UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ
BACHARELADO EM ENGENHARIA DE SOFTWARE

MÁRCIO DE SOUZA NOBRE

VERIFICAÇÃO DE CONFORMAÇÃO DE REGRAS DE DESIGN DO TRATAMENTO DE EXCEÇÃO EM JAVA

QUIXADÁ
2014

MÁRCIO DE SOUZA NOBRE

VERIFICAÇÃO DE CONFORMAÇÃO DE REGRAS DE DESIGN DO TRATAMENTO DE EXCEÇÃO EM JAVA

Trabalho de Conclusão de Curso submetido à Coordenação do Curso Bacharelado em Engenharia de Software da Universidade Federal do Ceará como requisito parcial para obtenção do grau de Bacharel.

Área de concentração: computação

Orientador Prof. Dr. Lincoln Souza Rocha

QUIXADÁ
2014

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca do Campus de Quixadá

N754v Nobre, Márcio de Souza
Verificação de conformação de regras de design do tratamento de exceção em Java / Márcio de Souza Nobre. – 2014.
49 f. : il. color., enc. ; 30 cm.

Monografia (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Software, Quixadá, 2014.
Orientação: Prof. Dr. Lincoln Souza Rocha
Área de concentração: Computação

1. Sistemas de software 2. Software – Testes 3. Java (Linguagem de programação de computador) I. Título.

CDD 005.14

MÁRCIO DE SOUZA NOBRE

VERIFICAÇÃO DE CONFORMAÇÃO DE REGRAS DE DESIGN DO TRATAMENTO DE EXCEÇÃO EM JAVA

Trabalho de Conclusão de Curso submetido à Coordenação do Curso Bacharelado em Engenharia de Software da Universidade Federal do Ceará como requisito parcial para obtenção do grau de Bacharel.

Área de concentração: computação

Aprovado em: _____ / junho / 2014.

BANCA EXAMINADORA

_____
Prof. Dr. Lincoln Souza Rocha (Orientador)
Universidade Federal do Ceará-UFC

_____
Prof. MSc. Marcio Espindola Freire Maia
Universidade Federal do Ceará-UFC

_____
Prof. MSc. Regis Pires Magalhães
Universidade Federal do Ceará-UFC

_____
Prof. MSc. Carlos Diego Andrade de Almeida
Universidade Federal do Ceará-UFC

Este trabalho é dedicado aos meus pais e ao meu avô Antônio Jasismino, que infelizmente não está mais conosco para testemunhar esse momento tão importante da minha vida.

AGRADECIMENTOS

Quero agradecer primeiramente a Deus por mais está vitória na minha vida.

Em segundo lugar, quero agradecer aos meus pais que sempre me apoiaram e veem apoiando durante toda a minha vida, me oferecendo suporte e força necessária para sempre continuar independente das adversidades que venham a aparecer. Sem minha família nada disso seria possível.

Quero agradecer ao meu orientador professor Lincoln Souza Rocha, que sempre me ajudou e acreditou em mim, foi devido ao seu apoio que este trabalho teve êxito.

Quero agradecer também a todos os meus companheiros de graduação que estiveram comigo desde o início desta jornada e que viveram junto comigo as alegrias e tristezas vividas nesse período.

Quero ressaltar entre todos os meus companheiros aquele que mais esteve próximo e pôde ver de perto minha luta para chegar até aqui. Só tenho a agradecer ao meu grande amigo Bruno Furtado, por seus conselhos e pela amizade que construímos nesses anos de curso.

Gostaria de agradecer também aos professores Flávio, Diego, Marcio e Regis pelas contribuições dadas a este trabalho.

A todos, só tenho a dizer, muito obrigado.

"Se fracassar, ao menos que fracasse ousando grandes feitos, de modo que a sua postura não seja nunca a dessas almas frias e tímidas que não conhecem nem a vitória nem a derrota. ”
(Theodore Roosevelt)

RESUMO

Os sistemas de software estão cada vez maiores em tamanho, complexidade e número de usuários, por este motivo existe uma grande probabilidade da ocorrência de erros durante a execução. Tais fatores fazem com que exista uma grande necessidade que os sistemas sejam confiáveis, ou seja, devem ser capazes de se recuperar de condições excepcionais de operação. Um componente chave para qualquer sistema confiável é o seu mecanismo de tratamento de exceção. No entanto, trabalhos recentes indicam que muitas aplicações industriais apresentam má qualidade do tratamento de exceção, isso porque os desenvolvedores tendem a negligencia-lo. Uma solução para melhorar o mecanismo de tratamento de exceção das aplicações é realizar uma verificação baseada na aplicação de regras de design ao código fonte para saber se ele está ou não em conformidade com as regras. No entanto, devido ao tamanho e a quantidade de fluxos de exceção dos softwares essa verificação é inviável de ser feita manualmente, por este motivo, e pela falta de ferramentas especificas voltadas ao tratamento de exceção, este trabalho propôs o desenvolvimento de uma API para realizar verificação de conformação, a API tem o objetivo de possibilitar a realização de buscas de divergência e ausência no código fonte. A verificação é feita por meio de testes de software, onde o framework de testes automatizados JUnit foi utilizado.

Palavras chave: Regras de Design. Conformação Arquitetural. Tratamento de Exceção.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1 INTRODUÇÃO

Os sistemas de softwares atuais são cada vez maiores em tamanho, número de usuários e complexidade estrutural (BARBOSA, 2012). Devido a essa natureza complexa, existe uma grande probabilidade da ocorrência de erros durante a sua execução. Essa possível ocorrência de erros pode causar prejuízos financeiros aos usuários do sistema e até mesmo para as organizações que os produzem, além de também colocar a vida de pessoas em risco. Junto ao crescimento gradual dos softwares, cresce também a necessidade de que eles sejam dignos de confiança, ou seja, eles devem prover serviços como esperado por seus usuários sob condições excepcionais de operação (BARBOSA, 2012). "No entanto, tais condições podem emergir durante o fluxo de execução de um sistema de software em decorrência à manifestação de falhas" (BARBOSA, 2012, p. 13). É necessário que essas condições excepcionais sejam tratadas de modo que o sistema seja capaz de identificar quando uma exceção ocorre e também, prover medidas de tratamento, a fim de minimizar os possíveis danos provenientes dessas condições excepcionais.

"Um componente chave de qualquer sistema de software confiável é o seu mecanismo de tratamento de exceção" (BRUNTINK; DEURSEN; TOURWÉ, 2006, p. 242). Esse mecanismo faz com que o sistema seja capaz de realizar a detecção de erros, e reagir a eles de forma apropriada, por exemplo, através da recuperação do erro ou sinalizando uma mensagem de erro especializada (BRUNTINK; DEURSEN; TOURWÉ, 2006). Os mecanismos de tratamento de exceção atualmente são os modelos mais comuns para lidar com a ocorrência de condições excepcionais em sistemas de software (BARBOSA, 2012). Tais mecanismos têm como objetivo melhorar os níveis de robustez e confiabilidade do software produzido provendo suporte automático e programático à detecção de condições excepcionais e à implementação de ações de recuperação (BARBOSA, 2012). Nas linguagens de programação o mecanismo de tratamento de exceção provê meios para estruturar as atividades de tolerância a faltas (*fault tolerance*) dentro do software (ROCHA, 2011).

Trabalhos recentes indicam que muitas aplicações industriais apresentam uma má qualidade do tratamento de exceção em seus projetos (SHAH; GORG; HARROLD, 2010). Na prática, muitos desenvolvedores acabam concentrando a maior parte dos seus esforços na implementação do comportamento normal do sistema (BARBOSA, 2012). Com isso, na maioria dos casos, toda a parte do desenvolvimento referente ao tratamento de exceção é

negligenciada. Também é prática comum entre os desenvolvedores postergar a implementação do tratamento de exceção apenas para as versões futuras do sistema (BARBOSA, 2012). Em alguns casos, de 40% a 72% das ações necessárias para a recuperação do estado interno em aplicações industriais são ações extremamente simples em que não há qualquer tentativa de recuperar o estado do sistema para um estado correto (BARBOSA, 2012). Segundo Barbosa (2012), essas ações tendem apenas a imprimir a pilha de execução associada à exceção, ou até mesmo não fazer nada.

Apesar de sua importância, vários estudos têm demonstrado que o tratamento de exceção é muitas vezes a menos bem compreendida, documentada e testada parte de um sistema de software (BRUNTINK; DEURSEN; TOURWÉ, 2006). Estudos realizados na área de tratamento de exceção apontam os desenvolvedores como os principais culpados pela baixa qualidade dos sistemas atuais com respeito ao tratamento de exceção. Shah, Gorg e Harrold (2010) conduziram, um estudo com desenvolvedores de software experientes e novatos com intuito, de entender o ponto de vista de cada um desses perfis com respeito ao tratamento de exceção. Esse estudo mostrou que para os desenvolvedores de software experientes o tratamento de exceção é uma parte crucial no processo de desenvolvimento. Em contraste, os desenvolvedores novatos tendem a negligenciar o tratamento de exceção. Estes últimos veem o tratamento de exceção como mais um recurso que lhes ajuda na depuração, ou seja, quando acontece uma exceção, eles usam a informação fornecida pelo rastreamento da pilha para entender o que causou a exceção. Por esse motivo, eles tendem a não investir tempo na implementação do código de tratamento de exceção, a menos que sua implementação contribua com a depuração (SHAH; GORG; HARROLD, 2010).

Outro estudo com intuito semelhante foi realizado em Shah, Gorg e Harrold (2008), o seu objetivo era responder a seguinte pergunta, "**Porque os Desenvolvedores Negligenciam o Tratamento de Exceções?**". Os resultados desse estudo mostraram que os desenvolvedores não estão satisfeitos com os mecanismos de tratamento de exceção existentes no Java, e também não gostam da maneira que linguagens de programação, como por exemplo Java, impõem a implementação do tratamento de exceção. Na literatura é possível encontrar vários problemas causados por uma baixa qualidade no tratamento de exceção. Por exemplo, Jiang et al (2005) afirmam que o veículo lançador Ariane 5 foi perdido devido a uma exceção não tratada, tal falha custou a ESA (Agência Espacial Européia) 10 anos de trabalho e 7 milhões de dólares, além da destruição de 500 milhões de dólares em carga científica.

Levando em consideração o crescimento da complexidade dos sistemas de software atuais, a alta probabilidade de ocorrência de erros e a negligência dos desenvolvedores em relação ao tratamento de exceção, podemos perceber a importância de se considerar o tratamento de exceção como parte integrante do processo de desenvolvimento, desde o projeto até a implementação. Uma solução para melhorar o tratamento de exceção das aplicações seria realizar uma verificação baseada em regras de design, ou seja restrições de projeto, aplicadas ao código fonte do software, assim seria possível descobrir se o sistema está ou não em conformidade com o que foi projetado. Nesse cenário, verificar a conformidade entre as regras de design do tratamento de exceção, e o código da aplicação pode, rapidamente, se tornar inviável se realizada manualmente – devido à complexidade e ao grande número de fluxos de exceção a serem seguidos (JÚNIOR; COELHO, 2011). Portanto, este trabalho tem como principal objetivo desenvolver uma API (Application Programming Interface) que possibilite a verificação automática de conformidade entre as regras de design do tratamento de exceção e o código fonte da aplicação. Para isso, três macro atividades foram desenvolvidas: (i) definição de um conjunto de regras de design para tratamento de exceção; (ii) desenvolvimento da API para verificação automática de conformidade entre as regras e o código fonte, chamada de DR4EH (Design Rules for Exception Handling); (iii) aplicação da API desenvolvida à um sistema exemplo com o intuito de validar o uso da API.

Nesse contexto, existem estudos semelhantes ao tema abordado neste trabalho que também utilizam ferramentas automáticas para realizar verificação de conformação, porém alguns, com objetivos e técnicas diferentes. Em Villela (2009), foi realizado um estudo com o intuito de prover uma solução para verificação de conformação baseada em análise estática. Seu objetivo era investigar uma solução para conformação arquitetural centrada na observação de que dependências inter-modulares impróprias são uma fonte importante de violações arquiteturais e, portanto, contribuem para o processo de erosão arquitetural (VILLELA, 2009). O trabalho realizado em Villela (2009) é o que mais se assemelha a este trabalho, pois o objetivo da API desenvolvida aqui, também inclui verificar dependência entre módulos, tal verificação será possível por meio de regras de design que envolvam dois módulos em uma determinada verificação de conformidade. Além disso, a sintaxe da linguagem DCL foi utilizada neste trabalho como inspiração para definição das regras de design para tratamento de exceção.

