UNIVERSIDADE FEDERAL DO CEARÁ

CENTRO DE CIÊNCIAS

DEPARTAMENTO DE COMPUTAÇÃO

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MESTRADO ACADÊMICO EM CIÊNCIA DA COMPUTAÇÃO

JUAREZ DE LIMA MENESES FILHO

ARCATCH: UMA SOLUÇÃO PARA VERIFICAÇÃO ESTÁTICA DE CONFORMIDADE ARQUITETURAL DO TRATAMENTO DE EXCEÇÃO

FORTALEZA - CEARÁ - BRASIL

2016

Juarez de Lima Meneses Filho

ArCatch: Uma Solução para Verificação Estática de Conformidade Arquitetural do Tratamento de Exceção

Trabalho apresentado ao Programa de Mestrado e Doutorado em Ciência da Computação - MDCC do Departamento de Computação da Universidade Federal do Ceará como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Lincoln Souza Rocha

Coorientadora: Profa. Dra. Rossana Maria de Castro Andrade

Fortaleza - Ceará - Brasil

2016

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

M488a Meneses Filho, Juarez de Lima.
ArCatch: Uma Solução para Verificação Estática de Conformidade Arquitetural do Tratamento de Exceção / Juarez de Lima Meneses Filho. – 2016.
81 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2016.
Orientação: Prof. Dr. Lincoln Souza Rocha.
Coorientação: Profa. Dra. Rossana Maria de Castro Andrade.

1. Design do Tratamento de Exceção. 2. Erosão do Tratamento de Exceção. 3. Verificação de Conformidade Arquitetural. I. Título.

CDD 005

Juarez de Lima Meneses Filho

ArCatch: Uma Solução para Verificação Estática de Conformidade Arquitetural do Tratamento de Exceção

Trabalho apresentado ao Programa de Mestrado e Doutorado em Ciência da Computação - MDCC do Departamento de Computação da Universidade Federal do Ceará como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Trabalho aprovado. Fortaleza - Ceará - Brasil, 23 de novembro de 2016:

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

Profa. Dra. Rossana Maria de Castro Andrade (Coorientadora)
Universidade Federal do Ceará

Prof. Dr. Marco Túlio Valente
Universidade Federal de Minas Gerais

Prof. Dr. Windson Viana de Carvalho
Universidade Federal do Ceará

Fortaleza - Ceará - Brasil
2016

Dedico este trabalho primeiramente a Deus, que sempre me deu paz, saúde e fé. A minha família, que sempre me incentivou. Aos amigos, pela parceria tanto nos momentos fáceis quanto nos difíceis. Aos professores, pela dedicação e comprometimento em seu ofício.

Agradecimentos

À Deus por sempre ter me dado paz, saúde, fé e me fazer seguir sempre em frente.

À minha família por me apoiar mesmo nos momentos difíceis, principalmente meus pais, Juarez de Lima Meneses e Helena Zita Lima, pela educação, amor, dedicação e muitas outras contribuições tanto para minha vida pessoal como profissional. À Regina Mara, minha namorada, pelo incentivo, apoio e compreensão. Ao Máquison Felipe, meu amigo de infância, pela parceria, ideias e apoio.

Aos meus colegas de mestrado Mardson Ferreira, Ildo Ramos, Rodolpho Uchoa, Leonara Braz, Mariana Carneiro, Marcio Correia, Willian Almeida, Geovanny Oliveira, Julio Sibaja e aos meus amigos de doutorado Luís Fernando, Ernando Gomes, Narciso Arruda e Bruno Leal, que sempre estiveram presentes nos momentos de estudo e muito trabalho, compartilhando grande parte da minha vida acadêmica no mestrado, os levo para a vida como grandes amigos.

Aos meus orientadores, Profa. Rosana Maria de Castro Andrade e Prof. Lincoln Souza Rocha, pela excelente orientação e dedicação para a elaboração deste trabalho e por seus conselhos e ensinamentos que levarei para toda a minha vida pessoal e profissional.

Aos professores Victor Campos, Carlos Fisch, João Paulo Pordeus, Windson Viana e Fernando Trinta, que contribuíram para minha formação acadêmica durante o mestrado.

Ao grupo de pesquisa GREat, por ter me acolhido e fornecido todo o suporte e infraestrutura necessários para o cumprimento das atividades realizadas durante o mestrado.

Ao CNPq pela bolsa de fomento concedida, imprescindível para realização deste trabalho de dissertação.

À todas as pessoas que contribuíram para a elaboração deste trabalho e para minha formação acadêmica de forma direta ou indireta.

"Você pode encarar um erro como uma besteira a ser esquecida,
ou como um resultado que aponta uma nova direção."
(Steve Jobs)

Resumo

O tratamento de exceções é uma técnica de recuperação de erros tipicamente empregada na melhoria da robustez de software. No entanto, estudos recentes relatam que o tratamento de exceção é comumente negligenciado pelos desenvolvedores e é a parte menos compreendida e documentada de um projeto de software. A falta de documentação e a dificuldade em compreender o design do tratamento de exceção pode levar os desenvolvedores a violarem decisões de design importantes, desencadeando um processo de erosão no design do tratamento de exceção. A verificação de conformidade arquitetural fornece meios para controlar a erosão arquitetural, verificando periodicamente se a arquitetura real mantem-se consistente com a arquitetura planejada. No entanto, as abordagens disponíveis não fornecem um suporte adequado para verificação da conformidade do tratamento de exceção. Para preencher essa lacuna, neste trabalho é proposta ArCatch: uma solução de verificação de conformidade arquitetural para lidar com a erosão do design do tratamento de exceção. ArCatch fornece: (i) uma linguagem específica de domínio declarativa para expressar restrições de design relativas ao tratamento de exceção; e (ii) um verificador de regras de design para verificar automaticamente a conformidade do tratamento de exceção. Foi avaliada a utilidade e a eficácia da abordagem proposta em um cenário de evolução composto por 10 versões de um sistema Java web existente. Cada versão foi verificada contra o mesmo conjunto de regras de design de tratamento de exceção. Com base nos resultados e no *feedback* fornecido pelo arquiteto de software do sistema, a ArCatch provou ser útil e eficaz na identificação de problemas de erosão do tratamento de exceção existentes e localizar suas causas no código-fonte.

Palavras-chave: Design do Tratamento de Exceção; Erosão do Tratamento de Exceção; Verificação de Conformidade Arquitetural.

Abstract

Exception handling is a common error recovery technique employed to improve software robustness. However, studies have reported that exception handling is commonly neglected by developers and is the least understood and documented part of a software project. The lack of documentation and difficulty in understanding the exception handling design can lead developers to violate important design decisions, triggering an erosion process in the exception handling design. Architectural conformance checking provides means to control the architectural erosion by periodically checking if the actual architecture is consistent with the planned one. Nevertheless, available approaches do not provide a proper support for exception handling conformance checking. To fulfill this gap, this work proposes ArCatch: an architectural conformance checking solution to deal with the exception handling design erosion. ArCatch provides: (i) a declarative domain-specific language for expressing design constraints regarding exception handling; and (ii) a design rule checker to automatically verify the exception handling conformance. The usefulness and effectiveness of the approach was evaluated in an evolution scenario composed by 10 versions of an existing web-based Java system. Each version was checked against the same set of exception handling design rules. Based on the results and the feedback given by the system's software architect, the ArCatch proved useful and effective in the identification of existing exception handling erosion problems and locating its causes in the source code.

Keywords: Exception Handling Design; Exception Handling Erosion; Architecture Conformance Checking.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface
AST	Abstract Syntax Tree
CSS	Cascading Style Sheets
DCL	Dependency Constraint Language
DSL	Domain-Specific Language
DSM	Dependency Structure Matrices
EBNF	Extended Backus-Naur Form
EPL	Exception Handling Policies Language
HTML	HyperText Markup Language
HW	Health Watcher
IC	Integração Contínua
IDE	Integrated Development Environment
IFTC	Idealized Fault-Tolerant Component
JDT	Java Development Tools
LDM	Lattix Dependency Manager
SAVE	Software Arquiteture Visualization and Evaluation
SQL	Structured Query Language
TamDera	Taming Drift and Erosion in Architecture
UML	Unified Modeling Language

Sumário

1 Introdução

Esta dissertação propõe a ArCatch, uma solução para verificação estática de conformidade arquitetural do tratamento de exceção. Na Seção 1.1 deste capítulo, são introduzidas a contextualização e a motivação deste trabalho. A definição do objetivo, as metas e as principais contribuições desta dissertação são declaradas na Seção 1.2. A Seção 1.3 finaliza o capítulo apresentando a estrutura organizacional do conteúdo deste documento.

1.1 Contextualização e Motivação

Na Engenharia de Software, o tratamento de exceção é uma técnica de recuperação de erros tipicamente empregada na melhoria da robustez de sistemas de software (GOODENOUGH, 1975; PARNAS; WüRGES, 1976). Uma exceção é um evento ou uma situação anormal detectada em tempo de execução, que interrompe o fluxo de controle normal de um programa (GARCIA et al., 2001). Quando isso ocorre, o mecanismo de tratamento de exceção desvia o fluxo de controle normal para um fluxo de controle anormal (excepcional), a fim de tratar aquela situação de excepcionalidade (BUHR; MOK, 2000). O mecanismo de tratamento de exceção fornece meios para estruturar o fluxo de controle excepcional, permitindo ao programador indicar em que partes do código-fonte exceções podem ser levantadas, tratadas e propagadas. A maioria das linguagens de programação convencionais fornecem construtos que dão suporte nativo a implementação e a estruturação do código de tratamento de exceção em sistemas de software (CACHO et al., 2014b).

A erosão arquitetural é um fenômeno que ocorre quando a arquitetura implementada (também conhecida como arquitetura descritiva) de um sistema de software diverge da sua arquitetura pretendida (também conhecida como arquitetura prescritiva) (PERRY; WOLF, 1992; SILVA; BALASUBRAMANIAM, 2012). Na verdade, esse fenômeno pode ser compreendido como um efeito colateral do processo não controlado da evolução de software, em que as alterações feitas no código-fonte levam a violações das regras de design estabelecidas na arquitetura pretendida (GURP; BOSCH, 2002; TAYLOR; MEDVIDOVIC; DASHOFY, 2009). Para lidar com esse problema, a verificação de conformidade arquitetural fornece meios para controlar o fenômeno da erosão arquitetural fazendo o monitoramento automático da conformidade entre a arquitetura implementada e a pretendida (PASSOS et al., 2010). Este controle sistemático visa garantir que as decisões de design do arquiteto (e os atributos de qualidade derivados delas) estejam corretamente refletidos na implementação do sistema (CARACCIOLO; LUNGU; NIERSTRASZ, 2015). Além disso, uma vez que a verificação da conformidade arquitetural exige uma especificação de design como entrada

(e.g., definição de módulos e restrições de dependência), o conhecimento sobre as decisões de design arquitetural torna-se melhor documentado e mais fácil de compartilhar.

Apesar de sua importância, estudos reportam que o tratamento de exceção é comumente negligenciado pelos desenvolvedores, sendo considerado a parte do software menos compreendida, documentada e testada (CRISTIAN, 1989; CABRAL; MARQUES, 2007; SHAH; GORG; HARROLD, 2010; MARINESCU, 2011; MARINESCU, 2013; KECHAGIA; SPINELLIS, 2014; ZHANG; ELBAUM, 2014). Além disso, a fim de promover a manutenibilidade, linguagens de programação modernas (e.g., C#, Ruby e Python) incorporaram flexibilidades em seus mecanismos internos de tratamento de exceção que, embora tornem as alterações no código-fonte mais ágeis por não obrigarem os desenvolvedores a seguirem as restrições impostas pelo tratamento de exceção (e.g., declarar na interface de cada método a lista de exceções que ele pode sinalizar e que, portanto, devem ser tratadas ou propagadas por métodos chamadores), acabam contribuindo para o negligenciamento do tratamento de exceção por parte dos programadores (CACHO et al., 2014a). Juntas, todas essas questões podem levar os desenvolvedores a violar a intenção do arquiteto sobre a concepção do design do tratamento de exceção durante as fases de desenvolvimento, manutenção e evolução do software. Violações no design do tratamento de exceção são perigosas porque podem levar: (i) o mecanismo de tratamento de exceção a se comportar de forma errônea ou imprópria em tempo de execução; e (ii) a aparição de faltas de tratamento de exceção no código fonte do software (EBERT; CASTOR, 2013; BARBOSA; GARCIA; BARBOS, 2014; EBERT; CASTOR; SEREBRENIK, 2015). Nesta dissertação, o problema de erosão relacionada ao design do tratamento de exceção é chamado de **erosão do tratamento de exceção**.

Ebert, Castor e Serebrenik (2015) apresentam evidências de que pouca atenção é dada pelos desenvolvedores ao tratamento de exceção durante o ciclo de vida do software. No entanto, eles concluem que é necessário construir novas soluções para ajudar engenheiros de software nas atividades de análise estática e teste do código de tratamento de exceção e seu fluxo de controle. O estado da arte das soluções de verificação de conformidade (MOOR et al., 2007; EICHBERG et al., 2008; TERRA; VALENTE, 2009; GURGEL et al., 2014; CARACCIOLO; LUNGU; NIERSTRASZ, 2015) não fornece um suporte adequado para a verificação da conformidade arquitetural do design de tratamento de exceção. Por exemplo, tais soluções não fornecem uma maneira direta de especificar e verificar a relação ternária entre elementos arquiteturais que envolva dois módulos (e.g., `A` e `B`) e uma exceção (e.g., `E`), como "*módulo* `A` *não pode sinalizar a exceção* `E` *para o módulo* `B`" ou uma exceção e vários módulos (e.g., $\texttt{M}_1$, ..., $\texttt{M}_n$), como "*exceção* `E` *não pode fluir através dos módulos* $\texttt{M}_1$, ..., $\texttt{M}_n$". Estes tipos de relações podem ser utilizados para expressar restrições de dependência em relação à propagação de exceções entre módulos e o seu fluxo de controle.

Particularmente, até onde a revisão de literatura realizada durante a elaboração

desta pesquisa de mestrado conseguiu abranger, o trabalho desenvolvido por Barbosa et al. (2016) representa a primeira solução que busca garantir que políticas específicas para o tratamento de exceção sejam verificadas contra o código-fonte a fim de assegurar que determinadas regras de design sejam seguidas pelos programadores.

Com o intuito de lidar com o problema da erosão do tratamento de exceção, este trabalho de mestrado propõe ArCatch, uma solução de verificação de conformidade arquitetural para o tratamento de exceção que provê: (i) uma linguagem específica de domínio declarativa (ArCatch.Rules) para expressar restrições de design relacionadas ao tratamento de exceção; e (ii) um verificador de regras de design (ArCatch.Checker) para verificar automaticamente a conformidade do tratamento de exceção.

1.2 Objetivo e Contribuições

Levando em conta a contextualização e a motivação expostas na seção anterior, esta dissertação de mestrado tem como **objetivo geral:**

> *Desenvolver uma solução de verificação de conformidade arquitetural para combater a erosão do tratamento de exceção.*

O objetivo foi baseado em (i) nos relatos encontrados na literatura sobre a negligência do tratamento de exceção em projetos de software; (ii) nos relatos também encontrados na literatura sobre os problemas gerados pela erosão arquitetural em projetos de software; e (iii) em não terem sido encontradas, durante a atividade de revisão da literatura conduzida neste trabalho, soluções que realizassem de forma completa a verificação da conformidade arquitetural do tratamento de exceção.

Para atingir este objetivo, ele foi decomposto em 3 (três) metas (MET):

MET01 Estabelecer um conjunto de regras anti-erosão para o tratamento de exceção, baseado nas possíveis relações de dependência entre módulos e exceções;

MET02 Desenvolver uma linguagem de domínio específico para especificação de regras de design anti-erosão para o tratamento de exceção;

MET03 Desenvolver e avaliar uma ferramenta de apoio a verificação de conformidade arquitetural do tratamento de exceção.

A partir desse objetivo e dessas metas, as principais contribuições (CTR) desta dissertação de mestrado são listadas a seguir:

CTR01 *ArCatch.Rules: Uma linguagem de descrição de regras de design para o tratamento de exceção.*
Uma DSL (*Domain-Specific Language*) interna escrita em Java denominada ArCatch.Rules foi desenvolvida com a finalidade de expressar os principais tipos de relacionamentos entre módulos e exceções e, com isso, permitir a especificação de regras anti-erosão para o tratamento de exceção. Desse modo, uma especificação escrita em ArCatch.Rules descreve regras de design relativas ao tratamento de exceção e mapeamentos entre elementos arquiteturais e o código-fonte, o que pode facilitar a disseminação do conhecimento sobre as decisões arquiteturais relativas ao tratamento de exceção entre a equipe de desenvolvimento.

CTR02 *ArCatch.Checker: Uma ferramenta para verificação de regras de design do tratamento de exceção.*
Uma ferramenta de suporte denominada ArCatch.Checker foi desenvolvida com o intuito de automatizar o processo de verificação estática da conformidade e identificar os possíveis problemas de erosão do tratamento de exceção. A ArCatch.Checker realiza a verificação automática de todas as regras de design do tratamento de exceção especificadas em ArCatch.Rules. Além disso, a ferramenta disponibiliza como saída um relatório contendo o resultado da verificação de cada regra, determinando se a regra está ou não aderente a especificação. Em caso de não aderência, detalhes sobre a causa da não conformidade são dados.

