como *FindBugs* e *JLint* mantêm o foco na procura por faltas que podem ocasionar falhas durante a execução do software e, por consequência, afetar o cliente do software, ferramentas como *CheckStyle* focam-se em detectar violações de regras de estilo de codificações.

Capítulo 6

Conclusão

Nesta dissertação foi apresentada uma técnica, intitulada testes de design, que tem como objetivo automatizar a checagem estrutural de conformidade entre regras de design de baixo nível e implementação. Um teste de design é uma forma de especificar um regra de design de baixo nível, além de ser um mecanismo automático para checar se a implementação está em conformidade com essa regra especificada.

Testes de design são escritos na mesma linguagem de programação da aplicação sob verificação. Essa estratégia foi adotada para simplificar a construção de regras de design. Embora vários trabalhos acadêmicos tenham sido propostos para atingir tal objetivo, o mecanismo mais utilizado para checagem de conformidade é inspeção manual de código. A análise do estado da arte nessa área nos levou a concluir que esse cenário se dá em decorrência da complexidade de se aplicar essas abordagens. Muitas delas requerem o aprendizado de uma linguagem diferente da aplicação sob verificação, o que aumenta a sua curva de aprendizado pode acarretar na não adoção da abordagem. Isto posto, foi possível detectar que o estado da prática também poderia ser melhorado, no que diz respeito a escalabilidade das soluções adotadas. Inspeção manual de código, por ser um trabalho manual, torna-se extremamente laborioso à medida que o software cresce. Além disso, é importante destacar o fato de estar sujeito a erros, por se tratar de uma atividade conduzida por humanos.

A proposta foi desenvolvida em paralelo com a ferramenta que a suporta. O DesignWizard - um analisador estático que extrai informações do bytecode de aplicações Java e modela essas informações em um grafo onde as entidade extraídas são os vértices e as relações entre as entidades são as arestas que ligam os vértices. Feito isso, o DesignWizard provê uma API

contendo serviços/métodos que devem ser utilizados para a construção de testes de design para aplicações Java. Em nossa abordagem, a fase de checagem e publicação dos resultados oriundos do teste é efetuada pelo framework de teste JUnit.

Dois aspectos foram utilizados para avaliar o suporte ferramental construído. Esses dois aspectos se relacionam aos problemas destacados no estado da arte (usabilidade) e no estado da prática (escalabilidade) na área de checagem de conformidade estrutural entre regras de design e implementação. Os experimentos realizados para avaliar a usabilidade da API do DesignWizard indicaram que os programadores não tiveram dificuldades em escrever testes de design utilizando a API, sobretudo pelo fato da linguagem ser a mesma linguagem de programação a qual eles já possuíam conhecimento. De fato, através do protocolo *Think Aloud* foi possível observar que a API cumpriu, na maioria dos casos, com as expectativas dos desenvolvedores na atividade de composição dos testes de design. Por fim, avaliamos a escalabilidade do DesignWizard, através dos resultados, foi possível concluir a eficiência da ferramenta na extração de informações e na checagem de regras de design, mesmo quando submetida a grandes projetos.

6.1 Resultados e Contribuições

Em termos concretos, as contribuições deste trabalho são:

- Proposta da abordagem de verificação de conformidade baseada em testes de design [BGF09]. Além disso, a abordagem de testes de design obteve o primeiro lugar na competição *Jazoon Rookie* [oJT09], realizada na Suiça. Essa competição avalia trabalhos realizados na linguagem Java e os elenca de acordo com sua relevância e aplicabilidade. Após uma disputa entre dezenas de trabalhos, a abordagem de testes de design obteve o primeiro lugar no concurso.

- Desenvolvimento de uma ferramenta que dá suporte à abordagem de testes de design - O DesignWizard.

- Aplicação da abordagem de testes de design na verificação de conformidade entre diagramas de classe especificados em UML e implementação [PBR08].