Já em Júnior e Coelho (2011) foi realizado um estudo com o intuito de avaliar a utilização de contratos de tratamento de exceção para especificar design rules de tratamento

de exceção de SPL (*Software Product Line*) e apresentar uma abordagem para verificar automaticamente as regras por meio de um conjunto de testes unitários usando a ferramenta JUnit gerados automaticamente, foi utilizada a programação orientada a aspectos para apoiar a verificação das design rules de exceção durante a execução dos testes automatizados. O trabalho desenvolvido em Júnior e Coelho (2011), é o que mais se assemelha a este trabalho, porém, aqui não foram definidos contratos de tratamento de exceção, além disso o foco das regras de design não são especificamente SPL. A idéia aqui é gerar uma API reuzável, que possa ser utilizada para testar qualquer código fonte escrito em Java. O framework de testes JUnit também foi utilizado para apoiar a API e automatizar a verificação das regras.

O principal argumento para reforçar a necessidade deste trabalho, é o fato de que segundo Barbosa (2012), há uma lacuna na literatura em termos de trabalhos que ajudam desenvolvedores a escrever código de tratamento de exceção com qualidade. Em especial, há poucos que discutem e propõem conjuntos de práticas e padrões de implementação de elementos de tratamento de exceção (BARBOSA, 2012). Contudo, este trabalho tem como principal público alvo desenvolvedores de software, que poderão fazer uso da API desenvolvida para verificar a conformação entre as regras de design e o código fonte do software.

2 REVISÃO BIBLIOGRÁFICA

Nessa seção serão apresentados os principais conceitos referentes a este trabalho, será feita uma breve introdução a fim de contextualizar o leitor sobre o tema do mesmo, logo em seguida será citado onde o dado conceito se relaciona com o trabalho em questão.

2.1 Regras de Design

Regra de design é um conceito simples de restrições de projeto, são restrições que devem ser seguidas durante o processo de desenvolvimento. Existem diferentes definições sobre o conceito de regras de design, a seguir serão citadas algumas dessas definições no contexto deste trabalho. De acordo com Júnior e Coelho (2011), regras de design definem padrões arquitetônicos que devem ser rigorosamente seguidas em todas as fases do ciclo de vida do software. Já segundo Morgan, Volder e Wohlstadter (2007), regras de design expressam restrições sobre o comportamento e a estrutura de um programa. Essas regras podem ajudar a garantir que o programa siga um conjunto das práticas estabelecidas e evita

certas classes de erros. Como já foi mencionado anteriormente, no contexto de desenvolvimento de software as regras de design impõem restrições ao desenvolvimento do sistema.

Regras de design definem maneiras como o sistema deve ou não ser construído e que práticas devem ser seguidas. O conceito de regras de design foi utilizado neste trabalho para definir restrições para o projeto e implementação do tratamento de exceção. As regras propostas formam implementadas via API a fim de torna-las reusáveis e passíveis de serem verificadas de forma automática. O objetivo do levantamento dessas regras de design é permitir ao engenheiro de software realizar verificações específicas no código fonte do software em busca de violações de decisões arquiteturais com respeito ao tratamento de exceção.

2.2 Conformação Arquitetural

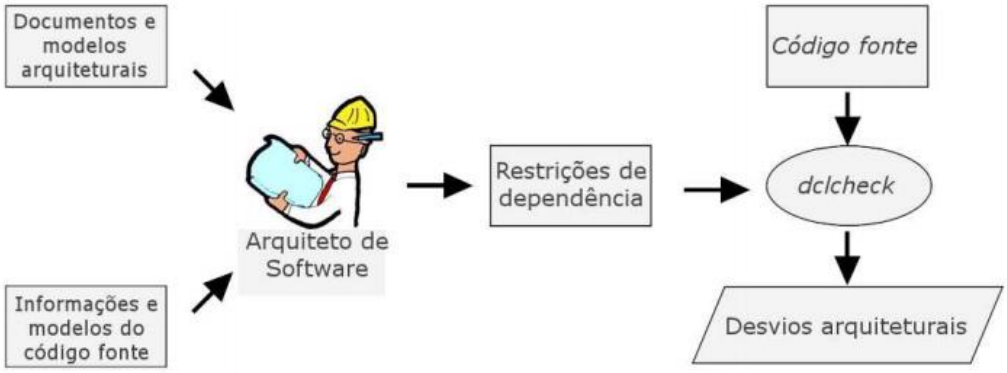
Define-se conformação arquitetural como a medida do grau de aderência da arquitetura implementada em código fonte de um sistema com a sua arquitetura planejada (VILLELA, 2009). A arquitetura de software é geralmente definida como um conjunto de decisões de projeto que tem impacto em cada aspecto da construção e evolução de sistemas de software (VILLELA, 2009). A definição de uma arquitetura engloba diversos padrões e boas práticas arquiteturais (TERRA; VALENTE, 2010). Isso inclui como os sistemas são estruturados em componentes e restrições sobre como tais componentes devem interagir. “Apesar de sua inquestionável importância, a arquitetura documentada de um sistema, quando disponível, geralmente não reflete a sua implementação atual” (VILLELA, 2009, p. 1). Apesar das pesquisas constantes na área de arquitetura de software e das diversas propostas de soluções no sentido de evitar o fenômeno de erosão arquitetural, que é uma indicação de que o software está se degenerando, ou seja está mudando em relação a sua arquitetura projetada, ainda é pouco comum que artefatos arquiteturais sejam mantidos em sincronia com os requisitos do sistema e sua implementação (VILLELA, 2009).

Contudo, com o decorrer do projeto os padrões e as boas práticas definidas na fase de planejamento tendem a se degradar fazendo com que os benefícios proporcionados pelo projeto da arquitetura (e.g., manutenibilidade, escalabilidade e; portabilidade) sejam anulados (TERRA; VALENTE, 2010). A degradação sofrida durante o processo de desenvolvimento, pode resultar na ocorrência de fenômenos como desvio e erosão arquitetural (VILLELA, 2009). Desvio arquitetural acontece quando uma decisão de projeto que não viola nenhuma

decisão principal presente na arquitetura planejada, é introduzida na arquitetura concreta. Já a Erosão arquitetural é um problema que afeta a maioria, se não todos, os grandes sistemas de software (GURP; BOSCH; BRINKKEMPER, 2003). Essencialmente, o problema é que, como o software evolui, ele é gradativamente alterado para atender a novos requisitos, defeitos de reparo ou atributos de qualidade (manutenção adaptativa, corretiva e perfectiva) (GURP; BOSCH; BRINKKEMPER, 2003). No entanto, estes requisitos podem entrar em conflito com os requisitos em iterações anteriores ou podem alterar os pressupostos em que decisões de projeto em iterações anteriores foram feitas (GURP; BOSCH; BRINKKEMPER, 2003).

A proposta para a conformação de arquitetura depende de técnicas de análise estática para detectar dependências estruturais que são indicadores de erosão arquitetural (TERRA; VALENTE, 2009). Conforme ilustrado na Figura 1, inicialmente o arquiteto de software deve definir as restrições de dependência válidas no sistema, utilizando para isso uma linguagem de restrição de dependências (TERRA; VALENTE, 2008). Para definição dessas restrições, o arquiteto deve se basear no modelo arquitetural (isto é, um modelo que represente a arquitetura esperada ou planejada de um sistema) e um modelo de código fonte, o qual pode incluir, por exemplo, diagramas de classe, de pacotes e de interação (TERRA; VALENTE 2008). Por fim, deve ser utilizada uma ferramenta que detecte pontos do código que violam as restrições de dependência especificadas (TERRA; VALENTE, 2008).

Figura 1 A abordagem proposta para conformação arquitetural



Fonte: Villela (2009).

Existem ferramentas que podem ser utilizadas para verificar se a implementação de um sistema atende à sua arquitetura planejada (VILLELA, 2009). Tais ferramentas

proveem meios para detectar desvios em relação à arquitetura planejada de um sistema e, portanto, são recursos importantes para prevenir erosão arquitetural (VILLELA, 2009). Como exemplo algumas ferramentas desse tipo são citadas a seguir:

- SemmleCode .QL (*Query Language*), uma linguagem de consulta em código fonte inspirada na sintaxe da linguagem SQL (VILLELA, 2009). .QL suporta várias tarefas de analise em código fonte, como, por exemplo, procura por erros (VILLELA, 2009).
- SAVE (*Software Architecture Visualization and Evaluation*), uma ferramenta para avaliação estática de arquiteturas de software (VILLELA, 2009). Baseada nos princípios de modelos de reflexão (*reflexion models*), SAVE compara a arquitetura planejada de um sistema com a sua arquitetura implementada, ou seja o código fonte (VILLELA, 2009). Como resultado dessa comparação, a ferramenta destaca relações convergentes, divergentes e ausentes entre os dois modelos (VILLELA, 2009).
- LDM (*Lattix Dependency Manager*), é uma ferramenta para conformação e gerenciamento arquitetural baseada no conceito de matrizes de dependência estrutural (*Dependency Structure Matrixes* ou DSM) (VILLELA, 2009). DSM são matrizes de adjacência utilizadas para representar dependências entre módulos de um sistema (VILLELA, 2009). A LDM basicamente estrai as DSMs do código fonte e apresenta claramente em forma de uma matriz quadrática o padrão arquitetural subjacente a aplicação verificada. LDM também suporta o conceito de (*design rules*) (VILLELA, 2009).

A princípio, soluções para verificação de conformação arquitetural podem ser classificadas em duas principais linhas de atuação (VILLELA, 2009): as que se baseiam em técnicas de análise estática, que será utilizada para realização deste trabalho, e as que se baseiam em técnicas de análise dinâmica. Seguindo a técnica de análise estática, Villela (2009) propôs uma solução para conformação arquitetural centrada em análise de dependências inter-modulares. Para isso, foi projetada uma linguagem que permite definir dependências aceitáveis e não aceitáveis de acordo com a arquitetura planejada, a linguagem de restrição de dependência (DCL), e também foi implementado um protótipo de ferramenta, chamada *DCLChecker*, que verifica se o código respeita restrições de dependências em DCL

(VILLELA, 2009). Assim como realizado em Villela (2009), este trabalho propõe o uso de ferramentas automáticas para verificação de conformação, porém, o foco aqui não é exclusivamente restrição de dependência entre módulos, e sim regras de design para tratamento de exceção.

"Tais restrições de projeto geralmente são definidas e verificadas manualmente. No entanto, em sistemas de grande porte, torna-se inviável a verificação manual dessas regras" (JÚNIOR; COELHO, 2011, p. 9). Essa inviabilidade de verificação motivou a realização deste estudo, assim uma API para verificação automática foi desenvolvida para realizar a verificação de conformidade entre as regras e o código fonte.

2.3 Tratamento de Exceção

O tratamento de exceção é considerado por muitos desenvolvedores como uma tarefa central no processo de desenvolvimento de software (BARBOSA, 2012). Os mecanismos de tratamento de exceção são modelos usados para estruturar o fluxo excepcional de um módulo de software através da detecção e sinalização da ocorrência de exceções (BARBOSA, 2012). Existem mecanismos de tratamento de exceção para diversas linguagens de programação, porém, este trabalho aborda apenas o tratamento de exceção em programas escritos na linguagem Java.

Exceções modelam comportamentos anormais de um sistema, que podem ser divididas em dois tipos básicos: (i) exceções internas, aquelas que são tratadas pelos tratadores associados com a região protegida onde a exceção foi lançada; e (ii) as exceções externas, que são aquelas capturadas por tratadores associadas com outras regiões protegidas (EBERT, 2013). Segundo Barbosa (2012), tratador de exceção é um conjunto de ações tomadas por um módulo de software a fim de reagir a uma exceção. Já região protegida, é um trecho de código que possui associado a si um ou vários tratadores de exceções (BARBOSA, 2012).