1.3 Organização do Documento

Este capítulo apresentou a problemática que motiva esta dissertação de mestrado, o objetivo e as metas alcançados e as principais contribuições geradas. O restante desse documento esta organizado da seguinte forma:

Capítulo 2 - Este capítulo é dedicado à fundamentação teórica desta dissertação de mestrado. Nele são apresentados os temas relacionados a arquitetura de software, tratamento de exceção e conformidade arquitetural do tratamento de exceção.

Capítulo 3 - Este capítulo é dedicado aos trabalhos relacionados a esta dissertação de mestrado. Nele são investigados trabalhos relacionados que propõem ferramentas de verificação estática da conformidade arquitetural com aspectos relevantes ao tratamento de exceção. Além disso, um resumo comparativo entre trabalhos é apresentado.

Capítulo 4 - Este capítulo é dedicado à apresentação da solução proposta por esta dissertação de mestrado. Nele são apresentados uma visão geral da ArCatch, sua sintaxe (ArCatch.Rules) e semântica (ArCatch.Checker).

Capítulo 5 - Este capítulo é dedicado à apresentação da avaliação realizada com a solução proposta por essa dissertação de mestrado. Nele são apresentados detalhes sobre a avaliação realizada com a aplicação Health Watcher.

Capítulo 6 - Este capítulo é dedicado às considerações finais da dissertação, análise dos objetivos de pesquisa, limitações e trabalhos futuros.

2 Fundamentação Teórica

Neste capítulo são abordados os principais conceitos relacionados a problemática tratada nesta dissertação de mestrado. O capítulo está dividido, essencialmente, em 3 seções descritas a seguir. Seção 2.1 que aborda o tema arquitetura de software (definições, degradação arquitetural e conformidade arquitetural). Seção 2.2 que trata do tema tratamento de exceções. Por fim, a Seção 2.3 que traz conceitos que relacionam as duas primeiras seções e ressalta questões ligadas a conformidade do tratamento de exceção.

2.1 Arquitetura de Software

Nesta seção são inicialmente apresentadas as definições básicas para o conceito de arquitetura de software (Seção 2.1.1). Na Seção 2.1.2, o processo de degradação arquitetural resultante da evolução não controlada do software é discutido em detalhes, além disso são apresentados os conceitos de conformação arquitetural e sua verificação.

2.1.1 Definições e Conceitos Básicos

Para Perry e Wolf (1992) a arquitetura de software consiste na fórmula (2.1) e na explicação de seus termos. De acordo com essa definição, a arquitetura de software é um conjunto de elementos arquiteturais que possuem alguma organização. Os elementos e sua organização são definidos por decisões tomadas para satisfazer objetivos e restrições. São destacados três tipos de elementos arquiteturais: (i) elementos de processamento – que usam ou transformam informação; elementos de dados – que contêm a informação a ser usada e transformada; e (iii) elementos de conexão – que ligam elementos entre si. Já a organização dita as relações entre os elementos arquiteturais. Essas relações restringem a interação dos elementos de forma a atingir o objetivo do sistema.

$$\text{Arquitetura} = \{\text{Elementos, Organização, Decisões}\} \tag{2.1}$$

Já para Bass, Clements e Kazman (1998) “a arquitetura de um programa ou de sistemas computacionais é a estrutura ou estruturas do sistema, a qual é composta de elementos de software, as propriedades externamente visíveis desses elementos, e os relacionamentos entre eles”. Embora diferente da definição de Perry e Wolf (1992), essa definição deixa explícito o papel da abstração na arquitetura (quando menciona propriedades externamente visíveis) e, também, quanto ao papel das múltiplas visões arquiteturais (estruturas do sistema). É importante mencionar uso do termo “elementos de software” como as peças fundamentais da arquitetura.

Outra definição para arquitetura de software é dada por Taylor, Medvidovic e Dashofy (2009), onde é dito que a arquitetura de software é o conjunto formado pelas principais decisões de projeto (*principal design decisions*) tomadas com respeito ao software em desenvolvimento ou em evolução. As decisões de projeto representam aspectos de desenvolvimento ou evolução de software que estão relacionadas com a estrutura, comportamento funcional, interação, propriedades não funcionais e implementação do software (TAYLOR; MEDVIDOVIC; DASHOFY, 2009). São chamadas de principais, aquelas decisões de projeto que possuem relevância do ponto de vista da arquitetura do software. Essas decisões são também referenciadas como decisões arquiteturais.

A ISO/IEEE 1471-2000 (ISO/IEC 42010, 2007) define arquitetura de software como sendo a organização fundamental de um sistema, representada por seus componentes, seus relacionamentos com o ambiente e pelos princípios que conduzem seu design e evolução. Essa definição guarda semelhanças com as anteriores por mencionar que a arquitetura compreende estrutura (elementos, módulos, componentes ou subsistemas), relações e decisões (ou princípios). Entretanto, ela vai além e, como a definição de Taylor, Medvidovic e Dashofy (2009), exibe uma preocupação com respeito a evolução do software. Neste projeto, o conceito de arquitetura de software adotado segue a essência das definições de Bass, Clements e Kazman (1998), Taylor, Medvidovic e Dashofy (2009) e ISO/IEC 42010 (2007).

O projeto arquitetural traz inúmeros benefícios ao desenvolvimento de software, dentre eles podem ser citados (TAYLOR; MEDVIDOVIC; DASHOFY, 2009): (i) melhoria da comunicação – a arquitetura pode ser usada como um ponto central para a discussão entre as partes interessadas (*stakeholders*) no software; (ii) possibilita análise prévia – com a existência de um projeto de arquitetura, é possível avaliar o software contra seus objetivos antes mesmo de ser construído; (iii) favorece o reuso em larga escala – uma arquitetura de software bem planejada pode ser reusada por uma família de software; e (iv) melhoria da documentação – a documentação do projeto de arquitetura serve como guia para a implementação, verificação, validação e evolução do software.

2.1.2 Degradação Arquitetural

Como mencionado anteriormente, a arquitetura do software pode ser referenciada sob duas perspectivas: (i) de design – arquitetura prescritiva; e (ii) de implementação – arquitetura descritiva. Arquitetura prescritiva compreende o conjunto de decisões arquiteturais que refletem a intenção do arquiteto. Segundo (TAYLOR; MEDVIDOVIC; DASHOFY, 2009), a arquitetura prescritiva é um conjunto `P` constituído pelas principais decisões arquiteturais realizadas pelos arquitetos em um determinado tempo `t`. Já a arquitetura descritiva é o conjunto de artefatos que realizam, em tempo de desenvolvimento, as decisões de design do arquiteto. Para (TAYLOR; MEDVIDOVIC; DASHOFY, 2009), arquitetura

descritiva é um conjunto `D` formado pelas principais decisões arquiteturais encapsuladas por todos os artefatos do conjunto `A`. Podem fazer parte do conjunto `A` a representação das decisões arquiteturais em UML (*Unified Modeling Language*), implementações em uma linguagem de programação, modelos dos estilos arquiteturais, padrões utilizados, dentre outros.

Idealmente, durante o processo de manutenção e evolução do software, é esperado que as alterações sejam feitas primeiramente na arquitetura prescritiva e só depois refletidas na arquitetura descritiva. Entretanto, devido a situações contingenciais (e.g., prazos curtos, inexistência ou má compreensão da documentação da arquitetura descritiva e falta de ferramentas de suporte apropriadas) essas alterações acabam sendo feitas diretamente na arquitetura descritiva e não sendo refletidas na arquitetura prescritiva. Esse processo, repetido várias vezes, acaba gerando uma discrepância entre as arquiteturas prescritiva e descritiva do software. Esse fenômeno é conhecido como **degradação arquitetural** (PERRY; WOLF, 1992; SILVA; BALASUBRAMANIAM, 2012).

A degradação arquitetural pode ser classificada em **desvio** ou **erosão**. A erosão arquitetural está relacionada com as alterações na arquitetura descritiva que levam à violação de restrições de design inter-componentes (e.g., a introdução, não intencional, de uma dependência entre dois módulos de um software). Por outro lado, o desvio arquitetural relacionado com as alterações na arquitetura descritiva que violam restrições intra-componente da arquitetura prescritiva (e.g., violação de princípios de modularidade, acoplamento e coesão).

O preço a pagar pelo não cumprimento ou pela violação das regras de design estabelecidas na arquitetura prescritiva pode ser alto (e.g., perda da escalabilidade planejada, redução da manutenibilidade e diminuição da confiabilidade) (SILVA; BALASUBRAMANIAM, 2012). Portanto, é importante estabelecer maneiras de mensurar quão alinhado está um software daquilo que foi planejado. Esse é exatamente o propósito da conformação arquitetural, medida do grau de adesão da arquitetura prescritiva de um software à sua arquitetura descritiva (TERRA; VALENTE, 2009). Dessa forma, verificar a conformidade arquitetural de um software consiste em analisar a arquitetura descritiva do software contra as decisões de design contidas na sua arquitetura prescritiva.

2.1.3 Conformidade Arquitetural

A conformidade arquitetural procura garantir a conformidade entre a implementação de software (i.e., arquitetura descritiva) e sua arquitetura planejada (i.e, arquitetura prescritiva). A conformidade arquitetural é alcançada quando não existe divergência entre a arquitetura real e uma planejada (EYCK et al., 2011). A verificação da conformidade arquitetural é um processo que verifica periodicamente se a arquitetura de software real está de acordo com a planejada. As principais técnicas de verificação de conformidade

arquitetural podem ser agrupadas em três abordagens (PASSOS et al., 2010): matrizes de dependência, linguagens de consulta e modelos de reflexão.

A primeira abordagem (i.e., Modelos de Reflexão), consiste em comparar o modelo de arquitetura (i.e., arquitetura prescritiva) produzido para o sistema, pelo arquiteto de software, com o modelo de código gerado (i.e., arquitetura descritiva). O arquiteto define um modelo arquitetural de alto nível contendo os módulos do sistema, as interfaces e suas dependências. Em seguida, à medida que o código fonte é escrito, o arquiteto define o mapeamento entre os elementos descritos na arquitetura (e.g., módulos e interfaces) e os elementos de implementação (e.g., classes e pacotes). Tipicamente, são utilizadas expressões regulares para definir esse mapeamento, no qual cada elemento de implementação que casa com a expressão regular pertence a um determinado módulo da arquitetura. A saída para as ferramentas que utilizam essa técnica são violações das restrições definidas. Um exemplo de ferramenta que utiliza essa abordagem é a SAVE[1] (*Software Architecture Visualization and Evaluation*) (MURPHY; NOTKIN; SULLIVAN, 1995).

A segunda abordagem (i.e., Linguagens de Consulta), consiste na realização de consultas quaisquer diretamente no código fonte. Todos os conceitos de modelagem e mapeamento arquitetural presentes na abordagem anterior não estão presentes nesta abordagem, em contrapartida, esta contempla o conceito de buscas pontuais no código fonte para aferir sua conformidade. Nesse caso, as consultas funcionam como forma verificação das regras de design estabelecidas na arquitetura prescritiva. Um exemplo de ferramenta que utiliza esse conceito é a .QL (pronunciada como "*dot-cue-el*"). A .QL é uma linguagem de consulta orientada a objetos, inspirada na linguagem SQL (*Structured Query Language*), desenvolvida e comercializada pela *Semmle Limited*[2].

A terceira abordagem (i.e., Matrizes de Dependência), consiste na análise do código fonte e na geração de uma matriz quadrada de dependência DSM (*Dependency Structure Matrices*) com base na análise. A DSM representa os elementos de implementação de um sistema (e.g., classes ou agrupamento de classes) na forma de linhas e colunas. Quando uma célula da matriz encontra-se marcada, significa que o elemento da linha `X` faz uso do (ou tem relação com) elemento da coluna `Y`. De forma geral, a verificação da conformação arquitetural ocorre por comparação entre a DSM extraída com base na análise do código fonte e a DSM idealizada pelo arquiteto de software. Um exemplo de ferramenta que utiliza essa técnica é a LDM[3] (*Lattix Dependency Manager*).

O modelo de reflexão é adotado na maioria das ferramentas modernas de verificação da conformidade arquitetural (CARACCIOLO; LUNGU; NIERSTRASZ, 2015). Isso decorre do fato de que o modelo de reflexão fornece meios para documentar adequadamente

[1] https://fc-md.umd.edu/save/
[2] http://semmle.com
[3] http://www.lattix.com

elementos arquiteturais e decisões de design, utilizando linguagens específicas de domínio e usando isso como entrada para verificar a conformidade do código fonte. Ao adotar esse modelo, o conhecimento arquitetural pode ser melhor documentado, compartilhado e discutido entre as partes interessadas.

2.1.4 Requisitos para Verificação de Conformidade

A verificação da conformidade arquitetural é um processo complexo que envolve conceitos, decisões de design e artefatos de software, que vão desde o mais alto nível de abstração (e.g., módulos, conectores e interfaces) até um nível de detalhamento mais profundo de implementação de software (e.g., classes, pacotes e estruturas de fluxo de controle) (PASSOS et al., 2010; EYCK et al., 2011; SILVA; BALASUBRAMANIAM, 2012). Para lidar com essa complexidade, soluções de conformidade de arquitetural devem ser projetadas e implementadas para satisfazer os seguintes requisitos: **Req. 1** - prover meios para especificar conceitos e decisões de design em nível arquitetural; **Req. 2** - prover meios para mapear os conceitos arquiteturais em artefatos de implementação de baixo nível; **Req. 3** - prover um suporte ferramental para executar a verificação de conformidade automática; e **Req. 4** - prover um *feedback* preciso sobre o grau de conformidade entre a arquitetura implementada e a planejada. Para cumprir estes requisitos, algumas questões de design devem ser levados em conta.

Em relação ao primeiro requisito (**Req. 1**), uma solução de verificação da conformidade deve fornecer uma notação uniforme para especificar entidades arquiteturais e restrições de dependência. Os elementos arquiteturais incluem módulos, interfaces, componentes, conectores e outros elementos que são relevantes ao nível arquitetural. Uma restrição de dependência (regra de design) expressa a decisão de projeto de um arquiteto de software, que indica como as entidades arquiteturais podem ser usadas em conjunto. Por exemplo, considerando o estilo arquitetural cliente/servidor, no qual o cliente faz solicitações ao servidor, as seguintes regras de design podem ser escritas: "*only* `Client` *can access* `Server`" and "`Server` *cannot access* `Client`". Nesse sentido, uma DSL adequada pode ser projetada e construída para cumprir o **Req. 1**.

Considerando o **Req. 2**, uma solução de verificação da conformidade deve fornecer uma forma para especificar o mapeamento entre as entidades arquiteturais e as entidades de implementação (i.e., artefatos de implementação de software, tais como classes, pacotes e bibliotecas). Normalmente, expressões regulares são utilizadas para executar este mapeamento (EYCK et al., 2011). Neste sentido, para cada entidade arquitetural, uma expressão regular é escrita e cada nome de entidade de implementação que corresponde a tal expressão é mapeada para esta entidade. No entanto, outras técnicas de mapeamento (e.g., matriz de rastreabilidade) podem ser aplicadas para alcançar o **Req. 2**.

O **Req. 3** está relacionado com a execução automática da tarefa de verificação da

conformidade. O verificador de conformidade deve ser projetado para extrair, a partir da especificação da arquitetura, todas as informações necessárias para realizar a verificação de conformidade. É necessário ser feito, a fim de identificar as entidades arquiteturais e as descrições de mapeamento entre entidades arquiteturais e de implementação. Se a especificação da arquitetura é expressa usando uma DSL, o verificador de conformidade pode usar um processador de linguagem para extrair automaticamente todas as informações necessárias. Para realizar o mapeamento (i.e., efetivar a ligação entre as entidades), todos os artefatos de implementação devem ser devidamente empacotados e armazenados de forma que possam ser achados. Para cada tipo de regra de design provido pela solução de conformidade arquitetural, um algoritmo deve ser projetado e implementado para verificar a validade da regra contra os artefatos de implementação de software.

Uma solução de verificação da conformidade arquitetural é inútil se não fornecer um feedback adequado aos arquitetos de software e desenvolvedores. O quarto requisito (**Req. 4**) compreende esta preocupação.Dessa forma, a solução de verificação da conformidade deve fornecer um relatório claro contendo informações úteis, tais como (i) quais regras de design foram violadas, que ajudam a definir o grau da conformidade arquitetural; e (ii) quando tais violações ocorrem na implementação do software, que podem ser utilizadas para direcionar onde as alterações devem ser feitas na implementação do software, afim de a alcançar a conformidade arquitetural. Além disso, tais informações podem ser utilizadas na recomendação de refatorações automáticas, afim de realinhar as arquiteturas implementada e planejada (TERRA et al., 2015).

2.2 Tratamento de Exceção: Uma Visão Geral

Esta seção é dedicada ao tratamento de exceção. Na Seção 2.2.1 é apresentado o conceito de exceção e a sua relação com falta, erro e falha. A Seção 2.2.2 aborda questões ligadas ao levantamento, sinalização e tratamento de exceções, bem como introduz o modelo de Componente Tolerante a Faltas Idealizado. Por fim, na Seção 2.2.3, uma visão geral sobre o tratamento de exceção na linguagem de programação Java é apresentado.