- Aplicação da ferramenta DesignWizard no contexto de uma técnica para análise de impacto de mudanças em código-fonte [HGF+08].

- Aplicação da ferramenta DesignWizard no ambiente empresarial. A ferramenta foi utilizada pela empresa CPM Braxis [CPM10] com intuito aumentar o controle de qualidade de seu processo de desenvolvimento. O DesignWizard foi utilizado no contexto de projetos reais e ajudou a organização a progredir do nível CMMI 3 para o nível 5, segundo foi declarado pela própria empresa;

- Um relato de experiência a respeito do experimento conduzido para avaliação de usabilidade da API do DesignWizard.

Em relação ao último item, é preciso destacar que a metodologia utilizada para avaliação da usabilidade da API do DesignWizard e o relato de experiência que a sua condução originou uma importante contribuição deste trabalho. Executar experimentos de usabilidade para APIs é uma atividade relativamente nova e que vem ganhando expressão ultimamente. Acreditamos ser de importante valia o fato de experimentar nossa solução com diversos programadores e aplicando casos de testes reais. De fato, conduzir experimento dessa natureza é laborioso. No entanto, os dados extraídos desse tipo de experimento são relevantes e importantes para avaliar a usabilidade da API, pois são feitos pelos potenciais clientes da ferramenta.

6.2 Trabalhos Futuros

Como trabalho futuro, pretendemos fazer um estudo amplo sobre a escrita de testes de design que especifiquem padrões de projeto. A ideia é construir uma suite de testes de design que verifiquem se padrões de projeto estão sendo utilizados de maneira correta pela implementação.

Além disso, um possível trabalho futuro é estender a ferramenta para que possa ser capaz de verificar propriedades dinâmicas de software. Atualmente, isso não é possível pois o DesignWizard analisa estaticamente as aplicações. No entanto, a inclusão de análise dinâmica pode fazer com que a API evolua de maneira a suportar especificação de testes de design que verifiquem, por exemplo, se uma determinada ordem de chamada de métodos está sendo

cumprida. Um trabalho prévio [PBR08], desenvolvido pelo Grupo de Métodos Formais (GMF), realiza a transformação de modelos UML, mais precisamente diagramas de classe, para testes de design. A extensão do DesignWizard para incorporar aspectos dinâmicos pode também viabilizar a transformação de diagramas comportamentais de UML. Por exemplo, seria possível construir um teste de design a partir de um diagrama de sequência.

Por último, um outro trabalho que pode ser realizado como forma de extensão deste trabalho de mestrado é o melhoramento do suporte ferramental da técnica no que diz respeito a checagem em tempo real de violações no código fonte da aplicações. Neste caso, a verificação de conformidade não seria feita a cada execução do teste de design. Sempre que o código da aplicação for modificado, o DesignWizard pode apontar violações de regras de design. Isso pode ser feito através da construção de *plugins* para o ambiente de desenvolvimento da ferramenta Eclipse [Fou06]. A ideia é que a verificação seja semelhante ao processo de compilação do código. Isto é, à medida que o desenvolvedor escreve código, a ferramenta pode destacar (sublinhar) as violações de regras.

Referências Bibliográficas

[ACN02] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187–197. ACM New York, NY, USA, 2002.

[Art06] C. Artho. Jlint-find bugs in java programs, 2006.

[BCK03] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. ProQuest Information and Learning Company, 2003.

[BDA+99] P. Bourque, R. Dupuis, A. Abran, J.W. Moore, and L. Tripp. The Guide to the Software Engineering Body of Knowledge. *IEEE SOFTWARE*, pages 35–44, 1999.

[BGF09] J. Brunet, D. Guerrero, and J. Figueiredo. Design tests: An approach to programmatically check your code against design rules. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 255–258, 2009.

[BLC02] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and Extensible Component Systems*, 2002.