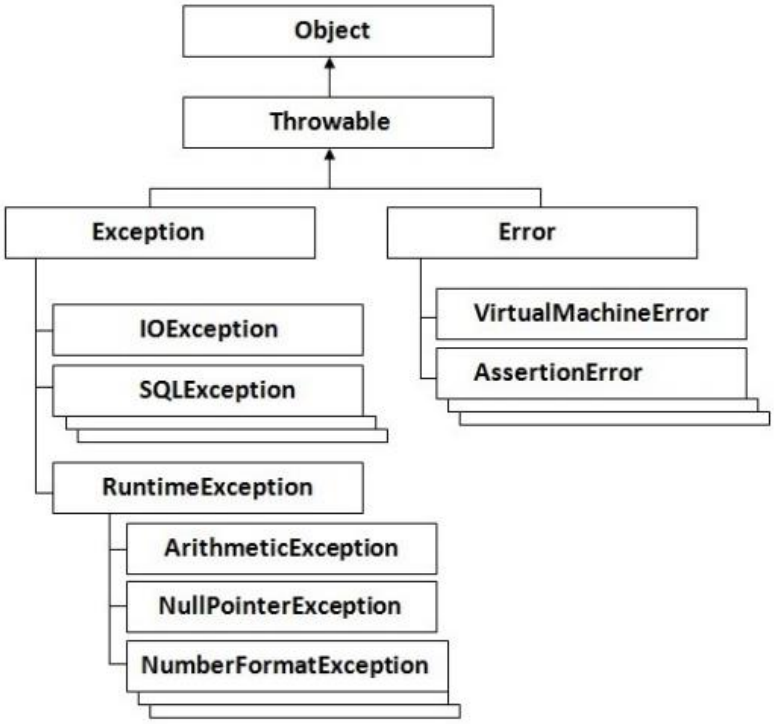
Existem na literatura conceitos que são importantes no entendimento do tratamento de exceção, a seguir alguns desses conceitos serão abordados:

- **Estado interno**: "é o conjunto de condições no qual um sistema de software se encontra em um determinado instante no tempo durante sua execução" (BARBOSA, 2012, p. 25).

- **Defeito**: “é um evento que altera o estado interno de um sistema de software para um estado em que o serviço não corresponde ao esperado, ou desejado, conforme o definido em alguma especificação” (BARBOSA, 2012, p. 26).
- **Erro**: “é um estado interno de um sistema de software que possibilita a ocorrência de um defeito” (BARBOSA, 2012, p. 26).
- **Falha**: “é uma causa hipotética de erro. Falhas podem ser imperfeições ou irregularidades que ocorrem em módulos de hardware ou software” (BARBOSA, 2012, p. 26).
- **Falta**: “é a causa física ou algorítmica de um erro” (ROCHA, 2014, p. 4).

“O tratamento de exceção em programas Java não faz qualquer distinção entre exceções internas e externas” (EBERT, 2013, p. 7), que são tratadas da mesma maneira. Regiões protegidas são definidas por blocos *try*. Os tratadores são definidos por blocos *catch* e as ações de limpeza são definas por blocos *finally* (EBERT, 2013). A instrução *throw* é usada para sinalizar uma exceção. Segundo Ebert (2013, p. 6), “ações de limpeza são as partes do código que executam ações para manter o estado do programa consistente independentemente da exceção sendo lançada”. Ações de limpeza são sempre iniciadas após a execução da região protegida, mesmo que uma exceção tenha sido lançada ou não. O Java é uma linguagem orientada a objetos, assim as exceções são objetos de classes que são subclasses de *Exception* (EBERT, 2013). Desse modo, os desenvolvedores podem criar novos tipos de exceção como subclasses de outras exceções (EBERT, 2013). Exceções são estruturadas de forma hierárquica como podemos ver na Figura 2.

Figura 2 Tratamento de exceção em Java



Fonte: Ebert (2013).

"O mecanismo de tratamento de exceção é uma das principais técnicas utilizadas para apoiar o desenvolvimento de sistemas confiáveis" (JÚNIOR; COELHO, 2011, p. 9). Ele visa melhorar a modularidade e robustez dos programas de duas maneiras (EBERT, 2013): (i) a separação explícita de código que lida com o comportamento excepcional do código normal; e (ii) na declaração de interfaces excepcionais. Os Mecanismos de tratamento também proveem suporte a detecção da ocorrência de condições excepcionais e à implementação de ações de recuperação (BARBOSA, 2012). "Um dos objetivos dos mecanismos de tratamento de exceção é permitir que desenvolvedores de software implementem sistemas mais robustos e tolerantes a falhas" (BARBOSA, 2012, p. 14). Para tanto é necessário que os desenvolvedores estruturem ações de recuperação adequadas. "Consideram-se aqui ações de recuperação adequadas aquelas capazes de restaurar um sistema de software para um estado correto após a ocorrência de uma exceção" (BARBOSA, 2012, p. 14).

O mecanismo de tratamento de exceção permite aos desenvolvedores definirem exceções e estruturarem o comportamento da atividade excepcional dos componentes através dos **tratadores de exceção** ou, simplesmente, **tratadores** (ROCHA, 2011). Dessa forma, quando uma exceção é lançada durante a atividade normal do sistema (ou componente), o mecanismo de tratamento de exceções desvia o **fluxo de controle normal** para o **fluxo de**

controle excepcional (ROCHA, 2011). Segundo Barbosa (2012), fluxo de controle excepcional é o fluxo realizado por um módulo a fim de recuperar o seu estado interno para um estado correto, e fluxo de controle normal é o fluxo onde o componente processa as requisições de serviço de acordo com a sua especificação.

No contexto de linguagens de programação, as exceções são tipicamente divididas em exceções pré-definidas e exceções definidas pelo usuário (BARBOSA, 2012). Exceções pré-definidas são declaradas implicitamente e estão relacionadas a condições excepcionais detectadas pelo mecanismo de suporte à execução da linguagem, ou ainda por hardwares e sistemas operacionais subjacentes (BARBOSA, 2012). Exceções definidas pelo usuário são exceções definidas e detectadas pelo usuário em exceções de aplicação e exceções de APIs (ou exceções de terceiros) (BARBOSA, 2012). No contexto de linguagens de programação, um mecanismo de tratamento de exceção define (BARBOSA, 2012):

- Como representar uma exceção;
- Como especificar regiões protegidas da ocorrência de exceção;
- Como definir um tratador de exceções;
- Como associar uma determinada exceção a um tratador;
- Como sinalizar a ocorrência de uma exceção; e
- Como declarar na interface de um modulo quais exceções ele pode sinalizar.

Umas das vantagens dos mecanismos de tratamentos de exceção é prover uma melhor separação entre a implementação da lógica relacionada ao provimento do serviço oferecido pelo sistema e a lógica responsável por recuperar o sistema para o estado correto, caso uma exceção ocorra durante a sua execução (BARBOSA, 2012).

O conceito de tratamento de exceção é abordado neste trabalho como um ponto importante e crucial do desenvolvimento de software devido aos benefícios que ele proporciona e aos problemas que um mecanismo ruim pode causar. Sendo assim, uma das frentes motivadoras deste trabalho é o fato de existir uma grande necessidade de garantir maior qualidade ao código de tratamento de exceção, a fim de tornar os sistemas de software mais confiáveis e robustos. Como já foi dito, os desenvolvedores de software têm por hábito negligenciar o tratamento de exceções, ocasionando assim, baixa qualidade no código dos

produtos de software. Desse modo, este trabalho propôs o desenvolvimento de uma API de verificação automática de conformidade do tratamento de exceção.

3 PROCEDIMENTOS METODOLÓGICOS

A capacidade de capturar, tratar e se recuperar de falhas é um ponto fundamental a ser atingido para quem deseja construir sistemas de software de qualidade. Um meio para se conseguir isso é através do mecanismo de tratamento de exceção, que permite que um sistema seja capaz de perceber e tratar uma condição excepcional de execução, fazendo com que o sistema saia de um estado de execução excepcional e retorne para um estado de execução normal. No entanto, não é isso que acontece nos sistemas de softwares desenvolvidos hoje, os desenvolvedores de software tendem a negligenciar o tratamento de exceção, e ignorar as decisões arquiteturais tomadas na fase de projeto em relação a este aspecto do desenvolvimento, fazendo assim com que os softwares possuam baixa qualidade de tratamento de exceção. Com base nessa afirmação, este trabalho propõe uma API Java que fornece suporte à verificação automática de conformidade em regras de design do tratamento de exceção e o código fonte do software, permitindo verificar o levantamento, tratamento, sinalização e propagação de exceções.

3.1 Definir regras de design para o tratamento de exceção

Nesta etapa inicial foram definidas as regras de design. Para isso, uma análise foi realizada para avaliar que tipos de regras seriam necessárias para abranger por completo o software e as necessidades do designer que vai verificar a sua aplicação. As regras definidas oferecem ao designer algumas características bem específicas que podem ser verificadas no código da aplicação, como, por exemplo, saber se uma classe ou pacote, captura um tipo de exceção que não deveria ser capturada. A definição das regras acabou sendo a primeira atividade realizada, pois decidiu-se, que uma vez definidas as regras, estas seriam usadas na avaliação das ferramentas existentes. As regras definidas nesta etapa serão descritas no tópico 4.

3.2 Analisar as ferramentas existentes

Nesta etapa foi realizada uma análise das ferramentas existentes na literatura que são utilizadas para verificação de conformação arquitetural. Para realizar a análise das

ferramentas levantadas, foram usados como critérios as regras de design definidas na fase anterior. De modo de que deve ficar explícito se a ferramenta possui suporte as regras ou não.

3.3 Implementar as regras de design

Nesta etapa foram implementadas as regras de design definidas inicialmente. Para a realização desta etapa foi necessário um estudo mais detalhado da ferramenta escolhida a fim de implementar as regras de design levantadas anteriormente.

3.4 Validar a ferramenta desenvolvida

O objetivo desta etapa é demonstrar o uso da API proposta fazendo o uso das regras de design na verificação de conformação e uma aplicação exemplo. Para isso foi desenvolvida uma aplicação Java chamada de IContatos. Para apoiar e automatizar a verificação, o framework JUnit foi então utilizado.

4 REGRAS DE DESIGN

4.1 Definição das regras

As regras de design para tratamento de exceção foram definidas com base na sintaxe usada nas restrições de dependência citadas em Villela (2009). Elas foram divididas em dois tipos de regras, as **regras de dependência** e as **regras de propagação**. As regras de dependência são formadas por três tipos, ou categorias, de restrições descritas a seguir:

- *Restrições de Dependência de Levantamento (RDL):* estão relacionadas com o levantamento de uma determinada exceção dentro de um módulo específico. Quando uma exceção *E* é levantada dentro de um módulo *M*, dizemos que *M* levantou *E*.
- *Restrições de Dependência de Sinalização (RDS):* estão relacionadas com a sinalização de uma determinada exceção para fora de um módulo específico. Quando uma exceção *E* é sinalizada por um módulo *M*, dizemos que *M* sinalizou *E*.
- *Restrições de Dependência de Tratamento (RDT):* estão relacionadas com o tratamento de exceção por parte de um módulo específico. Quando uma exceção *E* é tratada por um módulo *M*, dizemos que *M* trata *E*.

Já as regras de propagação são formadas por um único tipo de restrição, que é a restrição de propagação, descrita a seguir:

- *Restrições de Fluxo de Propagação (RFP):* estão relacionadas com a propagação de uma exceção entre dois módulos. Quando uma exceção *E* é sinalizada do módulo *M* para o módulo *N*, dizemos que *E* é propagada de *M* para *N*.

Além disso, as categorias de regras de design podem ser divididas em regras de divergência e ausência. As regras de divergência que ocorrem quando uma relação, que envolve um ou dois módulos e exceção, não prescrita na arquitetura, existe no código fonte. Já as regras de ausência ocorrem quando uma relação, que envolve um ou dois módulos e uma exceção, prescrita na arquitetura, não existe no código fonte.

4.2 Sintaxe e Semântica

Cada regra de design possui uma sintaxe clara e bem definida o que torna explicito o seu significado. Este trabalho foi pensado para ser reutilizável, para isso existe a necessidade de que as regras sejam simples, diretas e fáceis de serem compreendidas. Portanto, baseado na importância de construir regras claras, e no trabalho de Villela (2009), foi criada então uma sintaxe para a documentação das regras. A seguir será dada a definição da sintaxe e da semântica das regras de design definidas neste trabalho.

4.2.1 Regras de Dependência

As regras de design que expressam restrições de dependência, são divididas em regras de levantamento, sinalização e tratamento. Cada uma das regras de dependência é ainda dividida em mais dois subtipos, regras de divergência e regras de ausência.

Restrições de Dependência de Levantamento (RDL):

- Regras de Divergência:
  - *only M can-raise E*: somente os métodos das classes do módulo *M* podem levantar exceções do tipo *E*.
  - *M can-only-raise E*: os métodos das classes do módulo *M* podem, somente, levantar exceções do tipo *E*.
  - *M cannot-raise E*: os métodos das classes do módulo *M* não podem levantar exceções do tipo *E*.