2.2.1 O que é uma Exceção?

Segundo (AVIZIENIS et al., 2004), uma **falha** (*failure*) é um evento que ocorre quando o serviço entregue por um sistema (ou componente) desvia da sua especificação funcional. A falha de um serviço pode ser vista como uma transição entre o estado onde um serviço correto está sendo entregue para um estado onde um serviço incorreto passa a ser entregue. Portanto, uma falha é um evento passível de observação externa por parte dos clientes do serviço (i.e., seres humanos e outros sistemas e componentes). Um **erro** (*error*) é definido como uma parte do estado interno do sistema (ou componente) que,

se não tratado, pode levá-lo a uma posterior falha. A causa física ou algorítmica de um erro é chamada de **falta** (*fault*). A falta pode ser entendida como evento, ou sequencia de eventos, que pode levar o sistema (ou componente) a um estado interno errôneo. Consequentemente, por computação, esse estado errôneo pode propagar-se internamente até afetar o comportamento do sistema, resultando na ocorrência de uma falha. É importante mencionar que alguns erros não afetam o comportamento do sistema e, portanto, não causam falhas.

Uma **exceção** (*exception*) é um evento que modela uma situação em que o fluxo normal de execução do sistema não pode continuar (KIENZLE, 2008). Para que o sistema continue executando corretamente, o fluxo de execução deve ser desviado e uma computação adicional deve ser empregada para tratar aquela situação(BUHR; MOK, 2000). Em sistemas confiáveis, um erro, por ser considerado um evento que raramente ocorre durante a execução do sistema, pode ser modelado como uma exceção (GOODENOUGH, 1975; PARNAS; WüRGES, 1976). O tratamento de exceção provê meios para estruturar as atividades de tolerância a faltas (*fault tolerance*) através da recuperação de erros (ROCHA, 2013). Entretanto, exceções podem modelar outro tipo de situações, tais como (MILLER; TRIPATHI, 1997): (i) desvio - surgimento de um estado invalido, mas que é permitido pelo sistema; (ii) notificação - informação ao invocador da operação que o estado do sistema mudou; e (iii) idiomas - outros usos em que a ocorrência da exceção é rara em vez de anormal.

2.2.2 Exceções em Nível Arquitetural

Em nível arquitetural, exceções e seu fluxo de controle podem ser descritas por meio do modelo IFTC (*Idealized Fault-Tolerant Component*) (LEE; ANDERSON, 1990) (Figura 1). O modelo IFTC captura a essência por trás dos construtos do tratamento de exceção presentes nas principais linguagens de programação orientadas a objeto (GARCIA et al., 2001), como Java e C#. Nesse modelo, cada componente de software (chamado) pode receber solicitações de serviços de outros componentes (chamador). O componente chamado processa o pedido e envia de volta respostas normais ou exceções. Exceções podem ser classificadas em três categorias, conforme ilustrado na Figura 1: (i) exceções de interface - sinalizadas quando a requisição não está em conformidade com a interface de serviço do componente chamado; (ii) exceções de falha - sinalizadas para indicar que, por algum motivo, o componente chamado não pode responder/processar o serviço requisitado; e (iii) exceções internas - levantadas e tratadas dentro do componente chamado. As exceções sinalizadas (interface e falha) são nomeadas de exceções externas.

No modelo IFTC, a atividade do componente pode ser dividida em atividades normais e anormais (excepcionais) (Figura 1). Na atividade normal, o componente processa as requisições de serviço de acordo com sua especificação. Na atividade anormal, o

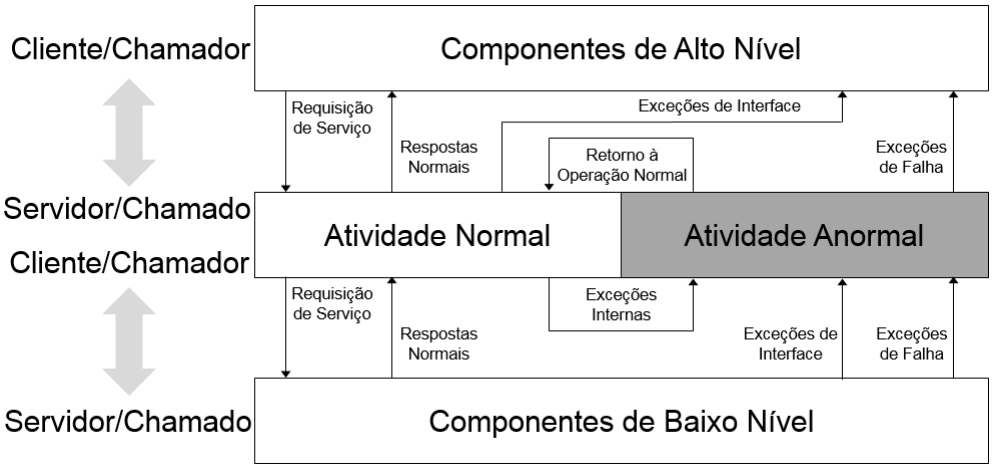


Figura 1 – Modelo de Componente Tolerante a Faltas Idealizado (adaptado de Garcia et al. (2001)).

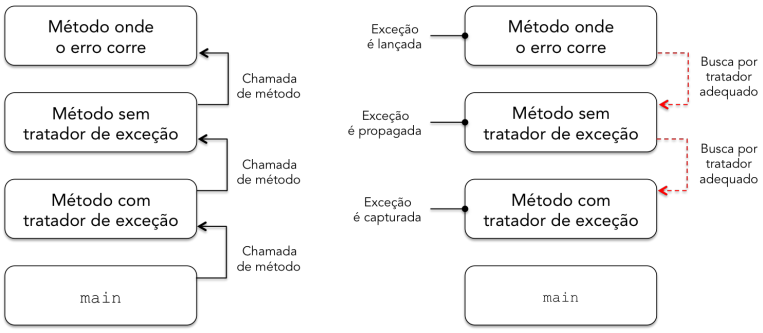
componente executa medidas de contingência para lidar com as exceções. Assim, um componente pode tratar exceções levantadas durante a sua atividade normal ou sinalizar exceções de componentes de baixo nível (chamados). No entanto, exceções que não podem ser tratadas por um componente são propagadas para componentes de alto nível (chamadores) e assim por diante. Entretanto, antes de realizar a propagação de uma exceção, um componente pode realizar tanto um relançamento quanto um remapeamento da exceção. O relançamento de uma exceção ocorre quando o componente captura a exceção, realiza alguma ação parcial de tratamento, e então a relança, forçando a continuidade da propagação da exceção. O remapeamento ocorre quando o componente captura a exceção, executa opcionalmente alguma ação parcial de tratamento, e em seguida levanta outro tipo de exceção, iniciando uma nova propagação de exceção.

Em tempo de desenvolvimento, o mecanismo de tratamento de exceção permite aos desenvolvedores definir exceções e estruturar as ações de tratamento por meio de tratadores de exceção. Os tratadores de exceção são partes dos componentes dedicadas a lidar com exceções (partes na cor cinza da Figura 1). Em tempo de execução, quando uma exceção é levantada, a mecanismo de tratamento de exceção desvia o fluxo de controle normal para um fluxo de controle excepcional, começando a busca por um tratador de exceção que pode tratar aquela exceção. A busca começa no o componente em que a exceção é levantada e prossegue através de todos os componentes da cadeia de requisição de serviço na ordem inversa em que foram chamados. Quando um tratador apropriado é encontrado, o mecanismo de tratamento de exceção passa a exceção para o respectivo tratador. Depois que a exceção é tratada, o sistema pode voltar à sua atividade normal. Caso contrário, se nenhum tratador for encontrado, o sistema fica fora de controle e é forçado a terminar a sua execução.

2.2.3 Exceções na Linguagem Java

Quando um erro ocorre (ou é identificado) em um programa Java, uma exceção é levantada (GALLARDO et al., 2014). O levantamento de uma exceção na linguagem Java é chamado de **lançamento** (*throwing*). Em Java, as exceções são representadas como objetos de classes específicas que possuem sua própria hierarquia. Essa hierarquia é dividida em duas categorias: as **exceções verificadas** (*checked exceptions*) e as **exceções não verificadas** (*unchecked exceptions*) (JENKOV, 2013). As exceções verificadas são aquelas que herdam direta ou indiretamente da classe `Exception` (e.g., `IOException` e `SQLException`), não possuindo em sua hierarquia de herança a classe `RuntimeException`. Já as exceções não verificadas são aquelas que herdam direta ou indiretamente de `RuntimeException` (e.g., `NullPointerException` e `ClassCastException`). Java não obriga o desenvolvedor a tratar exceções não verificadas, pois essas exceções são lançadas, tipicamente, pelo *runtime* da linguagem quando algum problema interno (e.g., tentativa de acesso a um elemento cujo índice está fora do limite de um vetor) ou externo (e.g., memória indisponível) ao programa é detectado. Por outro lado, as exceções verificadas requerem tratamento e são lançadas explicitamente por meio do construto `throw`.

Em Java, quando uma exceção é lançada, o fluxo de execução do programa é interrompido na linha de código que causou o lançamento da exceção. Desse modo, a execução do programa desvia para um ponto especifico do código que é capaz de tratar aquela exceção. Em Java, exceções podem ser lançadas usando o construto `throw`, propagadas usando o construto `throws`, e tratadas usando o construto `try-catch-finaly`. Esses construtos são explicados ao longo desta seção, entretanto, antes de abordá-los, é importante entender como funciona a pilha de chamada de métodos em Java. A Figura 2a ilustra uma pilha de chamada de métodos em Java. O método `main`, que iniciou a pilha de chamadas, encontra-se na parte inferior da pilha. Já o método do topo da pilha foi o último a ser chamado e é exatamente o local onde um erro ocorre.



(a) Pilha de chamada de métodos.

(b) Busca por tratador na pilha.

Figura 2 – Pilha de chamada de métodos (Figura 2a) e busca por tratador (Figura 2b) em Java (adaptado de Gallardo et al. (2014)).

Quando o erro é detectado, uma exceção é levantada. Nesse momento, o *runtime* da linguagem inicia uma busca na pilha de chamadas, em ordem reversa, por um método que possui um bloco de código capaz de tratar a exceção (veja Figura 2b). Esse bloco de código é chamado de tratador de exceção. Um tratador de exceção é considerado apropriado se o tipo da exceção lançada casa com o tipo de exceção que pode ser tratada pelo tratador (GALLARDO et al., 2014). Quando o tratador de exceção escolhido, diz-se que esse tratador captura (`catch`) a exceção. Se o *runtime* da linguagem busca exaustivamente na pilha de chamadas de métodos e não encontra o tratador apropriado, o *runtime* aborta sua execução e, por consequência, o programa também.

```
1 void metodoA() {
2    try {
3      metodoB();
4    } catch (E e) {
5      //trata a exceção
6    } finaly {
7      //sempre executa
8    }
9 }
```

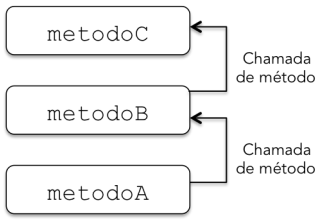
(a) Método Tratador

```
1 void metodoB() throws E {
2    (...)
3    metodoC();
4    (...)
5 }
6
7
8
9
```

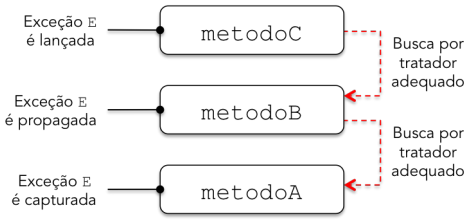
(b) Método Propagador

```
1 void metodoC() throws E {
2    if(<alguma condição>){
3    } else {
4      throw new E();
5    }
6 }
7
8
9
```

(c) Método Lançador



(d) Pilha de chamada.



(e) Busca por tratador na pilha.

Figura 3 – Exemplo de uso do `try-catch-finaly`.

A Figura 3 exemplifica o uso dos construtos `throw` (Figura 3c), `throws` (Figura 3b) e `try-catch-finaly` (Figura 3a). Na Figura 3c, a exceção do tipo `E` é lançada no método `metodoC`, linha 4 (`throw new E()`). Observe que na assinatura do `metodoC` contém o construto `throws` (linha 1) indicando que a exceção `E` será propagada para fora daquele método. Na Figura 3b, o método `metodoB` faz uma chamada ao método `metodoC` na linha 3. Nessa caso, como o método `metodoB` não fornece código de tratamento para uma possível ocorrência da exceção `E`, ele deve indicar na sua assinatura que propaga a exceção `E` (veja a linha 1 da Figura 3b). Já na Figura 3a, o método `metodoA`, que faz uma chamada ao método `metodoB`, provê um código de tratamento para a exceção `E` por meio do construto `catch` (veja linha 4 da Figura 3a). Observe que na linha 2 do método `metodoA` é utilizado o construto `try` para delimitar a região onde uma exceção pode ocorrer. Essa é a forma que Java permite ao programador indicar a região protegida. Por fim, na linha 6 do método `metodoA`, é apresentado o construto `finaly`. Esse construto circunda um bloco de comandos que sempre é executado, independente de alguma exceção ser lançada e capturada para tratamento ou não. Tipicamente utiliza-se o construto `finaly` para

estabelecer ações de limpeza (e.g., liberação de recursos). A Figura 3d ilustra a pilha de chamada de métodos e a Figura 3e a busca por tratador para os trechos de código dados.

2.3 Conformidade do Tratamento de Exceção

Esta seção é dedicada a apresentar questões relacionadas a conformidade do tratamento de exceção. Na Seção 2.3.1 é apresentado o conceito de relações de dependência e como esta dissertação usa e interpreta tal conceito, além disso, são detalhados os diferentes tipos de relações de dependência e como estas podem ser usadas para construir regras de design que tornam explícito o projeto de tratamento de exceção. Já a Seção 2.3.2 aborda questões ligadas a *bugs* do tratamento de exceção, explorando sua definição e contextualização.

2.3.1 Relações de Dependência

De acordo com o modelo IFTC descrito na Seção 2.2.2, exceções podem ser levantadas, re-levantadas, sinalizadas, tratadas e remapeadas por um módulo do sistema (componente) e podem fluir através de uma lista de vários módulos até serem tratadas. Estas relações podem ser expressas como diferentes tipos de relações de dependência entre exceções e módulos em nível arquitetural. Com base nessas relações, as restrições de dependência podem ser derivadas para descrever como exceções e módulos devem ser combinados, no sentido de expressar regras de design que regem o projeto de tratamento de exceção.

A Tabela 1 resume os diferentes tipos de relações de dependência entre exceções e módulos extraídos do modelo IFTC.

Tabela 1 – Resumo dos tipos de relações de dependência.

ID	Tipo	Descrição
DT01	`M` *raises* `E`	O módulo `M` levanta exceções do tipo `E`.
DT02	`M` *re-raises* `E`	O módulo `M` re-levanta exceções do tipo `E`.
DT03	`M` *handles* `E`	O módulo `M` trata exceções do tipo `E`.
DT04	`M` *signals* `E`	O módulo `M` sinaliza exceções do tipo `E`.
DT05	`M` *remaps* `E` to `F`	O módulo `M` remapeia exceções do tipo `E` para exceções do tipo `F`.
DT06	`E` *flows* $\texttt{M}_1 \ldots \texttt{M}_n$	As exceções do tipo `E` são levantadas e sinalizadas pelo módulo $\texttt{M}_1$ e fluem através dos módulos $\texttt{M}_2, \ldots, \texttt{M}_{n-1}$ até serem tratadas pelo módulo $\texttt{M}_n$.

Os tipos de dependência DT01-DT04 caracterizam as relações em que os módulos são capazes de levantar, propagar, tratar, e sinalizar exceções. Esses tipos de relações envolvem um módulo e um tipo de exceção. A tipo de dependência DT05 caracteriza uma relação em que um módulo é capaz de remapear exceções de algum tipo para outro. Este

tipo relação envolve um módulo e dois tipos de exceção. O último tipo de dependência (DT06) caracteriza uma relação em que um tipo de exceção pode ser propagada através de vários módulos até ser tratada. Este tipo relação envolve um tipo de exceção e vários módulos, representando o fluxo de controle excepcional de um tipo de exceção específica.

Um conjunto de regras de design para o tratamento de exceção podem ser expressas através da aplicação de modificadores semânticos (e.g., "*must*", "*cannot*", "*only... can*", e "*can... only*") para restringir as relações de dependência entre os módulos e tipos de exceção. Estas regras de design podem ser usadas para documentar e tornar explícito a intenção/decisão do arquiteto/designer sobre o tratamento de exceção e seu fluxo de controle. Por exemplo, essas regras de design podem deixar explícito que (i) os módulos podem, não podem, ou devem levantar, propagar, sinalizar, ou tratar um tipo de exceção específica; (ii) os módulos podem, não podem, ou devem propagar um tipo de exceção específica para outro; e (iii) tipos de exceção podem, não podem, ou devem fluir através de uma lista específica de módulos.

Nesta dissertação é proposta uma maneira de expressar esses tipos de regras de design e utilizá-las para verificar a conformidade do projeto de tratamento de exceção, a fim de combater problemas de erosão.