[BMS+08] J.K. Beaton, B.A. Myers, J. Stylos, S.Y.S. Jeong, and Y.C. Xie. Usability evaluation for enterprise SOA APIs. In *Proceedings of the 2nd international workshop on Systems development in SOA environments*, pages 29–34. ACM New York, NY, USA, 2008.

[Bro95] F.P. Brooks. *The mythical man-month*. Addison-Wesley Reading, MA;, 1995.

[Bud03] D. Budgen. *Software Design*. Addison Wesley, 2003.

[Bur05] O. Burn. Checkstyle homepage. *URL http://checkstyle. sourceforge. net/. last accessed in March*, 2005.

[CBA+06] W. Cirne, F. Brasileiro, N. Andrade, L.B. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the World, Unite!!! *Journal of Grid Computing*, 4(3):225–246, 2006.

[Cla03] S. Clarke. API Usability and the Cognitive Dimensions Framework, 2003.

[Cla04] S. Clarke. Measuring API usability. *DOCTOR DOBBS JOURNAL*, 29(5):1–5, 2004.

[Cla05] S. Clarke. Describing and measuring API usability with the cognitive dimensions. In *Cognitive Dimensions of Notations 10th Anniversary Workshop*. Citeseer, 2005.

[Cop03] T. Copeland. Static analysis with pmd. *OnJava. com*, 2003.

[CPM10] CPMBraxis. CPMBraxis, 2010. http://www.cpmbraxis.com/.

[ESM07a] B. Ellis, J. Stylos, and B. Myers. The factory pattern in api design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, pages 302–312. IEEE Computer Society Washington, DC, USA, 2007.

[ESM07b] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in api design: A usability evaluation. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 302–312, Washington, DC, USA, 2007. IEEE Computer Society.

[Fag76] ME Fagan. Design and code inspections to reduce errors in program development. *IBM Journal of Research and Development*, 15(3):182, 1976.

[FGS06] George Fairbanks, David Garlan, and William Scherlis. Design fragments make using frameworks easier. *SIGPLAN Not.*, 41(10):75–88, 2006.

[FL02] A. Forward and T.C. Lethbridge. *The relevance of software documentation, tools and technologies: a survey*. 2002.

[Fou06] E. Foundation. Eclipse, 2006.

[fxc] FXCop Team Page. *http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx, último acesso em Fevereiro de 2009.*

[GB99] E. Gamma and K. Beck. JUnit. *URL: http://www.junit.org (Acesso em Outubro./2008)*, 1999.

[GP96] T.R.G. Green and M. Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[Gra97] R.B. Grady. *Successful software process improvement*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1997.

[GS94] RB Grady and TV Slack. Key lessons in achieving widespread inspection use. *Software, IEEE*, 11(4):46–57, 1994.

[HGF+08] L. Hattori, D. Guerrero, J. Figueiredo, J. Brunet, and J. Damasio. On the precision and accuracy of impact analysis techniques. *Computer and Information Science, 2008. ICIS 08. Seventh IEEE/ACIS International Conference on*, pages 513–518, May 2008.

[HP04] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92–106, 2004.

[HVdMDV05] E. Hajiyev, M. Verbaere, O. de Moor, and K. De Volder. Codequest: querying source code with datalog. In *Conference on Object Oriented Programming Systems Languages and Applications*, pages 102–103. ACM New York, NY, USA, 2005.

[IEE98] IEEE. Ieee standard for software reviews. *IEEE Std 1028-1997*, Mar 1998.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An Overview of AspectJ. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 327–353, 2001.

[KP07] J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. In *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA'07), IEEE Computer Society, S*, volume 12, 2007.

[Lew82] C. Lewis. Using the thinking-aloud method in cognitive interface design. *IBM Research Report RC*, 9265(2):17, 1982.

[Lie89] KJ Lienberherr. Formulations and Benefits of the Law of Demeter. *ACM SIGPLAN Notices*, 24(3):67–78, 1989.

[LL00] M. Lippert and C.V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd international conference on Software engineering*, pages 418–427. ACM New York, NY, USA, 2000.