- Regras de Ausência
    - *M must-raise E*: todos os métodos das classes do módulo *M* devem levantar exceções do tipo *E*.

Restrições de Dependência de Sinalização (RDS):

- Regras de Divergência:
    - *only M can-signal E*: somente os métodos das classes do módulo *M* podem sinalizar exceções do tipo *E*.
    - *M can-only-signal E*: os métodos das classes do módulo *M* podem somente, sinalizar exceções do tipo *E*.
    - *M cannot-signal E*: os métodos das classes do módulo *M* não podem sinalizar exceções do tipo *E*.
- Regras de Ausência:
    - *M must-signal E*: todos os métodos das classes do módulo *M* devem sinalizar exceções do tipo *E*.

Restrições de Dependência de Tratamento (RDT):

- Regras de Divergência:
    - *only M can-handle E*: somente os métodos das classes do módulo *M* podem tratar exceções do tipo *E*.
    - *M can-only-handle E*: os métodos das classes do módulo *M* podem, somente tratar exceções do tipo *E*.
    - *M cannot-handle E*: os métodos das classes do módulo *M* não podem tratar exceções do tipo *E*.
- Regras de Ausência:
    - *M must-handle E*: todos os métodos das classes do módulo *M* devem tratar exceções do tipo *E*.

4.2.2 Regras de Propagação

As regras de design de propagação expressam relacionamentos envolvendo dois módulos e um determinado tipo de exceção.

Restrições de Fluxo de Propagação (RFP):

- Regras de Divergência:

- *only M can-signal E to N*: somente os métodos das classes do módulo *M* podem sinalizar exceções do tipo *E* para métodos das classes do módulo *N*.
- *M can-only-signal E to N*: os métodos das classes do módulo *M* podem, somente, sinalizar exceções do tipo *E* para métodos das classes do módulo *N*.
- *M cannot-signal E to N*: os métodos das classes do módulo *M* não podem sinalizar exceções do tipo *E* para métodos das classes do módulo *N*.

- Regras de Ausência:
  - *M must-signal E to N*: todos os métodos das classes do módulo *M* devem sinalizar exceções do tipo *E* para métodos das classes do módulo *N*.

5 AVALIAÇÃO DAS FERRAMENTAS EXISTENTES

Para o desenvolvimento deste trabalho foi realizado uma revisão bibliográfica, com o intuito de levantar as ferramentas existentes que poderiam ser usadas para verificar a conformidade entre regras de design e código fonte. Contudo, foram elencadas cinco ferramentas que foram analisadas com o objetivo de saber se elas são viáveis de serem utilizadas na verificação.

5.1 Critérios de avaliação

Para que ocorra a avaliação de uma determinada ferramenta é necessário que existam critérios para guiar o avaliador durante o processo. Portanto, foi definido que os critérios de avaliação seriam os tipos de regras de design que as ferramentas levantadas são capazes de verificar.

Já pensando em posteriormente avaliar o produto desenvolvido, foi decidido que as regras de design definidas neste trabalho deveriam ser passiveis de verificação por meio de outras ferramentas. Assim, elas serão os critérios de avaliação das ferramentas elencadas. Desse modo, as ferramentas devem possuir suporte à verificação de regras das seguintes categorias.

- RDL: restrições de dependência de levantamento. As ferramentas devem ser capazes de realizar verificações que envolvam a busca por violações de levantamento de exceções.

- RDS: restrições de dependência de sinalização. As ferramentas devem ser capazes de realizar verificações que envolvam a busca por violações de sinalização de exceções.
- RDT: restrições de dependência de tratamento. As ferramentas devem ser capazes de realizar verificações que envolvam a busca por violações de tratamento de exceções.
- RFP: restrições de fluxo de propagação. As ferramentas devem ser capazes de realizar verificações que envolvam a busca por violações na propagação de exceções entre os módulos.

Foi decidido que cada critério de avaliação representaria uma categoria das regras de design definidas neste trabalho. Vale ressaltar que o foco deste trabalho não está na avaliação das ferramentas de verificação levantadas, e sim no desenvolvimento de uma ferramenta que possibilite tal verificação, possivelmente estendendo uma ferramenta existente. O único objetivo desta avaliação é expor o foco de cada ferramenta, seu uso, e seus pontos negativos em relação ao contexto de exceções, assim como também o contraste entre elas e o produto deste trabalho.

Sendo assim, um ponto importante para demostrar a importância deste trabalho é confrontar as regras de design identificadas, e as ferramentas levantas expondo suas limitações na verificação do tratamento de exceção.

5.2 Análise das ferramentas

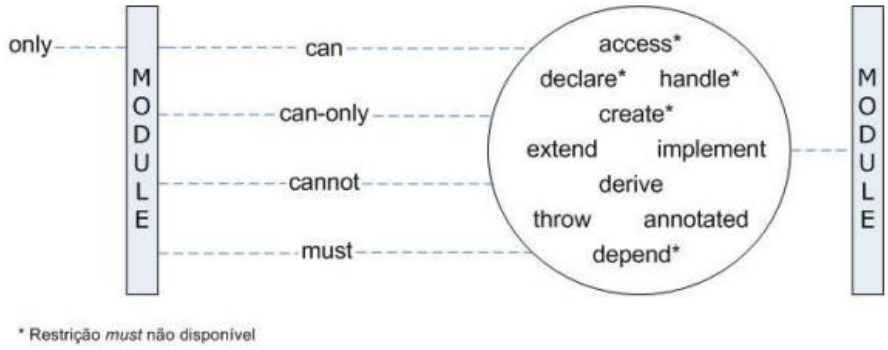
A seguir uma análise sobre cada ferramenta e suas principais características.

5.2.1 DCL - DCLChecker

DCL é uma linguagem de domínio específico, declarativa e estaticamente verificável que permite a definição de restrições de dependência entre módulos (TERRA; VALENTE, 2010). Assim o objetivo principal da linguagem é restringir a organização modular de um sistema de software e não o seu comportamento (TERRA; VALENTE, 2010). A DCL utiliza um modelo de granularidade fina para especificação de dependências estruturais comuns em sistemas orientados a objetos (VILLELA, 2009). Esse modelo permite a definição de dependências originadas a partir do acesso a atributos e métodos, declaração de variáveis, criação de objetos, extensão de classes, implementação de interfaces, ativação de exceções e uso de anotações (VILLELA, 2009).

Como as atuais linguagens orientadas a objetos permitem a módulos clientes referenciar quaisquer tipos públicos de outros módulos, a lógica por trás do projeto da linguagem DCL consiste em prover meios para controlar tais dependências (VILLELA, 2009). Basicamente, para capturar divergências, a DCL permite aos arquitetos de software especificar que certas dependências podem (***only can*** *e* ***can-only***) ou não podem (***cannot***) ser estabelecidas por módulos específicos (VILLELA, 2009). Além disso, para capturar ausências, DCL permite especificar que certas dependências devem estar presentes (***must***) em determinados módulos do sistema (VILLELA, 2009). A seguir na Figura 3 a sintaxe para restrições de dependência em DCL.

Figura 3 Sintaxe para declaração de restrições dependências em DCL



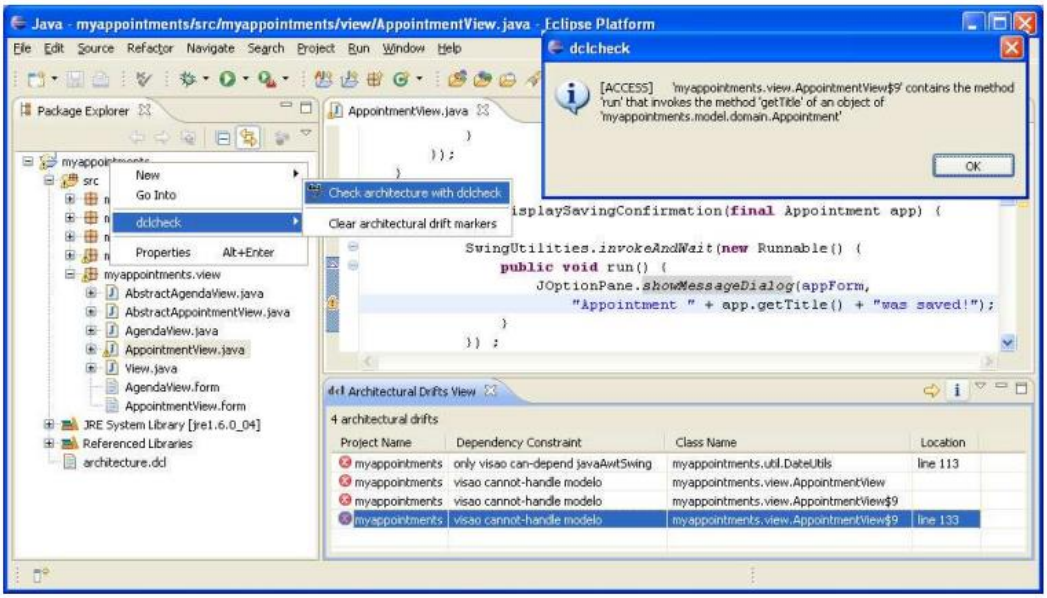
Fonte: Villela (2009).

A verificação de conformação é realizada por meio por meio de uma análise estática e automática com o auxílio da ferramenta dclchecker. A ferramenta é responsável por verificar se o código respeita as restrições de dependência em DCL (TERRA; VALENTE, 2010). A dclchecker utiliza um *framework* de análise e manipulação de *bytecode*, chamado de ASM, para extrair, a partir dos *bytecodes*, todas as dependências entre os módulos de um sistema (VILLELA, 2009). Segundo Ow2 (2013), ASM é uma estrutura de manipulação e análise de *bytecode* Java. Ele pode ser usado para modificar as classes existentes ou dinamicamente gerar classes diretamente na forma binária (OW2, 2013). Basicamente a implementação da ferramenta dclchecker possui quatro módulos principais.

- *Dependency Map Builder*: Este módulo é responsável por extrair todas as expectativas existentes entre os módulos de um sistema. Além do mais, ele é o único modulo que utiliza o *framework* ASM (VILLELA, 2009). Basicamente o mapa de dependências construído é uma estrutura de dados que associa cada classe A de um sistema a uma lista de classes dependentes B1, B2, ..., Bn (VILLELA, 2009).

- *Parser*: Este módulo efetua a leitura do conjunto de restrições de dependências em DCL e constrói, para cada restrição, uma estrutura de dados que contém seu quantificador, seu tipo, seus módulos de origem e destino (VILLELA, 2009).
- *Validator*: este módulo é responsável por verificar se o sistema respeita cada restrição de dependência analisada (VILLELA, 2009).
- *Output*: este módulo apresenta as violações detectadas pelo módulo anterior em uma visão da IDE Eclipse, como pode ser visto na figura 4 (VILLELA, 2009).

Figura 4 Output no dclchecker



Fonte: Villela (2009).

5.2.2 LDM

LDM é uma ferramenta de conformação e visualização arquitetural que utiliza DSMs para representar e gerenciar dependências inter-classes em sistemas orientados a objetos (VILLELA, 2009). Segundo Villela (2009), DSM é uma matriz quadrada cujas linhas e colunas são classes de um sistema orientado a objetos. Conforme ilustrado na Tabela 1, um x na linha referente à classe **A** e na coluna referente à classe **B** denota que a classe **B** depende da classe **A**, isto é, existem referências explícitas em **B** para elementos sintáticos de **A**. Uma

outra possibilidade é representar na célula (**A,B**) o número de referências que **B** contém para **A** (VILLELA, 2009).

Tabela 1 – Exemplo de DSM

		1	*2*	*3*
CLASS A	**1**	**.**	**X**	
CLASS B	**2**		**.**	
CLASS C	**3**			**.**

Fonte: Villela (2009).

O principal objetivo da ferramenta LDM é oferecer aos arquitetos uma ferramenta gráfica que permita revelar padrões arquiteturais e detectar dependências que possa indicar violações arquiteturais ou mesmo um projeto de software deficiente (VILLELA, 2009). Para verificar a conformidade entre o código fonte e as restrições de dependência definidas, LDM automaticamente extrai a DSM do código utilizando técnicas de análise estática (VILLELA, 2009). A LDM implementa um algoritmo de reordenação (ou particionamento). Basicamente, esse algoritmo decide a ordem de apresentação das linhas em uma DSM, iniciando pelos pacotes que são mais utilizados pelos outros pacotes (VILLELA, 2009). As regras de projeto em LDM possuem duas formas: **A can-use B** e **A cannot-use B**, indicando que classes do conjunto **A** podem (ou não podem) depender de classes do conjunto **B** (VILLELA, 2009).