2.3.2 Faltas de Tratamento de Exceção

Faltas (ou *bugs*) de tratamento de exceção são "faltas relacionadas com a definição, lançamento, propagação, tratamento e documentação de exceções" (BARBOSA; GARCIA; BARBOS, 2014). Segundo (EBERT; CASTOR, 2013), uma falta no tratamento de exceção pode "ocorrer quando a exceção é definida, lançada, propagada, tratada ou documentada; na ação de limpeza de uma região protegida onde a exceção é lançada; quando a exceção deveria ter sido lançada ou tratada embora não lançada ou tratada".

Ebert, Castor e Serebrenik (2015) apresenta uma consolidação dos catálogos de faltas (*bugs*) de tratamento de exceção propostos por (EBERT; CASTOR, 2013) e (BARBOSA; GARCIA; BARBOS, 2014). Este catálogo consolidado apresenta cerca de 15 classificações organizadas em categorias e subcategorias. Algumas dessas categorias de faltas podem estar relacionadas com um mau design do tratamento de exceção, podendo ser evitadas através da verificação estática da conformidade arquitetural do tratamento de exceção (BARBOSA et al., 2016). São exemplos de categorias de faltas de tratamento de exceção que podem ser evitadas via verificação de conformidade: (i) Remapeamento Destrutivo (*Destructive Remapping*) - ocorre quando uma exceção é capturada por um bloco *catch*, remapeada a um tipo de exceção diferente e em seguida é relançada; (ii) Exceção Não Capturada (*Uncaught Exception*) - ocorre quando uma exceção atinge um determinado ponto de entrada do sistema e não é tratada por tratador algum; (iii) Levantamento de Tipo Genérico Pouco Informativo (*Uninformative Generic Type Thrown*) - ocorre quando uma

exceção com um tipo excessivamente genérico é levantada e, portanto, os módulos clientes (chamadores) não podem implementar ações de tratamento adequadas; e (iv) Bloco Catch Excessivamente Genérico (*Overly-Generic Catch-Block*) - ocorre quando um bloco *catch* tem como argumento uma exceção com um tipo excessivamente genérico e indevidamente captura uma exceção de subtipo, levando o sistema a um estado inesperado de erro.

A falta de Remapeamento Destrutivo pode ser evitada garantindo que o sistema de software não viole a regras de design do tipo "`Modulo A` *cannot remap* `Exception E` to `Exception Y`". Onde um determinado módulo do sistema `A` não possa remapear uma exceção específica `E` para uma outra de tipo diferente `Y`. Já a falta de Exceção Não Capturada pode ser evitada garantindo que o sistema de software não viole as regras de design do tipo "`Modulo A` *must handle* `Exception E`. Onde um determinado módulo do sistema `A` deva, obrigatoriamente, tratar uma exceção específica `E`. A falta de Levantamento de Tipo Genérico Pouco Informativo pode ser evitada garantindo que o sistema de software não viole as regras de design do tipo "`Modulo A` *cannot raise* `Exception E`". Onde um determinado módulo `A`, que pode ser mapeado para representar todo o sistema, não possa levantar uma exceção específica `E` (exceção de tipo genérico). Já em relação ao quarto e último exemplo de categoria, a falta de Bloco Catch Excessivamente Genérico pode ser evitada garantindo que o sistema de software não viole as regras de design do tipo "`Modulo A` *cannot handle* `Exception E`". Onde um determinado módulo `A`, que pode ser mapeado para representar todo o sistema, não possa tratar uma exceção específica `E` (exceção de tipo genérico).

2.4 Considerações Finais

Este capítulo apresentou a fundamentação teórica necessária para o embasamento desta dissertação. Foram detalhados três grandes tópicos: Arquitetura de Software, Tratamento de Exceção e Questões Relacionadas a Conformidade do Tratamento de Exceção. Inicialmente na Seção 2.1 Arquitetura de Software, foram abordados as principais definições e conceitos básicos de arquitetura de software presentes nesta dissertação. Ainda na Seção 2.1 foram apresentados os conceitos de degradação e conformidade arquitetural. Já na Seção 2.2, foram explorados os detalhes sobre o tratamento de exceção, mostrando uma visão geral sobre o tema, foi definido desde o quê é tratamento de exceção até como funciona o mecanismo de tratamento de exceção nas linguagens de programação em geral. Ainda na Seção 2.2 foram abordados os temas tratamento de exceção em nível arquitetural e exceções na linguagem Java. Fechando o capítulo, na Seção 2.3 foram explicadas algumas questões sobre a conformidade do tratamento de exceção, definindo o conceito de relações de dependência e como essas relações podem derivar restrições de dependência entre módulos e exceções. Em um segundo momento, foram explicados os conceitos de falhas e *bugs* do tratamento de exceção. Toda a fundamentação teórica apresentada nesse capítulo

detalham e exploram em uma maior profundidade todos os aspectos mencionados na contextualização e motivação apresentada no Capítulo 1, e constroem a base fundamental para a compreensão e execução desta dissertação. No próximo capítulo serão apresentados os principais trabalhos relacionados a esta dissertação de mestrado, apresentando uma visão geral de cada uma das soluções propostas nesses trabalhos e uma análise comparativa entre cada uma das soluções incluindo a própria solução proposta nesta dissertação.

3 Trabalhos Relacionados

Neste capítulo, é apresentado um estudo comparativo sobre soluções existentes de verificação de conformidade arquitetural. Essas soluções foram analisadas levando em consideração o seu suporte a verificação de conformidade do tratamento de exceção com respeito as relações de dependência listadas na Tabela 1 da Seção 2.3.1 do Capítulo 2 desta dissertação. As soluções analisadas foram: Semmle.QL (MOOR et al., 2007), LogEn (EICHBERG et al., 2008), DCL (TERRA; VALENTE, 2009), TamDera (GURGEL et al., 2014), Dicto (CARACCIOLO; LUNGU; NIERSTRASZ, 2015) e EPL (BARBOSA et al., 2016). Inicialmente, na Seção 3.1 um breve resumo das soluções é oferecido. Em seguida, na Seção 3.2, uma visão comparativa das soluções é apresentada. Por fim, a Seção 3.3 é reservada às considerações finais deste capítulo.

3.1 Abordagens para Verificação da Conformidade Arquitetural

Nesta seção, são apresentados os principais trabalhos relacionados a esta dissertação de mestrado. O objetivo é apresentar, de forma resumida, as principais características de cada trabalho para que depois, na Seção 3.2, seja feita uma análise comparativa.

3.1.1 Semmle .QL

Semmle .QL (MOOR et al., 2007) (pronunciado como "dot-cue-el")[1] é uma ferramenta que realiza a verificação de conformidade por meio de consultas ao código fonte. Semmle.QL foi inspirada na linguagem de consulta SQL (*Structured Query Language*). Embora seja sintaticamente parecida com SQL, ela é semanticamente diferente. Sua semântica é baseada em Datalog, uma linguagem de consulta não procedural baseada na linguagem de programação lógica Prolog. Semmle .QL pode ser vista como uma linguagem de consulta orientada a objetos que permite definir consultas capazes de determinar regras de design sobre a sinalização de exceções.

3.1.2 LogEn

LogEn (EICHBERG et al., 2008) é uma ferramenta de verificação de conformidade baseada na relação de dependências entre elementos de implementação (e.g., classes e pacotes). Ela provê uma DSL (*Domain Specific Language*) interna escrita em Datalog. A verificação de conformidade em LogEn é feita estaticamente em um ambiente integrado ao Eclipse IDE. Além disso, LogEn oferece uma notação visual, denominada VisEn, que

[1] http://semmle.com

permite ao arquiteto definir regras de design em alto nível. Na avaliação realizada, foi constado que LogEn só provê suporte à especificação de regras de design relativas a sinalização de exceções.

3.1.3 DCL Suite

DCL (*Dependency Constraint Language*) (TERRA; VALENTE, 2009) é uma linguagem de descrição de restrições que faz parte do pacote DCL Suite[2], uma solução de verificação de conformidade arquitetural. DCL foi implementada como uma DSL textual que permite ao arquiteto declarar módulos, descrever mapeamentos entre módulos e as classes do sistema e especificar restrições de dependência entre módulos. Uma vez definida as restrições de dependência, o engenho (*engine*) verifica se essas restrições estão sendo seguidas pelo código fonte do sistema. Além disso, o verificador de conformidade de DCL fornece recomendações de como as violações detectadas podem ser corrigidas com a aplicação de refatorações automáticas (TERRA et al., 2015). DCL fornece suporte a especificação de regras de design que conseguem capturar decisões arquiteturais ligadas à sinalização de exceções.

3.1.4 TamDera

TamDera (*Taming Drift and Erosion in Architecture*) (GURGEL et al., 2014) é uma outra solução de verificação estática de conformidade arquitetural. Ela provê uma DSL que permite ao arquiteto especificar regras para detectar a degradação arquitetural em termos de erosão e desvio. As regras anti-erosão estão relacionadas com restrições de dependência entre módulos e as regras anti-desvio estão relacionadas a métricas de software (e.g., *Lines of Code* - LOC e *Coupling Between Objects* - CBO) cujos *threshold* estabelecidos os componentes indicados devem atender. Além de prover verificação de conformidade, TamDera promove a reutilização de regras de design entre projetos de mesmo domínio. TamDera dá suporte a verificação de regras de design que conseguem capturar decisões arquiteturais relacionadas à sinalização e ao tratamento de exceções.

3.1.5 Dicto

Dictō[3] (CARACCIOLO; LUNGU; NIERSTRASZ, 2015) é uma solução de verificação de conformidade que unifica funcionalidades fornecidas por ferramentas já existentes. Para isso, Dictō provê uma DSL que, de maneira semelhante aos demais trabalhos já mencionados, permite ao arquiteto especificar restrições de dependência entre módulos. Essa especificação é mapeada pelo engenho (*engine*) Probo para a ferramenta de verificação de conformidade subjacente que verifica se essas restrições de design estão em conformidade.

[2] http://aserg.labsoft.dcc.ufmg.br/dclsuite/
[3] http://scg.unibe.ch/dicto/

Além disso, Dictō oferece suporte a construção de adaptadores para que seja possível a inclusão de novas ferramentas, o que pode enriquecer o conjunto de funcionalidades já existentes. Dictō fornece suporte a especificação de regras de design que conseguem capturar decisões arquiteturais ligadas à sinalização e ao tratamento de exceções.

3.1.6 EPL

EPL (*Exception Handling Policies Language*)(BARBOSA et al., 2016) é uma solução de verificação de conformidade voltada para a verificação de políticas de tratamento de exceção em programas Java. Em EPL, o termo políticas de tratamento de exceção significa um conjunto de decisões de design que governam o uso de exceções em um projeto de software. O EPL *Verifier*, o verificador de conformidade de EPL, é composto basicamente por dois módulos: *Rule Checker* e *Facts Extractor*. O módulo *Rule Checker* recebe uma especificação da política como entrada e, para cada regra especificada na política, ele usa o módulo *Facts Extractor* para verificar se existe métodos no código-fonte que violem as regras. Para cada regra violada, o módulo *Rule Checker* apresenta uma lista de métodos do código-fonte que violam determinada regra. O módulo *Extractor Facts* foi desenvolvido com *Eclipse Java Development Tools* (JDT), permitindo a integração com a IDE Eclipse como forma de *plugin*. Ele é responsável por analisar o código-fonte do sistema e extrair as informações necessárias para o módulo *Rule Checker*. Basicamente, ele varre o código fonte do sistema, extrai sua *Abstract Syntax Tree* (AST) e, em seguida, à analisa com o objetivo de extrair as informações necessárias relacionadas às relações de dependência do tratamento de exceção. A solução fornece suporte à especificação de regras de levantamento, re-levantamento, sinalização e tratamento de exceções.

3.2 Comparação das Abordagens

A maioria das soluções de verificação de conformidade apresentadas na Seção 3.1 adotam a análise estática como abordagem de análise, exceto Dictō que provê suporte a verificação de algumas propriedades dinâmicas do software (e.g., desempenho em termos de tempo de resposta). Nesta seção, essas soluções são analisadas de forma comparativa com respeito ao suporte dado por cada solução à verificação de conformidade dos seguintes interesses de tratamento de exceção: levantamento, re-levantamento, sinalização, tratamento, remapeamento e fluxo de exceções.

A análise comparativa tem início com DCL Terra e Valente (2009) e TamDera Gurgel et al. (2014). DCL (TERRA; VALENTE, 2009), como mencionado na seção anterior, baseia-se em relações de dependência entre os módulos e fornece recomendações sobre como as violações detectadas podem ser corrigidas com refatorações automáticas. Já TamDera (GURGEL et al., 2014), permite que o arquiteto especifique as regras para

detectar a degradação arquitetural em termos de erosão e desvio. Ambas as soluções propostas fornecem: (i) uma DSL para especificar regras; (ii) um mecanismo para verificar a conformidade arquitetural; e (iii) são disponibilizadas na forma de *plugin* da IDE Eclipse. Ambas apoiam a especificação de regras de design relacionadas à sinalização de exceções, porém, apenas TamDera suporta regras relacionadas ao tratamento de exceção, como mostra a Tabela 2. Além disso, é importante ressaltar que tanto a DCL quanto a TamDera, apesar de não tratarem todos os aspectos relativos ao tratamento de exceção, possuem características relevantes que servem de inspiração para trabalhos futuros, como é o caso das recomendações de refatoração automática da DCL e as regras anti-desvio propostas em TamDera.

Tabela 2 – Resumo dos trabalhos relacionados.

Ferramenta	Levantamento	Re-levantamento	Sinalização	Tratamento	Remapeamento	Fluxo
.QL	✗	✗	✓	✗	✗	✗
LogEn	✓	✗	✗	✓	✗	✗
DCL	✗	✗	✓	✗	✗	✗
TamDera	✗	✗	✓	✓	✗	✗
Dictō	✗	✗	✓	✓	✗	✗
EPL	✓	✓	✓	✓	✓	✗
ArCatch	✓	✓	✓	✓	✓	✓

Continuando a análise, agora é dado foco às seguintes soluções: Semmle .QL (MOOR et al., 2007) e LogEn (EICHBERG et al., 2008). A LogEn (EICHBERG et al., 2008) baseia-se em relações de dependência entre conjuntos (e.g., módulos e classes) de vários tipos de granularidade, e fornece uma DSL, assim como em DCL e TamDera, que permite ao arquiteto definir conjuntos e expressar as suas restrições de dependência. Já Semmle .QL (MOOR et al., 2007), provê uma linguagem de especificação semelhante a SQL. Para os autores, o fato de Semmle .QL ser inspirada em SQL é um fator positivo, uma vez que, sendo uma linguagem familiar para a maioria dos desenvolvedores, ele encontra baixa resistência em sua adoção. De forma parecida com TamDera, Semmle .QL fornece uma linguagem de consulta que permite a checagem diretamente no código-fonte com suporte para métricas de software. Tanto Semmle .QL como LogEn são baseadas em Datalog. Enquanto Semmle .QL provê suporte à decisões de design relacionadas apenas à sinalização de exceções, LogEn fornece suporte ao levantamento e tratamento de exceções, como pode ser observado na Tabela 2.

Introduzindo Dictō (CARACCIOLO; LUNGU; NIERSTRASZ, 2015) na análise, temos que ela fornece uma unificação de funcionalidades providas por ferramentas existentes (e.g., PMD e JMeter) via DSL que abstrai detalhes de uso de cada uma das ferramentas subjacentes. Semelhante as demais soluções analisadas, Dictō permite ao arquiteto especificar, via DSL, restrições de dependência que são checadas de forma automática.

Vale ressaltar que, até este ponto da análise, Dictō é a solução que fornece o suporte mais abrangente para o tratamento de exceção, quando comparada as outras soluções já analisadas (i.e., DCL, TamDera, Semmle .QL e LogEn), como pode ser visto na Tabela 2. Ela suporta a especificação de regras de design que podem capturar decisões de design relacionadas à sinalização e tratamento de exceções. Dictō é superada nos critérios de suporte ao tratamento de exceção por EPL (BARBOSA et al., 2016), a próxima e última solução analisada.

A solução mais robusta e focada no design do tratamento de exceção é EPL (BARBOSA et al., 2016). EPL foi projetada como uma DSL para especificar e verificar políticas de tratamento de exceção em programas Java. Um ponto importante a ressaltar, é o uso do termo políticas de tratamento de exceção ao invés de regras de design do tratamento de exceção. Nesse caso, EPL usa o termo política de tratamento de exceção para se referir ao um conjunto coeso de regras de design que descrevem o design completo do tratamento de exceção para um projeto de software. Em relação ao suporte ao tratamento de exceção, sem dúvidas, EPL é a mais completa das soluções analisadas, como pode ser visto na Tabela 2. EPL dá suporte a especificação e verificação de políticas relacionadas ao levantamento, re-levantamento, sinalização, tratamento e remapeamento. Porém, nenhum suporte é dado ao fluxo de propagação de exceções através de vários módulos. Desse modo, vale ressaltar que, apesar de EPL dizer que fornece suporte a propagação (BARBOSA et al., 2016), essa propagação está relacionada apenas com a definição ou não de exceções que um método pode sinalizar. Na realidade quando ela define que um módulo M deve propagar uma exceção E, está apenas definindo uma regra que obriga o módulo M a sinalizar uma exceção E (para fora do módulo), não deixando explícito para qual outro módulo a exceção deve ser propagada.