[LLW03] K. Lieberherr, D.H. Lorenz, and P. Wu. A case for statically executable advice: checking the law of demeter with AspectJ. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 40–49. ACM New York, NY, USA, 2003.

[LM08] Mikael Lindvall and Dirk Muthig. Bridging the software architecture gap. *Computer*, 41(6):98–101, June 2008.

[LSF03] TC Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: the state of the practice. *Software, IEEE*, 20(6):35–39, 2003.

[MDVW07] C. Morgan, K. De Volder, and E. Wohlstadter. A static aspect language for checking design rules. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 63–72. ACM Press New York, NY, USA, 2007.

[MNS95] G.C. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: bridging the gap between source and high-level models. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, pages 18–28. ACM New York, NY, USA, 1995.

[oJT09] The International Conference on Java Technology. Jazoon Rookie Contest, 2009. http://jazoon.com/.

[Oli07] M.I.S. Oliveira. Ourbackup: Uma soluçao p2p de backup baseada em redes sociais. *Master's thesis, Universidade Federal de Campina Grande*, 2007.

[Par94] DL Parnas. Software aging. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 279–287, 1994.

[PBR08] W. Pires, J. Brunet, and F. Ramalho. UML-based design test generation. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 735–740. ACM New York, NY, USA, 2008.

[Rus91] Glen W. Russell. Experience with inspection in ultralarge-scale development. *IEEE Softw.*, 8(1):25–31, 1991.

[SC07a] J. Stylos and S. Clarke. Usability Implications of Requiring Parameters in Objects' Constructors. In *Proceedings of the 29th international conference on Software Engineering*, pages 529–539. IEEE Computer Society Washington, DC, USA, 2007.

[SC07b] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 529–539, Washington, DC, USA, 2007. IEEE Computer Society.

[Sin98] J. Singer. Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance*, pages 139–145. IEEE Computer Society Washington, DC, USA, 1998.

[SM08] J. Stylos and B.A. Myers. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 105–112. ACM New York, NY, USA, 2008.

[Som06] I. Sommerville. *Software Engineering*. Addison-Wesley, 2006.

[SSC96] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. *icse*, 00, 1996.

[SY02] M. Shomrat and A. Yehudai. Obvious or not?: regulating architectural decisions using aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 3–9. ACM Press New York, NY, USA, 2002.

[TV09] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Softw. Pract. Exper.*, 39(12):1073–1094, 2009.

[TvdH07] R.N. Taylor and A. van der Hoek. Software design and architecture: The once and future focus of software engineering. *Future of Software Engineering*, 2007.

[TWI05] J. Tian, J. Wiley, and W. InterScience. *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*. Wiley, 2005.

[vDGB05] H.W. van Dijk, B. Graaf, and R. Boerman. On the systematic conformance check of software artefacts. In *Proc. 2 ndEuropean Workshop on Software Architecture (EWSA 2005). Springer-Verlag, June*. Springer, 2005.

Apêndice A

Metamodelo De Design

Neste apêndice apresentamos com detalhes o metamodelo de design concebido para descrever a forma como idealizamos ser um design de baixo nível. A Figura A.1 mostra as entidades pertencentes ao metamodelo de design, bem como as possíveis relações entre essas entidades. É importante destacar que todas as relações são binárias e devem ser interpretadas da seguinte maneira:

Entidade chamadora <relação> *Entidade chamada*

O metamodelo de design possui sete entidades e suas respectivas relações. A seguir descrevemos com detalhes todas as relações possíveis entre essas entidades.

A.1 Método X Tipo

éContidoPor Indica que um método é declarado em um determinado tipo. A relação *Contém* é inversa a essa relação, ou seja, indica que um tipo contém um método.

Carrega Indica que um método carrega um tipo. Em Java, essa relação se dá através da referência a um determinado tipo como demonstrado no Código 14.