5.2.3 .QL – SemmleCode .QL

.QL é uma linguagem de consulta em código fonte que provê suporte a uma ampla gama de tarefas de desenvolvimento de software, tais como verificação de convenções de código, procura por erros, cálculo de métricas de engenharia de software, detecção de oportunidades de refatoração, etc (VILLELA, 2009).

A linguagem .QL é inspirada na linguagem SQL, o que torna sua sintaxe familiar para a maioria dos desenvolvedores (VILLELA, 2009). Existem ainda várias outras características que aumentam o seu poder de expressão para a consulta em código fonte (VILLELA, 2009). Por exemplo, a implementação interna da linguagem utiliza um *engine* Datalog, uma linguagem lógica mais restrita que o Prolog, para definir consultas recursivas ao longo de uma hierarquia de herança ou de um grafo de chamadas de métodos em sistemas orientados a objetos (VILLELA, 2009). Além disso, .QL permite o uso de conceitos de

orientação a objetos, tais como classes e herança, podem ser usados para estender a linguagem com novos predicados e construir bibliotecas de consultas (VILLELA, 2009).

Figura 5 Exemplo de consulta .QL

```
1: from RefType r1, RefType r2
2: where
3:   r1.fromSource() and depends(r1,r2) and
4:   (r2.fromSource() or isSwingApi(r2) or isSqlApi(r2))
5: select r1.getPackage(), r2.getPackage()
```

Fonte: Villela (2009).

Para realizar a verificação do código fonte é utilizada a ferramenta SemmleCode .QL. SemmleCode .QL é um plugin para a IDE Eclipse que permite a execução de consultas .QL sobre sistemas em Java (VILLELA, 2009).

A ferramenta inclui um editor de consultas em códigos Java cuja apresentação de seus resultados pode ser vista em forma de árvores, tabelas, gráficos, grafos e warnings reportados pelo ambiente de desenvolvimento (VILLELA, 2009). As consultas são expressas na linguagem .QL, e segundo Paz (2010), têm como principais objetivos:

- Detectar erros originários de programação ambígua ou práticas inadequadas de programação.
- Codificar regras de programação para garantir o cumprimento de bons estilos de codificação.
- Obter vários tipos de métricas.

5.2.4 SAVE

É uma ferramenta de conformação arquitetural centrada no conceito de modelo de reflexão de software (VILLELA, 2009). SAVE inclui um editor gráfico que permite a arquitetos construir modelos de alto nível. Isso é uma característica diferencial de ferramentas baseadas em modelos de reflexão, uma vez que elas explicitamente delegam aos arquitetos a construção do modelo idealizado de seus sistemas, ao invés de automaticamente inferir tal modelo a partir do código fonte (VILLELA, 2009).

O arquiteto deve definir um modelo de alto nível que represente a arquitetura planejada ou desejada de um sistema. Além disso os arquitetos também devem definir um

mapeamento entre o modelo de código fonte (isto é, a arquitetura implementada do sistema) e o modelo de alto nível proposto (VILLELA, 2009). Ao construir um modelo de alto nível, SAVE requer que os arquitetos definam as dependências aceitáveis entre os componentes arquiteturais propostos. Para esse propósito, eles devem incluir uma seta do componente de origem para o de destino (VILLELA, 2009).

Assim a ferramenta compara os dois modelos, o de alto nível e o modelo de código fonte e classifica as relações entre os componentes como (VILLELA, 2009):

- Convergente: quando uma relação prescrita no modelo de alto nível é seguida pelo código fonte;
- Divergente: quando uma relação não prescrita no modelo de alo nível existe no código fonte;
- Ausente: quando uma relação prescrita no modelo de alto nível não existe no código fonte.

5.2.5 DESIGN WIZARD

Design Wizard (DW) é uma biblioteca que fornece uma poderosa API Java que oferece serviços para adquirir informações sobre a estrutura do código fonte (PIRES; BRUNET; RAMALHO, 2008). A Design Wizard dá suporte ao designer na tarefa de compor testes de design sem acrescentar qualquer esforço para aprender uma nova linguagem e especificar o projeto esperado (PIRES; BRUNET; RAMALHO, 2008).

Design Wizard utiliza testes de design para realizar as verificações em código fonte. Segundo Pires, Brunet e Ramalho (2008), Teste de Design é um tipo de teste unitário que deve ser escrito considerando informações estruturais sobre as entidades do código e seus relacionamentos. O principal objetivo da construção de testes de design é verificar a conformidade do código usando testes de software, garantindo a qualidade na evolução do software e evitando erosão arquitetural (PIRES; BRUNET; RAMALHO, 2008). Um exemplo simples de teste de design é o teste que especifica o comportamento desejado para a comunicação entre as classes. Sejam **A** e **B** classes de um determinado projeto. Suponha que o arquiteto do projeto não quer permitir a classe **A** usar a classe **B**, ou seja, não deve existir qualquer chamada de método entre elas (PIRES; BRUNET; RAMALHO, 2008).

Os testes de design usam a API da Design Wizard para coletar informações sobre a estrutura. A fim de alcançar a análise estática do código, a Design Wizard lê as classes de um

código Java usando a estrutura de manipulação de *bytecode* ASM. Com isso, as informações sobre as estruturas do código fonte são extraídas e modeladas em classes, métodos, campos e suas relações (PIRES; BRUNET; RAMALHO, 2008). O uso de Design Wizard é muito simples, de forma que é apenas necessário que o designwizard.jar seja adicionado ao *classpath*, após isso o próximo passo é compor os testes e executá-los. Por esta razão, a ferramenta é totalmente independente de qualquer ambiente de desenvolvimento integrado (IDE) ou plataforma (PIRES; BRUNET; RAMALHO, 2008).

O processo para verificação da arquitetura acontece da seguinte forma, o arquiteto compõe o teste com o apoio da API de Design Wizard. Enquanto isso os desenvolvedores codificam o sistema (PIRES; BRUNET; RAMALHO, 2008). Em seguida, os testes são executados para verificar se o código escrito pelos desenvolvedores está em conformidade com o teste de design. Para isso, o framework JUnit tem sido utilizado para executar os testes e reportar falhas (PIRES; BRUNET; RAMALHO, 2008). A saída da verificação é a mesma saída do JUnit, ou seja, as falhas são relatadas com barra vermelha, enquanto o sucesso é relatado com barra verde (PIRES; BRUNET; RAMALHO, 2008).

Assim como em Pires, Brunet e Ramalho (2008), a verificação das regras deste trabalho será realizada por meio do framework de testes automatizados JUnit, e a saída do teste serão a barra de cor verde ou vermelha.

No desenvolvimento da API, apenas alguns componentes da Design Wizard foram utilizados neste trabalho, a seguir o nome de cada componente e a sua respectiva descrição.

- ClassNode: Pertence à classe Design. São objetos que a Design Wizard constrói automaticamente quando as classes são carregadas.
- MethodNode: Pertence à classe Design. Fornece informações sobre um único método ou um construtor em uma classe ou interface.
- getCatchedExceptions (): É um método da classe MethodNode. Retorna um *Set* de objetos ClassNodes que representam os tipos de exceções capturadas pelo método subjacente representado por este objeto methodnode.
- getThrownExceptions ()*:* É um método da classe MethodNode. É um método da classe MethodNode. Retorna um *Set* de objetos ClassNodes que representam os tipos de exceções declaradas a ser lançada pelo método subjacente representado por este objeto methodnode.

- getClass (String): Retorna uma entidade Classe associada ao nome especificado.
- getClass (Classe): Retorna um objeto ClassNode associado com a classe ou interface do objeto passado.
- getAllClasses (): Retorna um Set contendo objetos ClassNodes.
- getAllMethods (): Retorna um Set contendo objetos MethodNode.
- getCalleeMethods (): Retorna um Set de objetos MethodNode que são chamados por uma determinada entidade.
- getClassNode (): Retorna o ClassNode que representa determinado objeto.

5.3 Resultados da avaliação

Na sessão anterior foi realizada a análise de cada ferramenta levantada, ressaltando suas principais características e objetivos e pontos positivos e negativos relacionados ao contexto de exceção. Para guiar a avaliação foram utilizados como critérios o suporte à verificação de regras de design em relação à sinalização, levantamento, tratamento e propagação de exceções. As principais características e os pontos fortes e fracos são resumidos a seguir.

DCL: A avaliação realizada na ferramenta mostrou que apesar da linguagem possuir uma sintaxe que permite ter acesso a informações do tratamento de exceção e verificação de conformidade, fica claro que seu foco específico está nas restrições de dependência entre módulos. Embora ela ofereça regras que permitem de tratamento de exceção, essas regras ainda são muito superficiais e abaixo do nível que foi considerado para o desenvolvimento deste trabalho, isso porque ela não possui suporte específico ao design de tratamento de exceção. Contudo, a DCL se assemelha a este trabalho em um único ponto, a sintaxe das restrições de dependência foi importante na definição das regras de design deste trabalho.

LDM: O resultado da avaliação na ferramenta mostrou que é inviável a utilização da LDM para verificação de tratamento de exceção. A LDM possui suporte à definição de regras de design, no entanto, sua sintaxe e poder de expressão são muito simples, e não permitem a expressão de regras no contexto do tratamento de exceção. Como na DCL, a ferramenta LDM possui um objetivo específico, o seu foco é representar unicamente dependências entre módulos, por esse motivo ela não torna possível a verificação exceções.

.QL: A avaliação na linguagem .QL e na sua respectiva ferramenta, mostrou que ela é uma linguagem relativamente simples, o que se dá pelo fato de ter sua sintaxe baseada no SQL. Ela também possui um grande poder de expressão já que é originada da linguagem Datalog. No entanto, mesmo sendo simples e poderosa ela não foi considerada ideal para verificação de conformidade de regras de design. Ela não possui na sua sintaxe meios que permitam ao designer fazer consultas que envolvam o contexto de tratamento de exceção. Portanto, chegou-se à conclusão que é inviável o uso da ferramenta para a verificação de tratamento de exceção.

SAVE: A avaliação mostrou que assim como as ferramentas DCL e LDM, ela possui um objetivo muito específico, e que não diz respeito ao tratamento de exceção, pois o seu pronto principal é gerenciar a dependência entre módulos. Concluiu-se na avaliação, que ela não possui suporte à definição de regras de design e nem a verificação de tratamento de exceção. Desse modo, ela não poderia ser utilizada para verificação de conformidade no contexto de tratamento de exceção.

Design Wizard: A avaliação da ferramenta mostrou que é uma API flexível, de uso simples e que permite realizar diversos tipos de verificações no código fonte. Por esse motivo, ela foi a ferramenta escolhida para ser utilizada como suporte à API desenvolvida neste trabalho. As regras de design utilizam a Design Wizard para extrair informações sobre tratamento de exceção do código fonte, o que torna possível por exemplo, saber quem sinaliza, captura, ou levanta uma determinada exceção.

A seguir, na tabela 2, foram apresentados os resultados da avaliação das ferramentas.

Tabela 2 Resultados da avaliação das ferramentas

	DCL	*LDM*	*.QL*	*SAVE*	*DESIGNWIZARD*
RDL	Yes	No	No	No	Yes
RDS	No	No	No	No	Yes
RDT	Yes	No	No	No	Yes
RFP	No	No	No	No	Yes
Yes : Atende ao critério			**No: Não atende ao critério**		

Fonte: Elaborado pelo autor.

Observando os dados da tabela é possível perceber que a linguagem DCL, atende apenas aos critérios de levantamento e tratamento de exceção, isso porque em sua sintaxe ela expressa regras de que permitem realizar uma verificação superficial em relação a quem trata e quem levanta exceções. No entanto, ela se restringe a apenas essas verificações.

Já as ferramentas LDM, SAVE e .QL, não conseguiram atender a nenhum dos critérios especificados, isso porque essas ferramentas poderosas e eficazes nas suas verificações de conformação e consultas em código fonte, não possuem nenhum suporte ao tratamento de exceção, elas não dão ao designer a liberdade nem de construir, nem de usar regras que envolvam o contexto de exceção.