Além dos critérios de suporte a especificação e verificação de conformidade do design do tratamento de exceção, uma outra característica levada em conta na comparação das soluções foi **o nível de suporte dado pelas soluções a sua implantação em um ambiente de integração contínua (IC)**. Uma vez que o processo de erosão arquitetural ocorre durante a evolução do software, seria conveniente que as soluções analisadas pudessem ser implantadas em um ambiente de IC. Desse modo, seria garantido o monitoramento contínuo e automatizado da conformidade do tratamento de exceção ao longo das várias versões do software. Nessa perspectiva, apenas Dictō provê suporte a integração contínua, todas as outras soluções estão fortemente acopladas a IDEs específicas, dificultando o processo de sua utilização em um ambiente de IC. Tendo em mente essa característica, ArCatch foi concebida para dar suporte completo às características de design de tratamento de exceção e ser facilmente integrada a um ambiente de IC.

3.3 Considerações Finais

Este capítulo apresentou os trabalhos relacionados desta dissertação. Foram detalhadas as seguintes ferramentas: Semmle .QL, LogEn, DCL, TamDera, Dicto e EPL. Todas proveem suporte para verificar a conformidade arquitetural e possuem funcionalidades que proporcionam suporte ao tratamento de exceção. Adicionalmente, foi realizada uma discussão e análise de cada uma das soluções apresentadas comparando-as entre si e em relação a ArCatch, especificando em quais aspectos do tratamento de exceção as soluções proveem suporte e em quais destes nenhum suporte é dado. As análises e discussões apresentadas nesse capítulo reforçam a motivação apresentada no Capítulo 1. No próximo capítulo serão apresentados os detalhes da solução proposta, ArCatch.

4 ArCatch

Este capítulo é dedicado a descrição de ArCatch em termos da sua sintaxe, semântica e implementação. Inicialmente, uma visão geral de ArCatch é apresentada na Seção 4.1. Em seguida, na Seção 4.2, a EBNF que especifica a sintaxe de ArCatch é apresenta. A Seção 4.3 aborda os detalhes da semântica (formal e informal) de ArCatch. Em seguida, na Seção 4.4, detalhes de implementação e de uso de ArCatch são descritos. Por fim, na Seção 4.5 são realizadas as considerações finais desse capítulo.

4.1 Visão Geral

ArCatch tem como objetivo combater a erosão arquitetural do tratamento de exceção provendo uma forma padrão para documentar decisões de design relacionadas ao tratamento exceção e utilizá-las para verificar a conformidade do código fonte. Para atingir este objetivo, esta dissertação propõe (i) uma linguagem de especificação bem definida (ArCatch.Rules) para expressar regras de design do tratamento de exceção; e (ii) um verificador de regras de design (ArCatch.Checker) para executar automaticamente a verificação de conformidade. A Figura 4 descreve a abordagem proposta. ArCatch.Checker recebe como entrada o código fonte do software sob avaliação e as regras de design do tratamento de exceção escritas em ArCatch.Rules. Em seguida, é executada a verificação de conformidade e gerado um relatório que descreve quais as regras de design foram violadas e quais passaram na verificação.

Na abordagem proposta, o conhecimento dos arquitetos de software e dos desenvolvedores sobre o design do tratamento de exceção é documentado e compartilhado. Por exemplo, o conhecimento do arquiteto de software sobre as decisões de design necessárias para restringir o fluxo de controle excepcional no software em questão ficam documentadas como regras de design usando ArCatch.Rules (e.g., linhas 05-09 na Figura 4). Por exemplo, na Figura 4 a regra de design especificada na linha 6 restringe o tratamento de um conjunto de exceções (i.e., aquelas associadas ao elemento arquitetural `CTLEx`) por um conjunto de classes específicas (i.e., aquelas associadas ao elemento arquitetural `Controller`). A regra descrita na linha 6 significa: “classes do módulo `Controller` **não podem** tratar classes de exceção do módulo de exceção `CTLEx`”.

Diferente dos arquitetos, o conhecimento dos desenvolvedores sobre os detalhes de implementação presentes no código-fonte é útil e imprescindível para a realização do correto mapeamento entre elementos arquiteturais (i.e., módulos e exceções) e os seus

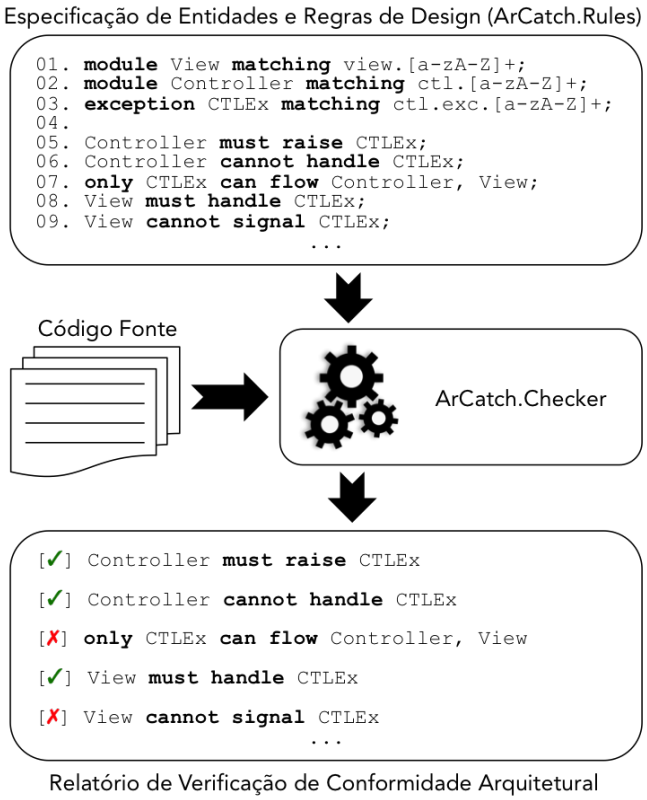


Figura 4 – Visão geral do funcionamento de ArCatch.

respectivos elementos de implementação (i.e., classes regulares e classes de exceção). Esse conhecimento também é documentado nas declarações de módulos e exceções escritas em ArCatch.Rules (e.g., linhas 01-03 na Figura 4). Por exemplo, na Figura 4 uma declaração de módulo especificada na linha 1 mapeia o módulo `View` para todas as classes no pacote `view` cujos nomes correspondem à expressão regular "`[a-Za-z]+[a-zA-Z0-9]*`". Na verdade, a especificação das regras de design do tratamento de exceção é um artefato de compartilhamento de conhecimento em que ambos, arquitetos de software e desenvolvedores, podem tirar vantagem para cumprir suas tarefas.

O relatório de verificação de conformidade arquitetural (na parte inferior da Figura 4) consiste em uma lista de todas as regras de design especificadas indicando quais regras foram aprovadas e quais não. Esse relatório é útil na identificação de quais partes do código fonte não está em conformidade com a especificação. Por exemplo, como mostrado na Figura 4, as regras de design especificadas nas linhas 5, 6 e 8 são válidas enquanto as regras de design especificadas nas linhas 7 e 9 são violadas. Além disso, para todas as regras infringidas, ArCatch.Checker gera um contra exemplo apontando quais partes do código fonte estão violando as regras. Por questões de legibilidade, essa informação não é mostrada na Figura 4, mas é apresentada posteriormente na Seção 4.4. Além disso, tanto a especificação das regras de design de tratamento de exceção quanto o relatório de verificação de conformidade, podem ajudar os arquitetos de software e desenvolvedores a melhor documentar, discutir, entender, aperfeiçoar, implementar e

evoluir as decisões de design arquitetural relacionadas ao tratamento exceção.

4.2 ArCatch.Rules: A Sintaxe

A Figura 5 descreve uma versão simplificada da EBNF (*Extended Backus–Naur Form*) de ArCatch.Rules. A especificação de design do tratamento de exceção ⟨*spec*⟩ é composta por declarações de entidades (⟨*entity*⟩) e de regras (⟨*rule*⟩). A declaração de entidade suporta dois tipos de elementos arquiteturais: módulos e exceções. Um módulo representa um conjunto de classes de implementação que fornecem uma funcionalidade bem definida e podem ser logicamente agrupadas. Uma exceção representa um conjunto de classes de implementação que representam exceções. As palavras-chave '`module`' e '`exception`' são utilizadas, respectivamente, na declaração de módulos e exceções. Ambos, módulos e exceções, possuem um identificador ⟨*id*⟩ (uma cadeia de caracteres que deve começar com uma letra) e uma expressão regular ⟨*regex*⟩ (uma sequência de caracteres que definem um padrão de busca) usada para mapear elementos de implementação (i.e., classes regulares ou classes de exceção).

⟨*spec*⟩ ::= (⟨*entity*⟩ | ⟨*rule*⟩)*

⟨*entity*⟩ ::= ('`module`' | '`exception`') ⟨*id*⟩ '`matching`' ⟨*regex*⟩ '`;`'

⟨*rule*⟩ ::= (⟨*only-can*⟩ | ⟨*can-only*⟩ | ⟨*cannot*⟩ | ⟨*must*⟩) '`;`'

⟨*relation*⟩ ::= '`raise`' | '`reraise`' | '`signal`' | '`handle`' | '`remap`' | '`flow`'

⟨*only-can*⟩ ::= '`only`' ⟨*id*⟩ '`can`' ⟨*relation*⟩ ⟨*id*⟩ ['`to`' ⟨*id*⟩ | ('`,`' ⟨*id*⟩)$^+$]

⟨*can-only*⟩ ::= ⟨*id*⟩ '`can`' ⟨*relation*⟩ '`only`' ⟨*id*⟩ ['`to`' ⟨*id*⟩ | ('`,`' ⟨*id*⟩)$^+$]

⟨*cannot*⟩ ::= ⟨*id*⟩ '`cannot`' ⟨*relation*⟩ ⟨*id*⟩ ['`to`' ⟨*id*⟩ | ('`,`'⟨*id*⟩)$^+$]

⟨*must*⟩ ::= ⟨*id*⟩ '`must`' ⟨*relation*⟩ ⟨*id*⟩ ['`to`' ⟨*id*⟩ | ('`,`'⟨*id*⟩)$^+$]

Figura 5 – EBNF simplificada de ArCatch.Rules.

A declaração de regra ⟨*rule*⟩ descreve como ArCatch.Rules expressa regras de design do tratamento de exceção como restrições de dependência entre exceções e módulos. Tais dependências podem ser expressas em termos de levantamento, re-levantamento, sinalização, tratamento, remapeamento e fluxo de exceção. Os modificadores semânticos *only... can*, *can... only*, *cannot*, e *must* são utilizados para dar a semântica adequada para cada regra de design.

Se cada uma das palavras-chave que modificam a semântica introduzida por cada declaração de regra forem abstraídas (i.e., $\langle only - can \rangle$, $\langle can - only \rangle$, $\langle cannot \rangle$, e $\langle must \rangle$), todas as regras de design derivadas seguem a mesma estrutura sintática a qual inclui (i) uma parte fixa que compreende $\ldots \langle id \rangle \ldots \langle relation \rangle \ldots \langle id \rangle$; e (ii) uma parte opcional que

pode ser '`to`' $\langle id \rangle$ ou (',' $\langle id \rangle$)$^+$. Em ambas as partes, fixa e opcional, o identificador $\langle id \rangle$ pode se referir tanto a uma exceção como a um módulo. A escolha depende do tipo de relação de dependência $\langle relation \rangle$ levada em conta na especificação regra de design.

Quando uma das palavras-chave '`raise`', '`reraise`', '`signal`' ou '`handle`' é escolhida na especificação da regra de design, o primeiro identificador da parte fixa refere-se a um identificador de módulo e o segundo refere-se a um identificador de exceção. Nesse caso, a parte opcional não pode ser utilizada. No entanto, se a palavra-chave '`remap`' é escolhida, a derivação da parte fixa é semelhante à anterior, mas o segundo identificador representa o identificador da exceção a ser remapeada. Já na parte opcional '`to`' $\langle id \rangle$, o identificador se refere a um identificador de exceção que determina a exceção para qual se deseja remapear.

Finalmente, se a palavra-chave '`flow`' for escolhida, o primeiro identificador na parte fixa deve ser um identificador de exceção e o segundo deve ser um identificador de módulo, referente ao módulo que inicia o fluxo de controle excepcional. Já na parte opcional (',' $\langle id \rangle$)$^+$, cada identificador refere-se a um identificador de módulo. Esses identificadores definem uma lista de módulos pelos quais a exceção pode fluir, até ser finalmente tratada pelo último módulo.

4.3 ArCatch.Checker: A Semântica

Na abordagem proposta, as regras de design de tratamento de exceção são especificadas usando a ArCatch.Rules em termos de módulos, exceções e relações de dependência entre estes. ArCatch.Checker é responsável por: (i) estabelecer uma ligação entre os módulos e as exceções declaradas e suas respectivas classes de implementação; e (ii) verificar as regras de design escritas em ArCatch.Rules contra o código fonte do software. Nesta seção é descrita a semântica formal das regras de design do tratamento de exceção.

4.3.1 Definições Básicas

Nesta seção são apresentadas um conjunto de definições necessárias para descrever, de forma precisa, a semântica informal (Seção 4.3.2) e formal (Seção 4.3.3) das regras de design do tratamento de exceção.

Definição 1 (Classe de Implementação) *Uma classe de implementação é uma 3-tupla* $\langle n, t, \Phi \rangle$ *onde* n *é o nome qualificado da classe,* t *é o tipo da classe, e* Φ *é um conjunto de métodos da classe.*

Definição 2 (Funções de Acesso) *Seja* $c = \langle n, t, \Phi \rangle$ *uma classe de implementação, a função* `name`(c) *retorna o nome* n *da classe* c*, a função* `type`(c) *retorna o tipo* t *da classe*

c, e a função `classMethods`(c) *retorna o conjunto de métodos* Φ *da classe c.*

Definição 3 (Elemento Arquitetural) *Um elemento arquitetural* $A = \langle n, t, \phi \rangle$ *é uma 3-tupla onde n é o nome do elemento,* $t \in \{\texttt{M}, \texttt{E}\}$ *é o tipo do elemento, que pode ser do tipo módulo* (`M`) *ou tipo exceção* (`E`)*, e* ϕ *é a expressão regular usada para mapear as classes de implementação a partir do código-fonte.*

Definição 4 (Função `match`) *Seja* ϕ *uma expressão regular e C um conjunto de classes de implementação, a função* $\texttt{match}(\phi, C) = \{c \mid c \in C \wedge \texttt{name}(c) \in \omega(\phi)\}$ *retorna todas as classes em C cujos nomes casam com o padrão definido pela expressão regular* ϕ*. A função* $\omega(\phi)$ *significa todas as palavras reconhecidas pela expressão regular* ϕ*.*

Definição 5 (Função `map`) *Seja* $A = \langle n, t, \phi \rangle$ *um elemento arquitetural, C um conjunto de classes de implementação,* ξ *o tipo raiz de uma hierarquia de tipos de exceção, e* $<:$ *uma relação de subtipo onde C,* ξ*, e* $<:$ *são definidas em conformidade com as regras da linguagem de programação subjacente. A função* `map`(A, C)*, que executa o mapeamento entre um elemento arquitetural e suas classes de implementação, é definida como segue:*

$$\texttt{map}(A, C) = \begin{cases} t == \texttt{M},\ \{c \mid c \in \texttt{match}(\phi, C) \wedge \neg(\texttt{type}(c) <: \xi)\} \\ t == \texttt{E},\ \{c \mid c \in \texttt{match}(\phi, C) \wedge \texttt{type}(c) <: \xi\} \end{cases}$$

Definição 6 (Função `methods`) *Seja* $M = \langle n, \texttt{M}, \phi \rangle$ *um módulo e C um conjunto de classes de implementação, a função* $\texttt{methods}(M,C) = \{\ m\ \mid\ \forall c \in \texttt{map}(M,C),\ m \in \texttt{classMethods}(c)\}$ *retorna todos os métodos definidos em cada classe de implementação do mapeamento* `map`(M, C)*.*

Definição 7 (Função `call`) *Sejam m e n métodos arbitrários e C um conjunto de classes de implementação, a função* `call`(C, m, n) *retorna* `true` *se* $\exists\ (c, d \in C \wedge m \in \texttt{classMethods}(c) \wedge n \in \texttt{classMethods}(d))$ *tal que a relação "n chama m" se verifica. A função retorna* `false` *em qualquer outro caso.*

Definição 8 (Função `chains`) *Sejam* $M_1, \ldots, M_n$ *módulos e C um conjunto de classes de implementação, a função* $\texttt{chains}(C, M_1, \ldots, M_n) = \{(m_1, \ldots, m_n)\ \mid\ \forall i \in [1, n),\ m_i \in \texttt{methods}(M_i, C) \wedge m_{i+1} \in \texttt{methods}(M_{i+1}, C) \wedge \texttt{call}(C, m_i, m_{i+1})\}$ *retorna todas cadeias de chamadas de métodos de tamanho n, iniciadas em métodos de classes do módulo* M_1 *e terminadas em métodos de classes do módulo* M_n*.*

4.3.2 Semântica Informal

Antes de apresentar a semântica formal de ArCatch, nesta seção serão definidas de maneira informal todas as regras de design que podem ser construídas com a ArCatch.Rules.