Recebe Relação entre um método e os tipos dos seus parâmetros. Por exemplo, para o Código 15, a relação extraída é:

`method` *recebe* `String`.

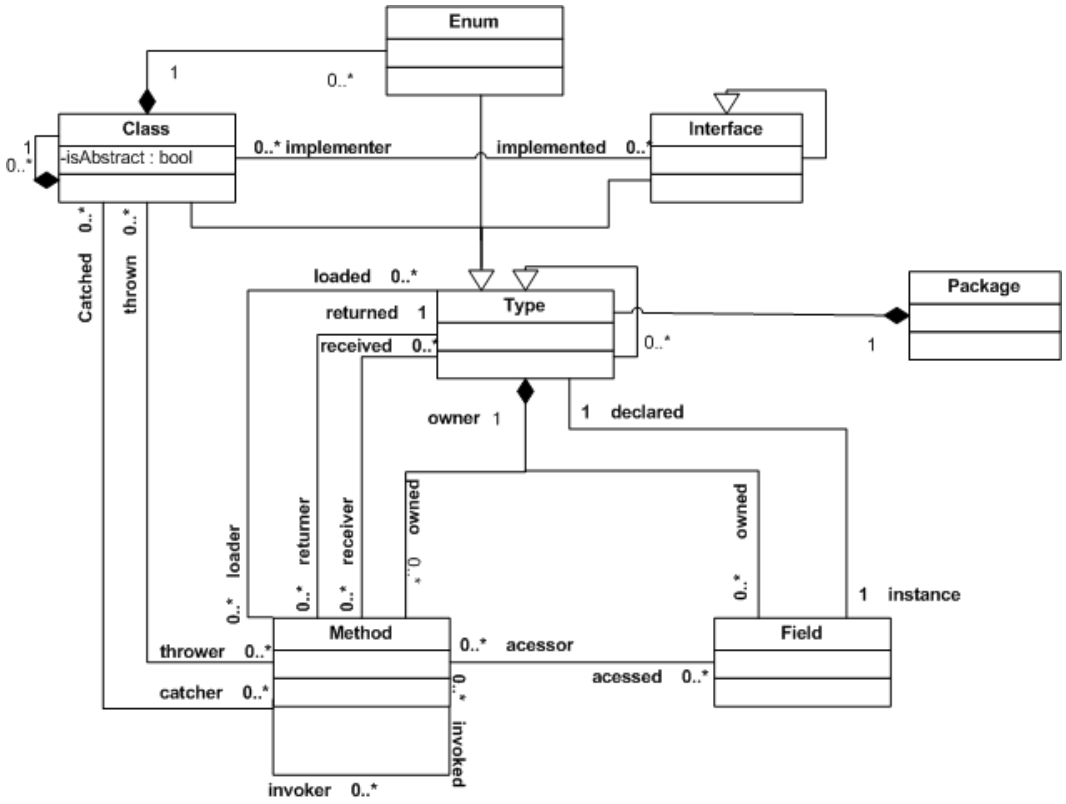


Figura A.1: Metamodelo de Design

Retorna Relação utilizada para especificar o tipo de retorno de um método. Por exemplo:

`Soma.soma()` *retorna* `BigInteger`.

A.2 Método X Classe

Captura Indica que há uma captura de exceção dentro do corpo de um método. Um exemplo de código que contém essa relação é o Código 16, já que o método `method()` captura a exceção `InexistentEntityException`.

Código 14 Exemplo de relação *carrega*.

```
public static Test suite() {
    TestSuite suite = new TestSuite("Test for DesignWizard");
    suite.addTest(new TestSuite(RelationTest.class));
}
```

Código 15 Exemplo de relação *recebe*.

```
public void method(String s) {
    //
}
```

Código 16 Exemplo de relação *captura*.

```
public void method() {
    try{
        ClassNode c = dw.getClass("ClassExample");
    } catch (InexistentEntityException e) {}
}
```

Lança Indica que um método possui em sua assinatura a cláusula *throws* indicando o possível lançamento de uma exceção. O Código 17 ilustra um exemplo dessa relação.