Por fim, a ferramenta Design Wizard, atende completamente todos os critérios especificados. Isso porque ela possui um grande poder de expressão e possibilita ao designer construir regras de design que envolvam tratamento de exceção. No entanto, um ponto fraco da Design Wizard é que o desenvolvimento das regras não é uma tarefa trivial de ser realizada. Raciocinar em relação às regras de tratamento é uma tarefa dispendiosa e que pode levar certo tempo, assim sua principal desvantagem é não possuir as regras de design já definidas e prontas para serem reutilizadas.

6 IMPLEMENTAÇÃO DA API

6.1 Tecnologia Utilizada

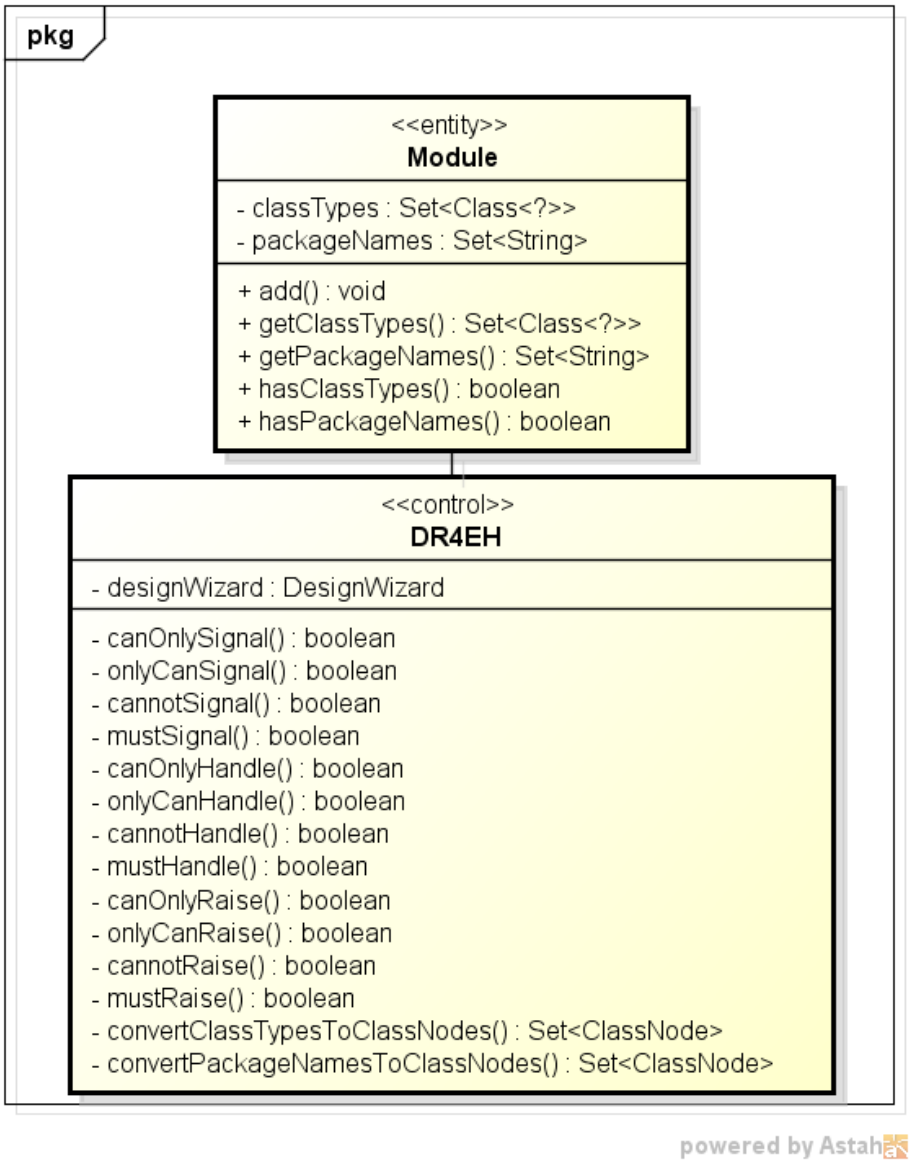
A API DR4EH (*Design Rules For Exception Handling*) foi desenvolvida na linguagem de programação Java e fazendo uso da API do Design Wizard. Com ela é possível extrair as informações estruturais do código fonte e realizar a verificação de conformação entre as regras e o código. O Java 7 foi a versão utilizada para o desenvolvimento da DR4EH. Para apoiar os testes o framework de testes automatizados JUnit na versão 3 foi utilizado para o desenvolvimento do trabalho.

6.2 Design da API

A API desenvolvida neste trabalho possui uma arquitetura simples. A DR4EH está dividida em duas classes, são elas a classe principal DR4EH, e a classe Module. A classe principal é onde estão implementadas todas as regras de design da API. Nela estão presentes as 16 regras de design que permitem verificar a sinalização, levantamento, tratamento e o fluxo de propagação de determinadas exceções. Já a classe Module, representa um módulo da aplicação a ser verificada, seja ele uma classe ou um pacote, que é passado como parâmetro

para a regras de design no momento da verificação, nela estão implementadas dois Sets, ou seja coleções para cada tipo de módulo, além disso ela é responsável pela adição dos módulos em suas respectivas coleções para que eles possam ser utilizados pela classe principal. Os métodos e atributos das classes da API podem ser vistos a seguir na Figura 6.

Figura 6 Diagrama de classes da DR4EH



Fonte: Elaborado pelo autor.

Para a implementação das regras foi definido que apenas dois tipos de parâmetros seriam necessários para verificação de conformidade. O primeiro tipo de parâmetro seria um módulo. Neste trabalho, um módulo foi considerado como um pacote, ou uma classe, que uma vez passada para o método seria convertida para o objeto necessário à verificação. O segundo

tipo de parâmetro deve ser a exceção que será verificada, a exceção deverá ser uma entidade de exception declarada na aplicação. Seguindo essa definição, as assinaturas dos métodos foram desenvolvidas como descrito na Figura 7:

Figura 7 Exemplo assinatura dos métodos

```
public boolean mustSignal(Module module, Class<?> exception) {

public boolean mustSignal(Module signalModule, Class<?> exception, Module handlerModule) {
```

Fonte: Elaborado pelo autor.

6.2.1 Classe DR4EH

As regras implementadas na classe principal, podem ser divididas em quatro categorias, *Signal Rules, Handle Rules, Raise Rules e Signal M to N Rules*. As regras da categoria *Signal Rules* são responsáveis por verificar o código à procura de violações nas sinalizações de exceções. O Java indica que exceções podem ser sinalizadas por um método utilizando a palavra throws em sua assinatura. Por exemplo, se um método deve sinalizar exceções do tipo DAOException, tudo que precisa ser feito é escrever throws DAOException na sua assinatura. Por outro lado, para que uma exceção seja levantada dentro de um método de uma classe basta usar a palavra throw, por exemplo, throw new DAOException () faz levantar uma exceção DAOException. As regras desta categoria são as seguintes:

- Um determinado modulo sinaliza uma exceção que apenas outro modulo poderia sinalizar (*onlyCanSignal*).
- Um modulo sinaliza uma exceção que está além das exceções que ele poderia sinalizar (*canOnlySignal*).
- Um modulo sinaliza um tipo de exceção especifica que não deveria ser sinalizada por ele (*cannotSignal*).
- Por fim, se um modulo não sinaliza uma exceção especifica que deveria ser sinalizada por ele (*mustSignal*).

Por meios destas regras de design é possível verificar a fundo se existem exceções que estão sendo sinalizadas ou não sinalizadas por módulos que violam as decisões principais de projeto. A seguir na figura 8 um exemplo de sinalização de exceção.

Figura 8 Exemplo de sinalização de exceção

```
public void adicionar(Contato contato) throws CTLException {
    try {
        dao.create(contato);
    } catch (DAOException dao) {
        throw new CTLException();
    }
}
```

Fonte: Elaborado pelo autor.

Além das regras específicas de sinalização existem também as de tratamento, que dizem respeito a quem captura as exceções que foram sinalizadas. São as regras da categoria *Handle Rules*. Na linguagem Java a captura de exceções pode ser percebida pelo uso da palavra catch, que significa captura. Quando um método possui por exemplo, um *catch (SQLException)*, significa que ele captura exceções do tipo SQLException e que ele pode também tratá-la. Seguindo este raciocínio, as regras desta categoria são responsáveis por verificar os seguintes casos:

- Um determinado modulo captura uma exceção que apenas outro modulo poderia capturar (*onlyCanHandle*).
- Um modulo captura uma exceção que está além das exceções que ele poderia tratar (*canOnlyHandle*).
- Um modulo captura um tipo de exceção especifica que não deve ser tratada por ele (*cannotHandle*).
- Por fim, se um modulo não captura uma exceção especifica que deveria ser tratada por ele (*mustHandle*).

Se existe uma decisão de projeto que restringe a captura de exceções por determinados módulos da aplicação as regras mencionadas fornecem suporte a este tipo de verificação. Sejam as verificações de ausência ou de divergência.

Figura 9 Exemplo de captura de exceção

```
public void adicionarContato(Contato contato) {
    try {
        controller.adicionar(contato);
    } catch (CTLException e) {
        e.printStackTrace();
    }
}
```

Fonte: Elaborado pelo autor.

Em relação ao levantamento de exceções existem as regras da categoria *Raise Signal*, estão regras estão relacionadas ao levantamento interno de exceções que acontece dentro do método. O levantamento de uma exceção é representado pelo uso da palavra *throw*, que pode ser traduzida como, jogar, lançar ou arremessar. Um método levanta uma exceção quando ele declara na assinatura do método, ou sinaliza como foi mencionado, mas não captura a mesma exceção sinalizada, dessa forma ele lança, ou levanta uma exceção para o método chamador, ou quem quer seja o responsável por tratá-la da maneira adequada. Uma vez definido o que é levantar uma exceção, as regras devem verificar os seguintes casos:

- Um determinado modulo levanta uma exceção que apenas outro modulo poderia levantar (*onlyCanRaise*).
- Um modulo levanta uma exceção que está além das exceções que ele poderia levantar (*canOnlyRaise*).
- Um modulo levanta um tipo de exceção especifica que não deve ser levantada por ele (*cannotRaise*).
- Por fim, se um modulo não levanta uma exceção especifica que deveria ser levantada por ele (*mustRaise*).

Figura 10 Exemplo de levantamento de exceção

```
public void update(Contato contato) throws DAOException {
    throw new DAOException();
}
```

Fonte: Elaborado pelo autor.

A última categoria de regras mas a não menos importante, é a categoria *Signal M to N Rules*. Pode-se perceber que as assinaturas das regras da categoria *Signal Rule*, se

assemelham com as regras desta categoria. De fato, elas são praticamente idênticas, com uma única diferença, que é o parâmetro de entrada da regra. Como as duas categorias estão relacionadas com a verificação da sinalização de exceções, os métodos foram sobrecarregados, assim o design precisa apenas decidir se sua verificação envolve apenas um, ou dois módulos. Caso sejam dois, tudo o que ele precisa fazer, é instanciar um novo modulo e adiciona-lo ao parâmetro do método.

As regras de *M to N*, dizem respeito a sinalização de exceções entre dois módulos estão de acordo com os seguintes critérios:

- Um determinado modulo M sinaliza uma exceção, que ele não poderia sinalizar, para o módulo N (*onlyCanSignal*).
- Um modulo M sinaliza uma exceção que está além das exceções que ele poderia sinalizar para N (*canOnlySignal*).
- Um modulo M sinaliza um tipo de exceção especifica, que não deveria ser sinalizada por ele, para o módulo N (*cannotSignal*).
- Por fim, se o modulo não sinaliza uma exceção especifica que deveria ser sinalizada para o módulo N (*mustSignal*).

6.2.2 Classe Module

A classe *Module* é formada por duas coleções da interface *Set*, uma coleção do tipo *Class* e outra do tipo *String*. Dessa forma, quando o designer instanciar um novo objeto do tipo *Module*, ele poderá adicionar ao objeto, tanto um tipo *String* que pode corresponder a um pacote da aplicação quanto um tipo *Class*, que pode ser entendido como uma classe específica da aplicação em questão.

No construtor da classe *Module* as coleções são instanciadas como um novo *HashSet* com o seu tipo correspondente. Em seguida para os tipos *Class* e *String*, foram implementados métodos para a adição nas coleções. Os métodos *add*, possuem uma assinatura semelhante, no entanto, com parâmetros de entrada diferentes, desse modo, os métodos são sobrecarregados, assim se um tipo *String* for passado para o método *add*, a adição será feita na coleção do tipo *String*, da mesma forma para um tipo *Class*. Feito isso, a entrada será adicionada de forma correta em sua respectiva coleção.