Sejam C o conjunto das classes de um sistema, $M = \langle mid, \mathtt{M}, \Phi \rangle$ um módulo, $E = \langle eid, \mathtt{E}, \Phi \rangle$ e $F = \langle fid, \mathtt{E}, \Phi \rangle$ módulos de exceção e $M_1 = \langle mid_1, \mathtt{M}, \phi \rangle$,...,$M_n = \langle mid_n, \mathtt{M}, \phi \rangle$ uma lista de n módulos. A semântica informal de ArCatch.Rules é dada como segue:

- **Regras Cannot**: São regras de design do tratamento de exceções que restringem o uso de determinados módulos (normais ou de exceções) a não poder realizar determinadas ações relacionadas ao tratamento de exceção. ArCatch suporta definição das seguintes regras:
  - mid `cannot raise` eid: Nenhuma classe $c \in \mathtt{map}(M, C)$ pode levantar classes de exceção $e \in \mathtt{map}(E, C)$.
  - mid `cannot reraise` eid: Nenhuma classe $c \in \mathtt{map}(M, C)$ pode relevantar classes de exceção $e \in \mathtt{map}(E, C)$.
  - mid `cannot signal` eid: Nenhuma classe $c \in \mathtt{map}(M, C)$ pode sinalizar classes de exceção $e \in \mathtt{map}(E, C)$.
  - mid `cannot handle` eid: Nenhuma classe $c \in \mathtt{map}(M, C)$ pode tratar classes de exceção $e \in \mathtt{map}(E, C)$.
  - mid `remap remap` eid `to` fid: Nenhuma classe $c \in \mathtt{map}(M, C)$ pode remapear classes de exceção $e \in \mathtt{map}(E, C)$ para classes de exceção $f \in \mathtt{map}(F, C)$.
  - eid `cannot flow` mid_1, ..., mid_n: Nenhuma classe de exceção $e \in \mathtt{map}(E, C)$ pode ser sinalizada por métodos das classes $c_1 \in \mathtt{map}(M_1, C)$ e fluir através das classes $c_i \in \mathtt{map}(M_i, C)$ até ser tratada em classes $c_n \in \mathtt{map}(M_n, C)$.
- **Regras Must**: São regras de design do tratamento de exceções que restringem o uso de determinados módulos (normais ou de exceções) a dever, de forma obrigatória, realizar determinadas ações relacionadas ao tratamento de exceção. ArCatch suporta definição das seguintes regras:
  - mid `must raise` eid: Todos os métodos das classes $c \in \mathtt{map}(M, C)$ devem levantar classes de exceção $e \in \mathtt{map}(E, C)$.
  - mid `must reraise` eid: Todos os métodos das classes $c \in \mathtt{map}(M, C)$ devem relevantar classes de exceção $e \in \mathtt{map}(E, C)$.
  - mid `must signal` eid: Todos os métodos das classes $c \in \mathtt{map}(M, C)$ devem sinalizar classes de exceção $e \in \mathtt{map}(E, C)$.
  - mid `must handle` eid: Todos os métodos das classes $c \in \mathtt{map}(M, C)$ devem tratar classes de exceção $e \in \mathtt{map}(E, C)$.
  - mid `must remap` eid `to` fid: Todos os métodos das classes $c \in \mathtt{map}(M, C)$ devem remapear classes de exceção $e \in \mathtt{map}(E, C)$ para classes de exceção $f \in \mathtt{map}(F, C)$.

  - *eid* `must flow` mid_1, ..., mid_n: Todas as classes de exceção $e \in \texttt{map}(E, C)$ devem ser sinalizadas por métodos das classes $c_1 \in \texttt{map}(M_1, C)$ e fluir através das classes $c_i \in \texttt{map}(M_i, C)$ até ser tratada em classes $c_n \in \texttt{map}(M_n, C)$.

- **Regras Only-Can**: São regras de design do tratamento de exceções que restringem o uso de somente determinados módulos (normais ou de exceções) a poder realizar determinadas ações relacionadas ao tratamento de exceção. ArCatch suporta definição das seguintes regras:

  - `only` *mid* `can raise` *eid*: Somente as classes $c \in \texttt{map}(M, C)$ podem levantar classes de exceção $e \in \texttt{map}(E, C)$.
  - `only` *mid* `can reraise` *eid*: Somente as classes $c \in \texttt{map}(M, C)$ podem relevantar classes de exceção $e \in \texttt{map}(E, C)$.
  - `only` *mid* `can signal` *eid*: Somente as classes $c \in \texttt{map}(M, C)$ podem sinalizar classes de exceção $e \in \texttt{map}(E, C)$.
  - `only` *mid* `can handle` *eid*: Somente as classes $c \in \texttt{map}(M, C)$ podem tratar classes de exceção $e \in \texttt{map}(E, C)$.
  - `only` *mid* `can remap` *eid* `to` fid: Somente as classes $c \in \texttt{map}(M, C)$ podem remapear classes de exceção $e \in \texttt{map}(E, C)$ para classes de exceção $f \in \texttt{map}(F, C)$.
  - `only` *eid* `can flow` mid_1, ..., mid_n: Somente as classes de exceção $e \in \texttt{map}(E, C)$ podem ser sinalizadas por métodos das classes $c_1 \in \texttt{map}(M_1, C)$ e fluir através das classes $c_i \in \texttt{map}(M_i, C)$ até ser tratada em classes $c_n \in \texttt{map}(M_n, C)$.

- **Regras Can-Only**: São regras de design do tratamento de exceções que restringem o uso de determinados módulos (normais ou de exceções) a poder somente realizar determinadas ações relacionadas ao tratamento de exceção. ArCatch suporta definição das seguintes regras:

  - *mid* `can-only raise` *eid*: As classes $c \in \texttt{map}(M, C)$ podem somente levantar classes de exceção $e \in \texttt{map}(E, C)$.
  - *mid* `can-only reraise` *eid*: As classes $c \in \texttt{map}(M, C)$ podem somente relevantar classes de exceção $e \in \texttt{map}(E, C)$.
  - *mid* `can-only signal` *eid*: As classes $c \in \texttt{map}(M, C)$ podem somente sinalizar classes de exceção $e \in \texttt{map}(E, C)$.
  - *mid* `can-only handle` *eid*: As classes $c \in \texttt{map}(M, C)$ podem somente tratar classes de exceção $e \in \texttt{map}(E, C)$.
  - *mid* `can-only remap` *eid* `to` fid: As classes $c \in \texttt{map}(M, C)$ podem somente remapear classes de exceção $e \in \texttt{map}(E, C)$ para classes de exceção $f \in \texttt{map}(F, C)$.

– *eid* `can-only flow` mid_1, ..., mid_n: As classes de exceções $e \in \texttt{map}(E, C)$ podem somente ser sinalizadas por métodos das classes $c_1 \in \texttt{map}(M_1, C)$ e fluir através das classes $c_i \in \texttt{map}(M_i, C)$ até ser tratada em classes $c_n \in \texttt{map}(M_n, C)$.

Após definir todas as regras de design do tratamento de exceção de maneira informal, a seguir na Seção 4.3.3, será apresentada a semântica formal das violações de regras de design providas em ArCatch.

4.3.3 Semântica Formal da Violação de Regras de Design

A Tabela 3 apresenta um conjunto de relações de dependência ligadas ao tratamento de exceção em nível de código-fonte. Essas relações capturam a semântica gerada pelo emprego de construtos de linguagem (e.g., `try-catch-finally` da linguagem Java) na estruturação do código de tratamento de exceção.

Tabela 3 – Convenções de relações de dependência.

Relação	**Significado**
`raise`(m,e)	O método m levanta a classe de exceção do tipo e.
`reraise`(m,e)	O método m relança a classe de exceção do tipo e.
`signal`(m,e)	O método m sinaliza a classe de exceção do tipo e.
`handle`(m,e)	O método m trata a classe de exceção do tipo e.
`remap`(m,e,f)	O método m remapeia a classe de exceção do tipo e para a classe de exceção do tipo f.
`flow`(e,m_1,...,m_n)	A classe de exceção do tipo e é sinalizada pelo método m_1 e fluí através dos métodos m_2,...,m_{n-1} até ser tratada pelo método m_n.

Com base nas definições apresentadas na Seção 4.3.1 e nas relações descritas na Tabela 3, a semântica formal empregada na verificação da violação das regras de design do tratamento de exceção são definidas como segue:

SR1 (**Cannot**). Violação de regras escritas com o modificador semântico *cannot*:

Caso 1. Seja $E = \langle eid, \texttt{E}, \phi_E \rangle$ uma exceção, $M = \langle mid, \texttt{M}, \phi_M \rangle$ um módulo, $\oplus$ uma das relações em {`raise`, `reraise`, `signal`, `handle`}, e S um conjunto de classes de implementação. As regras do tipo "mid `cannot` $\oplus$ eid" são violadas se $\exists$ ($m \in \texttt{methods}(M,S) \ \wedge \ e \in \texttt{map}(E,S)$), tais que $\oplus(m,e)$.

Caso 2. Sejam $E = \langle eid, \texttt{E}, \phi_E \rangle$ e $F = \langle fid, \texttt{E}, \phi_F \rangle$ exceções, $M = \langle mid, \texttt{M}, \phi_M \rangle$ um módulo, e S um conjunto de classes de implementação. As regras do tipo "mid `cannot remap` eid `to` fid" são violadas se $\exists$ ($m \in \texttt{methods}(M,S) \ \wedge \ e \in \texttt{map}(E,S) \ \wedge \ f \in \texttt{map}(F,S)$), tais que `remap`($m$,$e$,$f$).

Caso 3. Seja $E = \langle eid, \texttt{E}, \phi_E \rangle$ uma exceção, $M_1 = \langle mid_1, \texttt{M}, \phi_{M_1} \rangle$,..., $M_n = \langle mid_n, \texttt{M}, \phi_{M_n} \rangle$ uma lista de n módulos, e S um conjunto de classes de implementação. As regras

do tipo "*eid* `cannot flow` $mid_1,\ldots,$ mid_n" são violadas se $\exists$ ($e \in$ `map`(E,S) $\wedge$ $(m_1,\ldots,m_n) \in$ `chains`$(S,M_1,\ldots,M_n)$), tais que `flow`$(e,m_1,\ldots,m_n)$.

SR2 (**Must**). Violação de regras escritas com o modificador semântico *must*:

Caso 1. Seja $E = \langle eid, \mathtt{E}, \phi\rangle$ uma exceção, $M = \langle mid, \mathtt{M}, \phi\rangle$ um módulo, $\oplus$ uma das relações em {`raise`, `reraise`, `signal`, `handle`}, e S um conjunto de classes de implementação. As regras do tipo "*mid* `must` $\oplus$ *eid*" são violadas se $\nexists$ ($m \in$ `methods`(M,S) $\wedge$ $e \in$ `map`(E,S)), tais que $\oplus(m,e)$.

Caso 2. Sejam $E = \langle eid, \mathtt{E}, \phi\rangle$ e $F = \langle fid, \mathtt{E}, \phi\rangle$ exceções, $M = \langle mid, \mathtt{M}, \phi\rangle$ um módulo, e S um conjunto de classes de implementação. As regras do tipo "*mid* `must remap` *eid* `to` *fid*" são violadas se $\nexists$ ($m \in$ `methods`(M,S) $\wedge$ $e \in$ `map`(E,S) $\wedge$ $f \in$ `map`(F,S)), tais que `remap`(m,e,f).

Caso 3. Seja $E = \langle eid, \mathtt{E}, \phi\rangle$ uma exceção, $M_1 = \langle mid_1, \mathtt{M}, \phi\rangle,\ldots,M_n = \langle mid_n, \mathtt{M}, \phi\rangle$ uma lista de n módulos, e S um conjunto de classes de implementação. As regras do tipo "*eid* `must flow` $mid_1,\ldots,$ mid_n" são violadas se $\nexists$ ($e \in$ `map`(E,S) $\wedge$ $(m_1,\ldots,m_n) \in$ `chains`$(S,M_1,\ldots,M_n)$), tais que `flows`$(e,m_1,\ldots,m_n)$.

SR3 (**Only-Can**). Violação de regras escritas com o modificador semântico *only-can*:

Caso 1. Seja $E = \langle eid, \mathtt{E}, \phi\rangle$ uma exceção, $M = \langle mid, \mathtt{M}, \phi\rangle$ um módulo, $\oplus$ uma das relações em {`raise`, `reraise`, `signal`, `handle`}, e S um conjunto de classes de implementação. As regras do tipo "`only` *mid* `can` $\oplus$ *eid*" são violadas se $\exists$ ($c \in S \setminus$ `map`(M,S) $\wedge$ $m \in$ `classMethods`(c) $\wedge$ $e \in$ `map`(E,S)), tais que $\oplus(m,e)$.

Caso 2. Sejam $E = \langle eid, \mathtt{E}, \phi\rangle$ e $F = \langle fid, \mathtt{E}, \phi\rangle$ exceções, $M = \langle mid, \mathtt{M}, \phi\rangle$ um módulo, e S um conjunto de classes de implementação. As regras do tipo "`only` *mid* `can remap` *eid* `to` *fid*" são violadas se $\exists$ ($c \in S \setminus$ `map`(M,S) $\wedge$ $m \in$ `classMethods`(c) $\wedge$ $e \in$ `map`(E,S) $\wedge$ $f \in$ `map`(F,S)), tais que `remap`(m,e,f).

Caso 3. Seja $E = \langle eid, \mathtt{E}, \phi\rangle$ uma exceção, $M_1 = \langle mid_1, \mathtt{M}, \phi\rangle,\ldots,M_n = \langle mid_n, \mathtt{M}, \phi\rangle$ uma lista de n módulos, e S um conjunto de classes de implementação. As regras do tipo "`oly` *eid* `can flow` $mid_1,\ldots,$ mid_n" são violadas se $\exists$ ($e \in S \setminus$ `map`(E,S) $\wedge$ $(m_1,\ldots,m_n) \in$ `chains`$(S,M_1,\ldots,M_n)$), tais que `flows`$(e,m_1,\ldots,m_n)$.

SR4 (**Can-Only**). Violação de regras escritas com o modificador semântico *can-only*:

1. Seja $E = \langle eid, \mathtt{E}, \phi\rangle$ uma exceção, $M = \langle mid, \mathtt{M}, \phi\rangle$ um módulo, $\oplus$ uma das relações em {`raise`, `reraise`, `signal`, `handle`}, e S um conjunto de classes de implementação. As regras do tipo "*mid* `can` $\oplus$ `only` *eid*" são violadas se $\exists$ ($m \in$ `methods`(M,S) $\wedge$ $e \in S \setminus$ `map`(E,S)), tais que $\oplus(m,e)$.

2. Sejam $E = \langle eid, \texttt{E}, \phi \rangle$ e $F = \langle fid, \texttt{E}, \phi \rangle$ exceções, $M = \langle mid, \texttt{M}, \phi \rangle$ um módulo, e S um conjunto de classes de implementação. As regras do tipo "*mid* `can remap only` *eid* `to` *fid*" são violadas se $\exists\ (m \in \texttt{methods}(M,S)\ \wedge\ ((e \in \texttt{map}(E,S)\ \wedge\ f \in S \setminus \texttt{map}(F,S))\ \vee\ (e \in S \setminus \texttt{map}(E,S)\ \wedge\ f \in \texttt{map}(F,S))\ \vee\ (e \in S \setminus \texttt{map}(E,S)\ \wedge\ f \in S \setminus \texttt{map}(F,S)))$, tais que $\texttt{remap}(m,e,f)$.
3. Seja $E = \langle eid, \texttt{E}, \phi \rangle$ uma exceção, $M_1 = \langle mid_1, \texttt{M}, \phi \rangle, \ldots, M_n = \langle mid_n, \texttt{M}, \phi \rangle$ uma lista de n módulos, e S um conjunto de classes de implmentação. As regras do tipo "*eid* `can flow oly` $mid_1, \ldots, mid_n$" são violadas se $\exists\ (e \in \texttt{map}(E,S)\ \wedge\ (m_1, \ldots, m_k) \notin \texttt{chains}(S, M_1, \ldots, M_n))$, tais que $\texttt{flows}(e, m_1, \ldots, m_k)$ com $k > 1$.

4.4 Detalhes de Implementação

A ArCatch foi implementada em Java e sua versão atual fornece suporte à verificação de conformidade do tratamento de exceção de programas Java. O código fonte de ArCatch está disponível na internet no serviço de hospedagem de repositórios de software GitHub[1]. Em ArCatch.Checker, todas as informações de código fonte relevantes para o processo de verificação são extraídas usando a ferramenta Design Wizard[2] e a *Java Compiler Tree* API[3]. O Design Wizard fornece meios para extrair as dependências de classe do programa, como as árvores de herança de classe e grafos de chamadas de métodos para alimentar o algoritmo que verifica as regras de design providas em ArCatch. Já a *Compiler Tree API* fornece suporte para inspecionar a AST (*Abstract Syntax Tree*) de programas Java, ajudando na identificação dos casos de levantamento, relevantamento e remapeamento que ocorrem no código fonte.

A ArCatch.Rules foi implementada em Java como uma DSL interna (a.k.a, *fluent* API) (FOWLER, 2010). Isso lhe confere uma facilidade de ser incorporada a um ambiente de integração contínua via o conceito de testes de design (BRUNET; GUERRERO; FIGUEIREDO, 2009). Teste de design são scripts de teste que verificam automaticamente se uma implementação está em conformidade com uma regra de design específica. Nesse caso, as regras de design são implementadas diretamente na linguagem de programação alvo utilizando alguma ferramenta de automação de teste (e.g., JUnit[4]).