Código 17 Exemplo de relação *lança*.

```
public void method() throws InexistentEntityException{
    ClassNode c = dw.getClass("ClassExample");
}
```

A.3 Método X Método

Invoca Indica que um método invoca um outro método da aplicação. A relação *éInvocadoPor* é a relação inversa, isto é, indica que um método é invocado por outro.

A.4 Método X Atributo

Acessa Indica que um método acessa um atributo. Nesse cenário, acesso é qualquer referencia a um atributo dentro de um método, seja essa referência de leitura ou escrita.

A.5 Atributo X Tipo

éInstanciaDe Indica que um atributo é de um determinado tipo. Podendo esse tipo ser Enum, Classe, ClasseAbstrata ou Interface.

éContidoPor Indica que um atributo é declarado em um determinada tipo.

A.6 Pacote X Tipo

Contém Indica que um pacote contém um determinado tipo. A relação inversa consiste em identificar que um tipo é declarado em um pacote. Essa relação recebe o nome de *éDeclaradoEm*.

A.7 Classe X Classe

Contém Indica que uma classe contém outra classe. Este é o caso da especificação de classes internas e anônimas. Nesse caso há também a presença da relação inversa *éDeclaradoEm*.

A.8 Classe X Interface

Implementa Essa relação identifica que uma classe implementa uma determinada interface. Por exemplo:

Código 18 Exemplo de relação *implementa*.

```
public class LinhaDeExecucao implements Runnable{}
```

A.9 Classe X Enum

Contém Essa relação identifica que uma classe contém um *Enum*. Naturalmente, um Enum pode ser contido por uma classe. Trata-se da relação inversa *éDeclaradoEm*.

A.10 Tipo X Tipo

Herda Identifica uma relação de herança entre um tipo e outro. Por exemplo:

Código 19 Exemplo de relação *extends*.

```
public class Integer extends Number{}
```

Apêndice B

Questionário para Avaliação da Usabilidade da API do DesignWizard

Neste apêndice, está descrito o questionário elaborado e aplicado com intuito de coletar dados a respeito da opinião dos desenvolvedores em relação a API do DesignWizard.

Inicialmente, ao entrevistado, foi requisitado seu nome, idade e experiência com a linguagem Java. Além disso, as seguintes perguntas foram realizadas:

1. Você já havia escrito teste de design com o DesignWizard? (sim ou não)
2. Quão fácil foi traduzir o pseudocódigo para código DW + JUnit? (Muito fácil, Fácil, Razoável, Difícil ou Muito difícil)
3. Quando você lê o código feito, quão fácil é saber o que cada parte faz? (Muito fácil, Fácil, Razoável, Difícil ou Muito difícil)
4. Quão fácil é saber qual classe e método usar para derterminada subtarefa? Por exemplo: saber as classes de um pacote. Ou saber se um ClassNode é interface ou classe. (Muito fácil, Fácil, Razoável, Difícil ou Muito difícil)
5. Avalie a documentação da API. (Muito Boa, Boa Razoável, Ruim ou Muito Ruim)
6. Em sua opinião, a quantidade de código necessária para compor os testes é: Grande, Justa ou Pequena? Justifique sua resposta.

7. Se você precisasse mudar o código, você teria: Muito esforço, Esforço moderado ou Pouco esforço?

8. Aponte defeitos na API do DW que se tornaram dificuldades enquanto você estava implementando um teste de design.

9. Aponte métodos ausentes na API do DW que se tornaram dificuldades enquanto você estava implementando um teste de design.

10. Aponte qualidades na API do DW que tornaram mais fácil a escrita de um teste de design.

11. Sugestione métodos para a API do DesignWizard.

12. Em resumo, qual a sua opinião em relação a usabilidade da API? Muito Fácil, Fácil, Razoável, Difícil ou Muito difícil?