Para acessar a coleção correta na classe *Module*, foram dois criados dois métodos de retorno booleano, são eles o *hasClassTypes* e o *hasPackageNames*, o objetivo desses

métodos é retornar um booleano para o chamador informando que, ou a coleção de *ClassTypes* não está vazia, então a entrada foi um tipo *Class*, ou a informação seria que a coleção *PackageNames* não está vazia, o que indica que a entrada adicionada é um tipo *String*. Assim, o método que chamou a verificação sabe que tipo de entrada foi passada e que coleção ele deve acessar. Por fim, a classe *Module* possui ainda o método *Get* que se faz necessários para o acesso da classe da principal.

Figura 11 Classe Module

```
public class Module {

    private Set<Class<?>> classTypes;

    private Set<String> packageNames;

    public Module() {
        classTypes = new HashSet<Class<?>>();
        packageNames = new HashSet<String>();
    }
    public void add(Class<?> classType) {
        classTypes.add(classType);
    }
    public void add(String packageName) {
        packageNames.add(packageName);
    }
    public Set<Class<?>> getClassTypes() {
        return classTypes;
    }
    public Set<String> getPackageNames() {
        return packageNames;
    }
    public boolean hasClassTypes() {
        return !classTypes.isEmpty();
    }
    public boolean hasPackageNames() {
        return !packageNames.isEmpty();
    }
}
```

Fonte: Elaborado pelo autor.

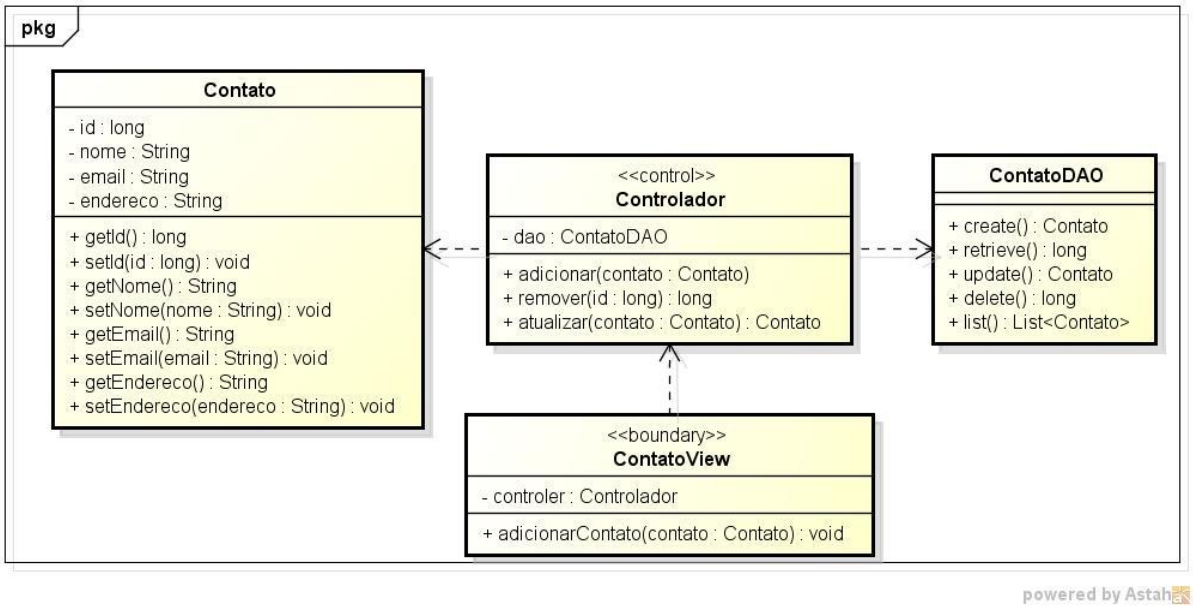
7 VALIDAÇÃO DA API DESENVOLVIDA

Este trabalho propôs o desenvolvimento de uma API que possibilita a verificação automatizada de conformidade em regras de design do tratamento de exceção em código fonte Java. Durante o desenvolvimento do trabalho foram expostas as ferramentas existentes no mercado que são utilizadas para verificação de conformidade. Cada ferramenta foi analisada e

avaliada em relação ao contexto de exceções, suas principais características foram levantadas e depois comparadas aos critérios de avaliação definidos neste trabalho. Os critérios seguiam a ideia de o que cada ferramenta deveria possuir para tornar a verificação do tratamento de exceção uma ação possível de ser realizada.

Uma vez desenvolvida a API, é necessário que ela seja validada. Para isso, foi desenvolvida uma aplicação mock, o IContatos como é possível ver na Figura 12. Nela existem várias situações relacionadas ao tratamento de exceção, como por exemplo sinalização, levantamento e tratamento de exceções.

Figura 12 Diagrama de classes do IContatos

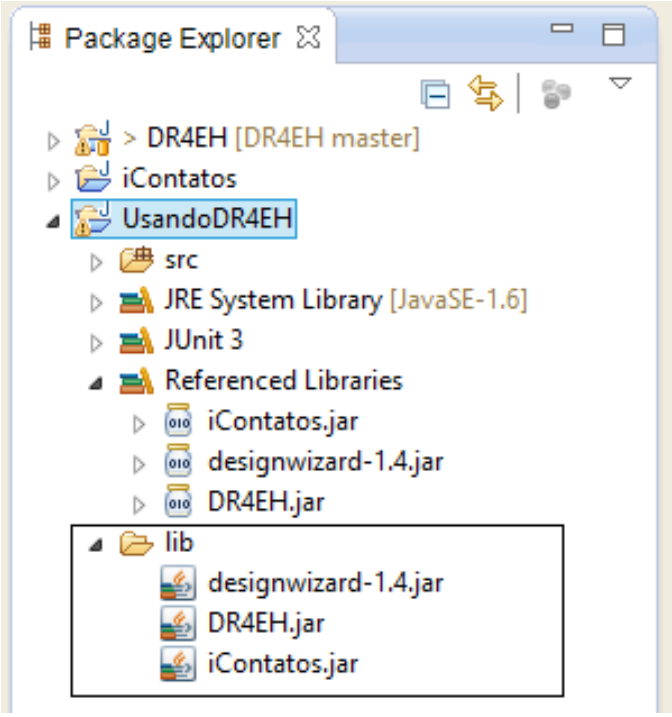


Fonte: Elaborado pelo autor.

Para auxiliar na verificação de conformidade foi utilizado o framework de teste automatizado JUnit, a ideia é verificar a conformação entre regras e código por meio de testes de software, ou como já foi citado por Pires, Brunet e Ramalho (2008), testes de design, que são um tipo de teste unitário que é escrito considerando informações estruturais do código fonte.

Os procedimentos para realizar a verificações são bastantes simples. Antes de tudo é necessário gerar um arquivo .jar da aplicação que será verificada. Com o arquivo .jar gerado, o próximo passo é criar uma pasta na raiz do projeto, e colocar nesta pasta o .jar do projeto juntamente com as APIs DR4EH, e Design Wizard. Feito isso, elas devem ser adicionadas ao classpath.

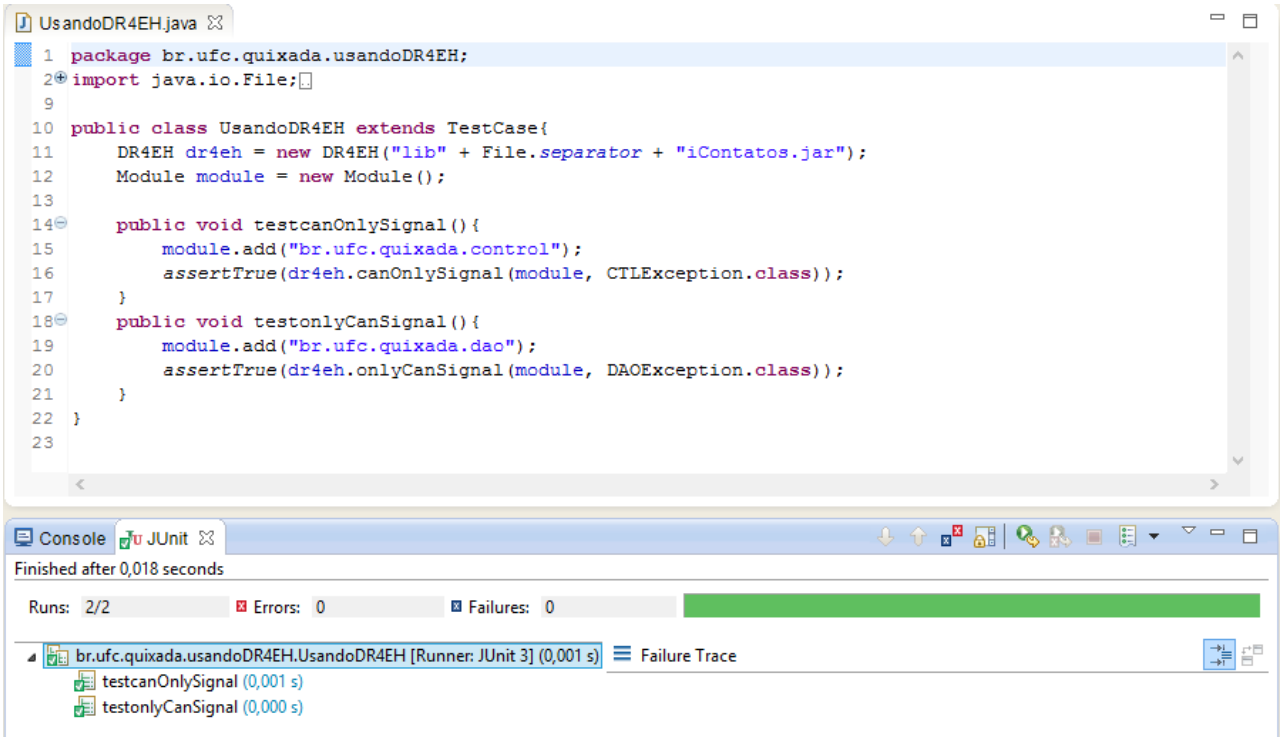
Figura 13 Pasta lib com os .jars inclusos ao classpath



Fonte: Elaborado pelo autor.

Com os jars necessários já adicionados ao classpath da aplicação, a API já pode ser usada para realizar verificações de conformidade. Suponha que no planejamento do projeto IContatos, foi tomada a decisão de que apenas os métodos das classes de um determinado módulo, podem sinalizar um tipo de exceção. Além dessa, foi tomada outra decisão, segundo ela os métodos das classes de um determinado módulo, só podem sinalizar um tipo determinado de exceção, para ter certeza que essas decisões foram cumpridas uma verificação precisa ser feita no código, para isso basta utilizar as regras já definidas na DR4EH como é possível ver a seguir na Figura 14.