ArCatch necessita de apenas 5 etapas para que possa ser utilizada corretamente. A primeira delas é a etapa de configuração, necessária para que ArCatch localize o sistema de software a ser anualizado. Para isso, basta o usuário informar o caminho para o código fonte e para o código binário (`.jar`) do software em questão (Listagem 4.1).

[1] https://github.com/lincolnrocha/ArCatch
[2] https://github.com/joaoarthurbm/designwizard
[3] https://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/
[4] http://junit.org/

Listagem 4.1 – Exemplo de Configuração do Código Fonte Alvo.

```
1 ArCatch.config("source_code_path", "binary_code_(.jar)_path");
```

Na segunda etapa, o usuário declara todos os módulos do sistema que deseja analisar. Para isso, ele nomeia cada módulo e fornece a expressão regular de mapeamento entre os módulos declarados e as suas classes de implementação, seguindo a notação descrita na Listagem 4.2.

Listagem 4.2 – Exemplo de Declaração dos Módulos.

```
1 ModuleElement view = ArCatch.element().module("View").matching("banksys.view.\\w
      +").build();
2
3 ModuleElement control = ArCatch.element().module("Control").matching("banksys.
      control.\\w+").build();
4
5 ModuleElement model = ArCatch.element().module("Model").matching("banksys.model
      .\\w+").build();
```

A terceira etapa é semelhante a etapa anterior, porém nesta são declarados os módulos de exceção do sistema. Nesse caso, o usuário precisa mapear as classes de exceções para módulos de exceção em nível arquitetural, como exemplificado na Listagem 4.3.

Listagem 4.3 – Exemplo de Declaração das Exceções.

```
1 ExceptionElement modelEx = ArCatch.element().exception("ControlEx").matching("
      banksys.control.exception.\\w+").build();
2
3 ExceptionElement controlEx = ArCatch.element().exception("ModelEx").matching("
      banksys.model.exception.\\w+").build();
```

A quarta etapa consiste na especificação das regras de design que caracterizam a política de tratamento de exceção (i.e., o design do tratamento de exceção) adotada para o sistema alvo. Nesta etapa, o usuário faz uso da linguagem ArCatch.Rules para definir todas as restrições as quais o sistema deve atender, como exemplificado na Listagem 4.4.

Listagem 4.4 – Exemplo de Especificação de Regras de Design.

```
1  DesignRule r1 = ArCatch.rule().only(model).canRaise(modelEx).build();
2
3  DesignRule r2 = ArCatch.rule().only(model).canSignal(modelEx).build();
4
5  DesignRule r3 = ArCatch.rule().module(control).mustHandle(modelEx).build();
6
7  DesignRule r4 = ArCatch.rule().module(control).canOnlySignal(controlEx).build();
8
9  DesignRule r5 = ArCatch.rule().exception(modelEx).cannotFlow(model, control,
       view).build();
10
11 DesignRule r6 = ArCatch.rule().only(control).canRemap(modelEx).to(controlEx).
       build();
12
13 DesignRule r7 = ArCatch.rule().module(view).mustHandle(controlEx).build();
14
15 DesignRule r8 = ArCatch.rule().module(view).cannotHandle(modelEx).build();
```

A quinta e última etapa é responsável por realizar a verificação das regras de design definidas na etapa anterior. Nesse caso, pode-se realizar de duas formas: (i) acionando o método `ArCatch.checker().addRule(rule)` para adicionar as regras que devem ser verificadas, seguido do comando `ArCatch.checker().checkAll()` para finalmente realizar a verificação da conformidade; e (ii) acionando o método `ArCatch.checker().check(rule)` que faz a verificação de cada regra de modo individual, retornando verdadeiro se o código está em conformidade com a regra ou falso em caso contrário. As listagens 4.5 e 4.6, exemplificam, respectivamente, as duas maneiras de se realizar a verificação das regras de design com a ArCatch.Checker.

Listagem 4.5 – Exemplo de Verificação das Regras (Estratégia I).

```
1  ArCatch.checker().addRule(r1);
2  ...
3  ArCatch.checker().addRule(r8);
4
5  ArCatch.checker().checkAll();
```

Listagem 4.6 – Exemplo de Verificação das Regras (Estratégia II).

```
1  boolean resultR1 = ArCatch.checker().check(r1);
2  ...
3  boolean resultR8 = ArCatch.checker().check(r8);
```

ArCatch.Checker fornece um relatório de conformidade contendo informações úteis sobre quais regras de design foram violadas e onde tais violações ocorrem no código fonte do software em análise. Para isso, se for utilizado a estratégia (i) `ArCatch.checker().checkAll()`, um relatório da verificação de conformidade em formato textual é gerado na pasta `./report`. No entanto, se utilizado a estratégia (ii) `ArCatch.checker().check(rule)`, as informações sobre a violação da regra podem ser obtidas por meio do método `DesignRule.getReport()`

de cada regra verificada. A seguir, a Listagem 4.7 mostra um exemplo de relatório de verificação da conformidade gerado por ArCatch.Checker.

Listagem 4.7 – Exemplo de Relatório de Conformidade Arquitetural.

```
1  ================================================================================
2  ArCatch.Checker Exception Handling Conformance Checking Report
3  --------------------------------------------------------------------------------
4  Label: (V) = Rule Pass | (X) = Rule Fail
5  ================================================================================
6  ...
7  --------------------------------------------------------------------------------
8  (X) R2: only (Model) can signal (ModelEx) 8 ms
9
10  -Model module implementation classes:
11   -banksys.model.AbstractAccount
12   -banksys.model.OrdinaryAccount
13   -banksys.model.SavingsAccount
14   -banksys.model.SpecialAccount
15   -banksys.model.TaxAccount
16
17  -ModelEx exception implementation classes:
18   -banksys.model.exception.InsufficientFundsException
19   -banksys.model.exception.NegativeAmountException
20
21  -Rule Violations
22     -Method [banksys.control.BankController.doDebit(java.lang.String, double)]
          is signaling the exception [lib.exceptions.InsufficientFundsException]
23     -Method [banksys.control.BankController.doDebit(java.lang.String, double)]
          is signaling the exception [lib.exceptions.NegativeAmountException]
24 --------------------------------------------------------------------------------

25 ...
```

4.5 Considerações Finais

Este capítulo apresentou ArCatch, uma solução de verificação estática de conformidade arquitetural que tem com objetivo combater a erosão do tratamento de exceção. Detalhes sobre a sintaxe (ArCatch.Rules) e a semântica (ArCatch.Checker) de ArCatch foram apresentados, respectivamente, nas seções 4.2 e 4.3. Em seguida, na Seção 4.4, detalhes técnicos de implementação e uso de ArCatch foram descritos. No próximo capítulo uma avaliação de ArCatch é apresentada como forma de demonstrar a sua viabilidade.

5 Avaliação

Este capítulo é dedicado a avaliação de ArCatch. Na Seção 5.1 é feita uma introdução ao procedimento utilizado na avaliação de ArCatch. Em seguida, a Seção 5.2 descreve o sistema adotado na avaliação e seus cenários de evolução. Na Seção 5.3, é descrita toda a especificação de design do tratamento de exceção para o sistema adotado. A Seção 5.4 apresenta detalhes sobre como ajustes nos mapeamentos da especificação foram feitos em cada versão do sistema adotado para acomodar os efeitos das ações de manutenção e evolução. A Seção 5.5 é dedicada a apresentação e discussão dos resultados da avaliação. Por fim, na Seção 5.6 são realizadas as considerações finais deste capítulo.

5.1 Introdução

A avaliação de ArCatch foi conduzida com o intuito de evidenciar a sua viabilidade em termos de (i) expressividade em definir regras de design para o tratamento de exceção; e (ii) eficácia na identificação de problemas de erosão do tratamento de exceção durante atividades regulares manutenção e evolução[1] em sistemas de software. O primeiro objetivo (i) corresponde o nível de detalhe com o qual a linguagem ArCatch.Rules consegue expressar os elementos arquiteturais (módulos e exceções), os seus relacionamos e fazer o mapeamento desses para o código fonte do sistema estudado, tornando explícito o design do tratamento de exceção. Por outro lado, o segundo objetivo (ii) corresponde o nível de detalhe com o qual o verificador ArCatch.Checker é capaz de detectar e localizar no código fonte do sistema analisado problemas de erosão do tratamento de exceção após a realização de atividades de manutenção e evolução. Para isso, ArCatch foi submetida a um cenário de avaliação que consiste na análise de conformidade de 10 versões de um sistema existente contra um conjunto de regras de design de tratamento de exceções escritas em ArCatch. Essas regras de design buscam garantir o atendimento a uma política de tratamento de exceção definida para o sistema em análise.

5.2 Sistema Adotado e Cenários de Mudança

O sistema alvo adotado nesta avaliação foi o Health Watcher (HW), um sistema desenvolvido com o objetivo de melhorar a qualidade dos serviços prestados por instituições

[1] No escopo deste trabalho, são consideradas como atividades de manutenção aquelas que envolvem mudanças no software que não adicionam (ou removem) funcionalidades e/ou características (*features*). Por outro lado, são consideradas atividades de evolução aquelas que resultam em adição ou remoção de funcionalidades e/ou características (*features*).

de saúde (SOARES; LAUREANO; BORBA, 2002). HW é um sistema web desenvolvido em Java que permite aos cidadãos apresentar queixas relativas a questões de saúde. De posse das queixas, as instituições oficiais de saúde podem conduzir investigações e diligencias com o propósito de descobrir a veracidade das queixas e tomar as medidas necessárias. HW foi escolhido porque possui um design de tratamento de exceção bem definido (intencional) e tem sido usado em vários estudos empíricos sobre modularidade de software e tratamento de exceção (FERRARI et al., 2010; GURGEL et al., 2014; OIZUMI et al., 2015). HW segue um estilo arquitetural multicamadas (Figura 6) composto por 4 camadas: *View* (`ViL`), *Distribution* (`DiL`), *Business* (`BuL`) e *Data* (`DaL`).

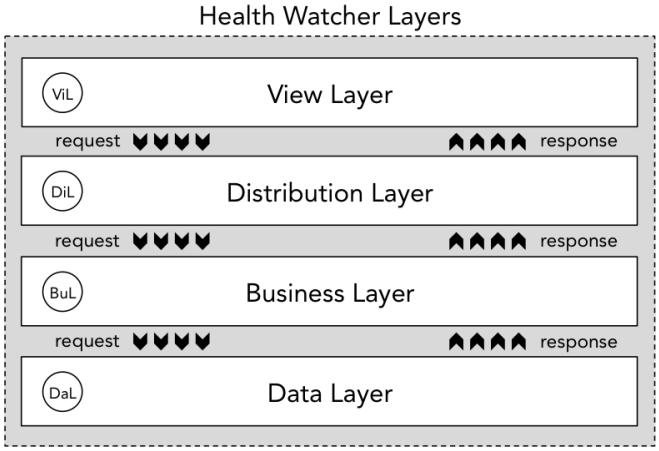


Figura 6 – Arquitetura multicamadas de Health Watcher.

Todas as 10 versões de HW utilizadas nesta avaliação, incluindo a versão base v1, estão disponíveis na Internet[2] e caracterizadas na Tabela 4.

Tabela 4 – Resumo do conjunto de dados.

Versão	**Linhas de Código**	**Número de Classes**	**Número de Pacotes**
v1	7070	80	19
v2	7498	92	20
v3	7989	104	21
v4	8124	106	22
v5	8344	108	22
v6	8422	112	23
v7	8521	116	23
v8	8577	120	24
v9	10054	132	24
v10	9938	136	25

O conjunto de versões analisadas compreende um número de tarefas de manutenção típicos (e.g., refatorações e extensões de módulos abstratos) e mais complexas de evolução do sistema (e.g., incrementos de funcionalidades) que estão todas descritas em (GREENWOOD et al., 2007) e resumidas na Tabela 5.

[2] http://ptolemy.cs.iastate.edu/design-study/#healthwatcher

Tabela 5 – Resumo das mudanças sofridas em cada versão de Health Watcher (Adaptado de (GREENWOOD et al., 2007)).

Versão	Mudança
v2	Refatorar múltiplos Servlets para melhorar a extensibilidade.
v3	Assegurar que o estado queixa não possa ser atualizado uma vez fechado, afim de proteger as queixas de múltiplas atualizações.
v4	Encapsular operações de atualização para melhorar a sustentabilidade usando práticas comuns de engenharia de software.
v5	Melhorar o encapsulamento de interesses da distribuição para melhorar o reuso e a personalização.
v6	Generalizar o mecanismo de persistência para melhorar o reuso e extensibilidade.
v7	Remover as dependências nos objetos de requisição e resposta do Servlet para facilitar o processo de adicionar novas interfaces gráficas.
v8	Generalizar o mecanismo de distribuição para melhorar a reuso e extensibilidade.
v9	Novas funcionalidades adicionais para dar suporte a consulta de mais tipos de dados.
v10	Modularizar o tratamento de exceção e incluir um comportamento de recuperação de erros mais eficaz para os tratadores.

5.3 Especificação do Design do Tratamento de Exceção

Cada camada da arquitetura HW foi representada como um módulo e mapeados para as correspondentes classes de implementação em nível de código-fonte. A Listagem 5.1 mostra esse mapeamento[3] realizado para a versão v1 de HW. A camada `ViL` foi mapeada para todas as classes do pacote "`healthwatcher.view.servlets`" (Listagem 5.1, linha 2). A camada `DiL` foi mapeada para todas as classes do pacote "`lib.distribution.rmi`" e as classes `IFacade`, `HealthWatcherFacade` e `HealthWatcherFacadeInit` (Listagem 5.1, linha 4). A camada `BuL` foi mapeada para as classes dos subpacotes de "`healthwatcher.business`" (Listagem 5.1, linha 6). Finalmente, a camada `DaL` foi mapeada para todas as classes de pacotes e subpacotes de "`healthwatcher.data`" e "`lib.persistence`" (Listagem 5.1, linha 8).

Listagem 5.1 – Mapeamento das Camadas de Health Watcher.

```
1 ModuleElement viL, diL, buL, daL;
2 viL = ArCatch.element().module("ViL").matching("healthwatcher.view.servlets.\\w*
      ").build();
3
4 diL = ArCatch.element().module("DiL").matching("(lib.distribution.rmi.\\w*|
      healthwatcher.view.IFacade|healthwatcher.business.(HealthWatcherFacade|
      HealthWatcherFacadeInit))*").build();
5
6 buL = ArCatch.element().module("BuL").matching("healthwatcher.business.(
      complaint|employee|healthguide).\\w*").build();
7
8 daL = ArCatch.element().module("DaL").matching("(healthwatcher.data|lib.
      persistence).(\\w*.)*\\w*").build();
```

[3] O símbolo "\w" representa um caractere da palavra: `[a-zA-Z_0-9]`.

Todas as exceções definidas na versão v1 de HW estão no pacote `lib.exceptions`. Com base na documentação de HW e na análise de código fonte, seis grupos de exceções foram definidos e mapeados como mostra a Listagem 5.2. A exceção `DiLEx` representa as exceções definidas pelo usuário (i.e., definida pelo programador) relacionadas com a camada `DiL`, as exceções `BuLEx` estão relacionadas com a camada `BuL`, e as exceções `DaLEx` estão relacionadas com a camada `DaL`. As `SVTEx` e `SQLEx` são exceções definidas pela plataforma e `AllEx` representa todas as exceções definidas pelo usuário.

Listagem 5.2 – Mapeamento das Exceções.

```
1  ExceptionElement diLEx, buLEx, daLEx, sqlEx, svtEx, allEx;
2  diLEx = ArCatch.element().exception("DiLEx").matching("(java.rmi.RemoteException
       |lib.exceptions.CommunicationException)*").build();
3
4  buLEx = ArCatch.element().exception("BuLEx").matching("lib.exceptions.(
       ObjectAlready)\\w*").build();
5
6  daLEx = ArCatch.element().exception("DaLEx").matching("lib.exceptions.(
       Persistence|ObjectNot|Repository|Transaction)\\w*").build();
7
8  sqlEx = ArCatch.element().exception("SQLEx").matching("java.sql.SQLException").
       build();
9
10 svtEx = ArCatch.element().exception("SVTEx").matching("javax.servlet.
       ServletException").build();
11
12 allEx = ArCatch.element().exception("AllEx").matching("lib.exceptions.(\\w*.)*\\
       w*").build();
```

Observe que, uma vez que o sistema evolui, os mapeamentos das classes nas camadas também mudam. Assim, para cada versão de HW, foi necessário executar alguns refinamentos ao mapeamento, afim de capturar as mudanças ocorridas de uma versão para outra.