Figura 14 Verificando o IContatos com a DR4EH

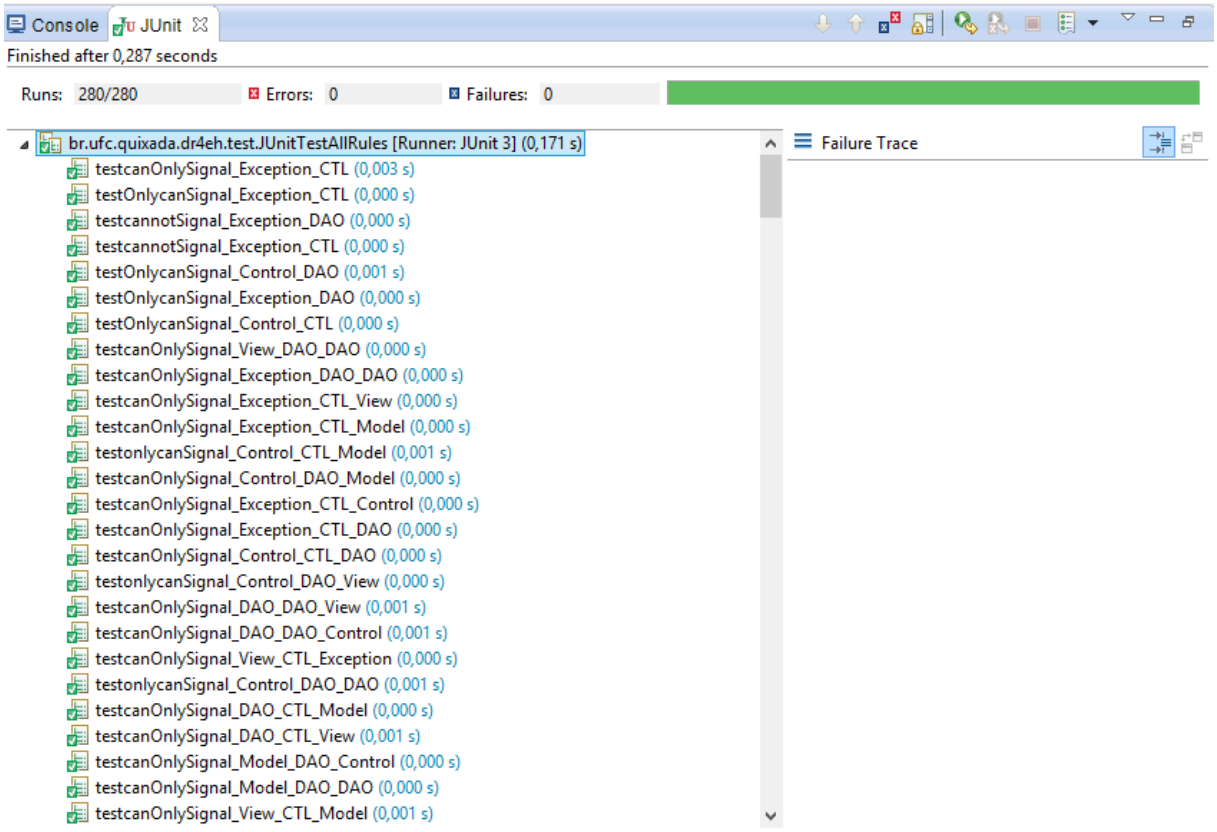


Fonte: Elaborado pelo autor.

Uma vez executados os testes, o JUnit exibira uma barra vermelha indicando falha na verificação, ou seja não conformidade, ou exibirá uma barra verde indicando que o código está em conformidade com as regras de design.

Durante o desenvolvimento a API passou por uma bateria de testes automatizados por meio do JUnit, o objetivo era garantir que versão após versão ela ainda mantinha sua corretude em relação as verificações de conformidade. Portanto, com foco na aplicação IContatos foram escritos 280 testes de software, com o objetivo de verificar cada regra e possibilidade de entrada de dados, ou seja, todas as possiblidades de combinações entre módulo e exceção foram testadas e obtiveram resultados positivos que podem ser vistos na Figura 15.

Figura 15 Conjunto de testes realizados na API



Fonte: Elaborado pelo autor.

Além do funcionamento correto da API, outro ponto que também foi considerado na avaliação foram os critérios levantadas para avaliar as ferramentas existentes, o produto deste trabalho não seria completo se ele não suprisse a necessidades apontadas em cada uma das ferramentas levantadas.

Após a realização dos testes, podemos concluir que a DR4EH atingiu o objetivo proposto, ela serve ao seu propósito, possibilita a verificação de regras de tratamento de exceção a um nível que nenhuma das ferramentas levantadas consegue chegar. Por fim, o objetivo principal proposto por este trabalho foi atingido.

8 CONSIDERAÇÕES FINAIS

Este trabalho propôs o desenvolvimento de uma API Java que oferecesse suporte à verificação de conformação entre regras de design do tratamento de exceção em código fonte.

Para atingir o objetivo deste trabalho, alguns passos foram definidos. O primeiro passo foi avaliar as ferramentas de verificação de conformação existentes na literatura. Em seguida foram definidas as regras do tratamento de exceção. Com as regras já em mãos a implementação pôde ter início, as regras foram implementadas usando a linguagem de programação Java. E o último mas não menos importante passo foi a validação da API DR4EH.

Durante a execução dos passos foram encontradas algumas dificuldades. Elas surgiram durante a definição das regras de design, o estudo da documentação da Design Wizard e na implementação das regras. Percebemos o quão difícil é raciocinar em relação ao tratamento de exceção e definir regras que sejam relevantes para uma verificação de conformação nesse contexto. Contudo, as dificuldades foram superadas e ao final da execução dos passos, o objetivo principal deste trabalho foi atingido, a API DR4EH foi desenvolvida e testada com sucesso.

Este trabalho mostrou a necessidade e a importância de raciocinar em relação ao tratamento de exceções, pois apesar de ser um mecanismo importante na detecção e recuperação de falhas, o tratamento de exceção é ignorado pela maioria dos desenvolvedores. O tratamento de exceção é umas das partes menos compreendida dos sistemas de software atuais, existem estatísticas que afirmam que em alguns casos de 40% a 72% das ações de tratamento de exceção das aplicações industrias, são ações extremamente simples e que não buscam recuperar o estado interno do sistema (BARBOSA, 2012).

Essa falta de comprometimento com o principal mecanismo de tolerância a falhas das linguagens de programação motiva este trabalho. Sabendo que os desenvolvedores ignoram o tratamento de exceção uma solução para fazer com que ele seja levado mais a sério, seria uma verificação, ainda em tempo de desenvolvimento, do código de tratamento de exceção da aplicação. Assumindo que na fase de planejamento decisões principais de projeto foram tomadas para assegurar uma melhor qualidade e robustez do software, uma verificação de conformidade entre o código de tratamento de exceção e as decisões principais de projeto, ou seja regras de design, poderia ser realizada. Como resultado, o designer teria conhecimento das violações em relação ao tratamento de exceção.

Porém, verificar a conformidade entre regras de design e código de tratamento de exceção pode rapidamente se tornar inviável de ser realizada manualmente devido à complexidade e ao grande de número de fluxos de exceções a serem seguidos (JÚNIOR; COELHO, 2011). Sabendo desta limitação, a DR4EH foi desenvolvida, ela permite a

verificação ainda em tempo de desenvolvimento por meio de testes automatizados utilizando o framework JUnit, o que torna os testes mais rápidos e simples de serem construídos.

Alguns trabalhos foram de grande importância para o desenvolvimento deste, dentro eles, talvez o mais importante seja o trabalho de Pires, Brunet e Ramalho (2008), onde foi desenvolvida uma poderosa API Java com o intuito de extrair informações estruturais do código fonte para realização de testes de design. Foi baseado neste, que a ideia da DR4EH surgiu como opção de ferramenta com o objetivo na verificação de conformação de exceções.

A maior contribuição deste trabalho foi o desenvolvimento de uma API de código aberto, para verificação de conformação do tratamento de exceções. O resultado deste trabalho poderá ser acessado e utilizado por outros desenvolvedores e pesquisadores que quiserem aprender mais sobre a ferramenta. O código fonte da aplicação está disponível no github no endereço (https://github.com/marciosn/DesignRulesForExceptionHandling).

O próximo passo para a ferramenta, seria a definição de mais regras de design, com o intuito de tornar a ferramenta ainda mais abrangente em relação ao tratamento de exceção. Adicionar ainda a possibilidade de um teste contendo várias exceções e módulos aninhados, gerando um grande e único teste para uma parte especifica da aplicação. Entendemos que este trabalho abre portas para futuros novos trabalhos ou para extensões deste, durante o desenvolvimento citamos problemas com a baixa qualidade do tratamento de exceção das aplicações atuais, foram mostrados estudam que corroboram com essa afirmação, mostramos o quão é difícil raciocinar em relação ao tratamento de exceção, e ressaltamos a importância de ferramentas automatizadas para os processos de verificação de conformação no código fonte. Assim, expomos nossa ideia da importância do tratamento de exceção e a necessidade de ferramentas capazes de auxiliar nessa busca constante por qualidade nos sistemas de software.

REFERÊNCIAS

BRUNTINK, Magiel; DEURSEN, Arie Van; TOURWÉ, Tom. **Proceedings of the 28th international conference on Software engineering. In: International Conference on Software Engineering**, 28., 2006, [S. l.]. Discovering Faults in Idiom-Based Exception Handling**.** [S. l.]: Acm, 2006. p. 242 - 251.

BARBOSA, Eiji Adachi Medeiros. **Sistema de Recomendação para Código de Tratamento de Exceções.** 2012. 125 f. Dissertação (Mestrado) - Puc-rio, Rio de Janeiro, 2012.

CAI, Yuanfang; WANG, Hanfei; WONG, Sunny. **Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures**. In: COMPARCH, 9., 2006,

[S. l.].Leveraging Design Rules to Improve Software Architecture Recovery**.** [S. l.]: Acm, 2013. p. 133 - 142.

EBERT, Felipe. **An Exploratory Study on Exception Handling Bugs in Java Programs.** 2013. 97 f. Dissertação (Mestrado) - Ufpe, Recife, 2013.

GURP, Jilles Van; BOSCH, Jan; BRINKKEMPER, Sjaak. Design Erosion in Evolving Software Products. In: **EVOLUTION OF LARGE-SCALE INDUSTRIAL SOFTWARE EVOLUTION**, ., 2003, Amsterdam. Design Erosion in Evolving Software Products**.** Amsterdam: Elisa Workshop, 2003. p. 134 - 139.

JIANG, Shujuan et al. An Approach to Automatic Testing Exception Handling. **Acm Sigplan Notices**, [S. l.], n. , p.34-39, 08 ago. 2005.

MORGAN, Clint; VOLDER, Kris De; WOHLSTADTER, Eric. A Static Aspect Language for Checking Design Rules. In: **PROCEEDINGS OF THE 6TH INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT**, 6., 2007, [S. l.]. A Static Aspect Language for Checking Design Rules. [S. l.]: Aosd, 2007. p. 63 - 72.

OW2, Consortium. **ASM.** Disponível em: <http://asm.ow2.org/>. Acesso em: 28 nov. 2013.

PAZ, Joao Antonio Cardoso da Mata Oliveira da. **Verificação de consultas .SQL usando alloy**. 2010. 60 f. Dissertação (Mestrado) - Curso de Engenharia de Teleinformática, Univeridade do Minho, S.l, 2010.

PIRES, W. F. N. ; BRUNET, J. A. M. ; RAMALHO, F. S. ; GUERRERO, D. S. . UML-based Design Test Generation. In: **23nd Annual ACM Symposium on Applied Computing (SAC 2008)**, Fortaleza - Brasil. **23nd Annual ACM Symposium on Applied Computing (SAC 2008)**, Software Engineering (SE), 2008.

ROCHA, Lincoln S.. **Tratamento de Exceção em Sistemas Ubíquos: Evolução do Tratamento e Requisitos Desafiadores**. 2011. 10 f. Dissertação (Qualificação Doutorado) - Ufc, [S. l.], 2011.

SALES JÚNIOR, Ricardo J.; COELHO, Roberta. Preserving the Exception Handling Design Rules in Software Product Line Context: A Practical Approach. In**: DEPENDABLE COMPUTING WORKSHOPS (LADCW), 2011 FIFTH LATIN-AMERICAN SYMPOSIUM ON**, 5., 2011, Sao Jose Dos Campos. Preserving the Exception Handling Design Rules in Software Product Line Context: A Practical Approach. [S. l.]: Ladcw, 2011. p. 9 - 16.

SHAH, Hina B.; GORG, Carsten; HARROLD, Mary Jean. **Understanding Exception Handling: Viewpoints of Novices and Experts.** Ieee Transactions On Software Engineering, [S. l.], n. , p.150-161, abr. 2010.

SHAH, Hina; GÖRG, Carsten; HARROLD, Mary Jean. **Why Do Developers Neglect Exception Handling?** In: ACM SIGSOFT, 16., 2008, Atlanta. WEH. Atlanta: Fse, 2008. p. 62 - 68.

TERRA, Ricardo; VALENTE, Marco Tulio. **A dependency constraint language to manage object-oriented software architectures**. Software—practice And Experience, [S. l.], p. 1073-1094. 08 jun. 2009.

TERRA, Ricardo; VALENTE, Marco Tulio. **Definição de Padrões Arquiteturais e seu Impacto em Atividades de Manutenção de Software.** VII Workshop de Manutenção de Software Moderna (WMSWM), p. 1-8, 2010.

TERRA, Ricardo; VALENTE, Marco Tulio**. Verificação Estática de Arquiteturas de Software utilizando Restrições de Dependência**. II Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS), p. 24-37, 2008.

VILLELA, Ricardo Terra Nunes Bueno. **Conformação Arquitetural utilizando Restrições de Dependência entre Módulos**. 2009. 87 f. Dissertação (Mestrado) - Pontifícia Universidade Católica de Minas Gerais, Belo Horizonte, 2009.