Foi definida uma política de tratamento de exceção para avaliar o design do tratamento de exceção de HW com base na intenção do arquiteto de software de HW (coletado através de uma entrevista não estruturada) e nas boas práticas recomendadas pela BluePrints Design Patterns da Oracle[4] para arquiteturas multicamadas de sistemas JAVA. Na política estabelecida, uma exceção pode ser levantada e/ou sinalizada por uma camada qualquer. Quando uma camada específica (chamada) sinaliza uma exceção, essa exceção pode somente ser propagada para a camada imediatamente superior (chamadora), a qual é responsável por capturar a exceção e realizar ações de tratamento (estratégia *catch-and-handle*), colocando o sistema de volta em seu fluxo de controle normal. Se esta exceção não pode ser tratada naquele escopo, a camada chamadora deve executar um remapeamento de tipo de exceção e sinalizar um novo tipo de exceção para a próxima camada superior (estratégia *catch-and-remap*). Esse processo se repete até que a situação

[4] http://www.oracle.com/technetwork/java/patterns-139816.html

excepcional seja finalmente tratada em alguma camada superior. Exceções sinalizadas por componentes de terceiros (e.g., bibliotecas externas ou APIs da linguagem de programação) para uma camada específica, devem ser tratadas nesta mesma camada (*catch-and-handle*) ou serem remapeadas e sinalizadas para a próxima camada superior (*catch-and-remap*).

Após o processo de mapeamento, um conjunto de regras de design são definidas (Tabela 6) para fazer cumprir a política estabelecida. Cada regra de design reforça um aspecto específico da política de tratamento de exceção. Por exemplo, a regra R01 impõe que as exceções sinalizadas pela camada inferior `DiL` devam ser tratadas pela camada superior `ViL` (estratégia *catch-and-handle*). As regras R06 e R10 têm um propósito similar. As regras R07 e R11 garantem que a estratégia *catch-and-remap* é usada. As regras R14 e R15 impõem que exceções `SQLEx` sinalizadas por componentes terceiros devem ser tratadas pela camada `DaL`. A regra R03 impõe que nenhuma exceção definida pelo usuário pode ser sinalizada pela camada `ViL`. Finalmente, a R16 impõe que exceções `DaLEx` não podem fluir através dos módulos `DaL`, `BuL` e `DiL`.

Tabela 6 – Regras de design do tratamento de exceção para Health Watcher.

ID	Regra
R01	ArCatch.rule().module(viL).mustHandle(diLEx).build()
R02	ArCatch.rule().only(viL).canSignal(svtEx).build()
R03	ArCatch.rule().module(viL).cannotSignal(allEx).build()
R04	ArCatch.rule().only(diL).canRaise(diLEx).build()
R05	ArCatch.rule().only(diL).canSignal(diLEx).build()
R06	ArCatch.rule().module(diL).mustHandle(buLEx).build()
R07	ArCatch.rule().only(diL).canRemap(buLEx).to(diLEx).build()
R08	ArCatch.rule().only(buL).canRaise(buLEx).build()
R09	ArCatch.rule().only(buL).canSignal(buLEx).build()
R10	ArCatch.rule().module(buL).mustHandle(daLEx).build()
R11	ArCatch.rule().only(buL).canRemap(daLEx).to(buLEx).build()
R12	ArCatch.rule().only(daL).canRaise(daLEx).build()
R13	ArCatch.rule().only(daL).canSignal(daLEx).build()
R14	ArCatch.rule().only(daL).canHandle(sqlEx).build()
R15	ArCatch.rule().module(daL).cannotSignal(sqlEx).build()
R16	ArCatch.rule().exception(daLEx).cannotFlow(daL, buL, diL).build()

5.4 Ajustes na Especificação ao Longo das Versões

Esta seção apresenta detalhes sobre cada uma das versões de HW utilizadas na avaliação, a fim de explorar o conteúdo de cada elemento arquitetural presente na especificação descrita na Seção 5.3, suas mudanças de versão para versão e o impacto das

mudanças nos mapeamentos feitos entre os elementos arquiteturais (módulos e exceções) e as classes de implementação.

5.4.1 Versão 1

A versão 1 de HW é a versão base do conjunto de versões de HW considerados na avaliação. A Figura 7 mostra todos os pacotes presentes na primeira versão de HW. Em seguida, com base nos mapeamentos realizados usando ArCatch.Rules, apresentados na Seção 5.3, é mostrado (listagens 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11 e 5.12) como cada uma das entidades de implementação da versão 1 de HW ficaram mapeadas nas entidades arquiteturais definidas. Nas próximas subseções, são detalhadas cada uma das mudanças ocorridas em HW e seu impacto direto nos mapeamentos realizados para cada uma das versões utilizadas.

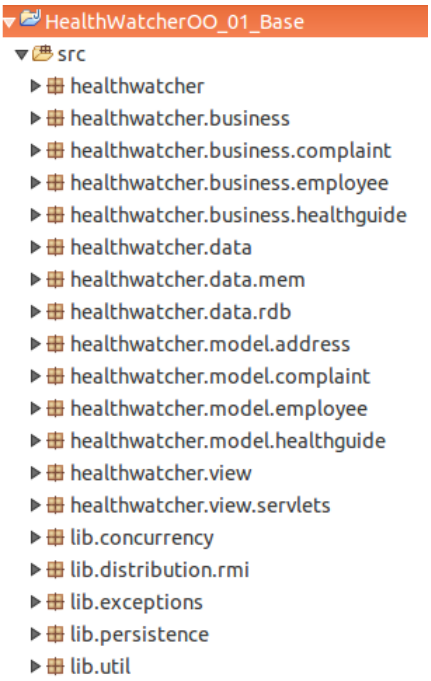


Figura 7 – Visão de pacotes da versão 1 (base) de Health Watcher.

Listagem 5.3 – Classes de Implementação Mapeadas para o Módulo ViL.

```
-ViL module implementation classes:
  -healthwatcher.view.servlets.ServletGetDataForSearchBySpeciality
  -healthwatcher.view.servlets.ServletUpdateHealthUnitData
  -healthwatcher.view.servlets.ServletInsertEmployee
  -healthwatcher.view.servlets.ServletUpdateComplaintData
  -healthwatcher.view.servlets.ServletSearchComplaintData
  -healthwatcher.view.servlets.ServletWebServer
  -healthwatcher.view.servlets.ServletInsertAnimalComplaint
  -healthwatcher.view.servlets.ServletSearchHealthUnitsBySpecialty
  -healthwatcher.view.servlets.ServletUpdateComplaintSearch
  -healthwatcher.view.servlets.ServletInsertSpecialComplaint
  -healthwatcher.view.servlets.ServletSearchDiseaseData
  -healthwatcher.view.servlets.ServletUpdateEmployeeData
  -healthwatcher.view.servlets.ServletConfigRMI
  -healthwatcher.view.servlets.ServletGetDataForSearchByDiseaseType
  -healthwatcher.view.servlets.ServletInsertFoodComplaint
  -healthwatcher.view.servlets.HWServlet
  -healthwatcher.view.servlets.ServletUpdateHealthUnitSearch
  -healthwatcher.view.servlets.ServletSearchSpecialtiesByHealthUnit
  -healthwatcher.view.servlets.ServletGetDataForSearchByHealthUnit
  -healthwatcher.view.servlets.ServletLogin
  -healthwatcher.view.servlets.ServletUpdateEmployeeSearch
```

Listagem 5.4 – Classes de Implementação Mapeadas para o Módulo DiL.

```
-DiL module implementation classes:
  -healthwatcher.business.HealthWatcherFacadeInit
  -healthwatcher.view.IFacade
  -lib.distribution.rmi.IteratorRMISourceAdapter
  -lib.distribution.rmi.IIteratorRMITargetAdapter
  -healthwatcher.business.HealthWatcherFacade
  -lib.distribution.rmi.IteratorRMITargetAdapter
```

Listagem 5.5 – Classes de Implementação Mapeadas para o Módulo BuL.

```
-BuL module implementation classes:
  -healthwatcher.business.complaint.ComplaintRecord
  -healthwatcher.business.healthguide.HealthUnitRecord
  -healthwatcher.business.healthguide.MedicalSpecialityRecord
  -healthwatcher.business.complaint.DiseaseRecord
  -healthwatcher.business.employee.EmployeeRecord
```

Listagem 5.6 – Classes de Implementação Mapeadas para o Módulo DaL.

```
-DaL module implementation classes:
  -healthwatcher.data.mem.ComplaintRepositoryArray
  -healthwatcher.data.mem.SpecialityRepositoryArray
  -healthwatcher.data.IDiseaseRepository
  -healthwatcher.data.IComplaintRepository
  -healthwatcher.data.rdb.ComplaintRepositoryRDB
  -healthwatcher.data.rdb.AddressRepositoryRDB
  -healthwatcher.data.rdb.HealthUnitRepositoryRDB
  -healthwatcher.data.IHealthUnitRepository
  -healthwatcher.data.ISpecialityRepository
  -lib.persistence.PersistenceMechanism
  -healthwatcher.data.rdb.EmployeeRepositoryRDB
  -lib.persistence.IPersistenceMechanism
  -healthwatcher.data.mem.SymptomRepositoryArray
  -healthwatcher.data.rdb.SpecialityRepositoryRDB
  -healthwatcher.data.mem.EmployeeRepositoryArray
  -healthwatcher.data.IEmployeeRepository
  -healthwatcher.data.mem.HealthUnitRepositoryArray
  -healthwatcher.data.mem.DiseaseTypeRepositoryArray
  -healthwatcher.data.IAddressRepository
  -healthwatcher.data.ISymptomRepository
  -healthwatcher.data.rdb.DiseaseTypeRepositoryRDB
```

Listagem 5.7 – Classes de Exceção Mapeadas para o Módulo BuLEx.

```
-BuLEx exception implementation classes:
  -lib.exceptions.ObjectAlreadyInsertedException
```

Listagem 5.8 – Classes de Exceção Mapeadas para o Módulo DiLEx

```
-DiLEx exception implementation classes:
  -lib.exceptions.CommunicationException
  -java.rmi.RemoteException
```

Listagem 5.9 – Classes de Exceção Mapeadas para o Módulo DaLEx.

```
-DaLEx exception implementation classes:
  -lib.exceptions.ObjectNotFoundException
  -lib.exceptions.RepositoryException
  -lib.exceptions.PersistenceSoftException
  -lib.exceptions.PersistenceMechanismException
  -lib.exceptions.ObjectNotValidException
  -lib.exceptions.TransactionException
```

Listagem 5.10 – Classes de Exceção Mapeadas para o Módulo SVTEx.

```
-SVTEx exception implementation classes:
  -javax.servlet.ServletException
```

Listagem 5.11 – Classes de Exceção Mapeadas para o Módulo SQLEx.

```
-SQLEx exception implementation classes:
  -java.sql.SQLException
```

Listagem 5.12 – Classes de Exceção Mapeadas para o Módulo AllEx.

```
-AllEx exception implementation classes:
  -lib.exceptions.ObjectAlreadyInsertedException
  -lib.exceptions.RepositoryException
  -lib.exceptions.InvalidSessionException
  -lib.exceptions.ObjectNotValidException
  -lib.exceptions.InvalidDateException
  -lib.exceptions.InsertEntryException
  -lib.exceptions.UpdateEntryException
  -lib.exceptions.CommunicationException
  -lib.exceptions.ObjectNotFoundException
  -lib.exceptions.PersistenceSoftException
  -lib.exceptions.PersistenceMechanismException
  -lib.exceptions.SituationFacadeException
  -lib.exceptions.TransactionException
```

5.4.2 Versão 2

Na versão 2 de HW, o conjunto de fatorações realizadas alteraram a estrutura do código (principalmente das classes *servlets*) para incorporar a implementação do padrão de projeto *Command*. Essa mudança fez surgir o pacote `healthwatcher.view.command` na estrutura de HW. Entretanto, os demais pacotes listados na Figura 7 foram mantidos. Essa mudança tem impacto direto na camada *View* representada pelo módulo `ViL`. Na versão 2, o módulo `ViL` passa a compreender as classe de implementação mostrados na Listagem 5.13.

Após as alterações, o pacote `healthwatcher.view` foi alterado, restando apenas duas classes no pacote `healthwatcher.view.servlets`. As demais classes advindas da versão 1 migraram para o novo pacote `healthwatcher.view.command` e foram refatoradas para se adequarem ao padrão *Command*. Dessa forma, foram desacoplados os objetos que invocam as operações, daqueles que sabem como executá-las. Para mapear as classes do novo pacote para o módulo `ViL`, uma pequena atualização no mapeamento foi realizada, como mostra a Listagem 5.14. Dessa forma, todas as classes presentes em `healthwatcher.view.servlets` e `healthwatcher.view.command` foram mapeadas para o módulo `ViL`.

Listagem 5.13 – Classes Mapeadas para o Módulo ViL na Versão 02 de HW.

```
-ViL module implementation classes:
  -healthwatcher.view.command.UpdateHealthUnitSearch
  -healthwatcher.view.command.UpdateComplaintList
  -healthwatcher.view.command.ConfigRMI
  -healthwatcher.view.command.InsertSpecialComplaint
  -healthwatcher.view.command.InsertAnimalComplaint
  -healthwatcher.view.command.LoginMenu
  -healthwatcher.view.command.SearchHealthUnitsBySpecialty
  -healthwatcher.view.command.UpdateHealthUnitData
  -healthwatcher.view.command.UpdateEmployeeData
  -healthwatcher.view.command.SearchComplaintData
  -healthwatcher.view.servlets.HWServlet
  -healthwatcher.view.command.Login
  -healthwatcher.view.command.GetDataForSearchByDiseaseType
  -healthwatcher.view.command.UpdateHealthUnitList
  -healthwatcher.view.servlets.ServletWebServer
  -healthwatcher.view.command.Command
  -healthwatcher.view.command.InsertFoodComplaint
  -healthwatcher.view.command.UpdateEmployeeSearch
  -healthwatcher.view.command.UpdateComplaintData
  -healthwatcher.view.command.UpdateComplaintSearch
  -healthwatcher.view.command.InsertEmployee
  -healthwatcher.view.command.GetDataForSearchByHealthUnit
  -healthwatcher.view.command.GetDataForSearchBySpeciality
  -healthwatcher.view.command.SearchDiseaseData
  -healthwatcher.view.command.SearchSpecialtiesByHealthUnit
```

Listagem 5.14 – Mapeamento para o Módulo ViL na Versão 02 de HW.

```
viL = ArCatch.element().module("ViL").matching("healthwatcher.view.(servlets|
    command).[a-zA-Z_$][a-zA-Z0-9_$]*").build();
```

5.4.3 Versão 3

Na versão 3 de HW as mudanças realizadas buscaram assegurar que o estado de uma queixa não pudesse ser atualizado após ter assumido o estado de fechado. Essas mudanças foram feitas através da implementação do padrão de projeto *State*, que culminaram com a adição de mais um pacote ao sistema (`healthwatcher.model.complaint.state`), porém, mantendo-se os demais pacotes presentes na versão anterior. Essa mudança tem impacto direto na camada *View* e *Business* representadas respectivamente pelos módulos `ViL` e `BuL`. Entretanto, essas alterações não trazem nenhuma necessidade de mudança nos mapeamentos definidos na especificação de design do tratamento de exceção.

5.4.4 Versão 4

Na versão 4 de HW as mudanças realizadas foram feitas por meio da implementação do padrão de projeto *Observer*. Essa mudança é refletida no adição de um novo pacote ao sistema (`lib.patterns.observer`), mantendo-se os demais pacotes presentes na versão anterior. Essa mudança teve impacto direto na camada *View* e *Business* representadas respectivamente por `ViL` e `BuL`. Entretanto, não houve a necessidade de alterações nos mapeamentos da especificação. O padrão *Observer* foi implementado para melhorar a forma como as mudanças de estado em determinados objetos são propagadas para objetos dependentes dessas mudanças. Dessa forma, o acoplamento entre esses objetos é reduzido e o seu encapsulamento é aumentado, uma vez que a interação entre objetos observadores e observados ocorre sem a necessidade dos primeiros conhecerem a estrutura interna dos segundos.

5.4.5 Versão 5

Na versão 5 de HW as mudanças realizadas tiveram como objetivo melhorar o encapsulamento do interesse (*concern*) de distribuição. Para isso, o padrão de projeto *Adapter* foi implementado no sistema. Essa mudança não resultou no acréscimo de novos pacotes ao sistema como em versões anteriores, mas consistiu na implementação de novas interfaces *Adapter* nos pacotes `healthwatcher.view` e `healthwatcher.business`. Essa mudança teve impacto direto nas camadas *View*, *Distribution* e *Business* representadas respectivamente por `ViL`, `DiL` e `BuL`. Nesta versão, não ocorrem alterações nos mapeamentos definidos para os módulos, mas novas classes de implementação passaram a fazer parte de pacotes existentes, sendo automaticamente mapeadas para o módulo `DiL`. As novas classes `IfacadeRMITargetAdapter`, `HealthWatcherFacade` e `RMIFacadeAdapter` foram adicionadas ao pacote `healthwatcher.business` e a classe `RMIServletAdapter` ao pacote `healthwatcher.view`. Dessa forma, o módulo `DiL` passa a ser composto por 8 classes de implementação, como mostra a Listagem 5.15.

Listagem 5.15 – Classes Mapeadas para o Módulo DiL na Versão 05 de HW.

```
1 -DiL module implementation classes:
2   -healthwatcher.view.IFacade
3   -healthwatcher.business.IFacadeRMITargetAdapter
4   -lib.distribution.rmi.IteratorRMISourceAdapter
5   -lib.distribution.rmi.IIteratorRMITargetAdapter
6   -healthwatcher.business.HealthWatcherFacade
7   -lib.distribution.rmi.IteratorRMITargetAdapter
8   -healthwatcher.business.RMIFacadeAdapter
9   -healthwatcher.view.RMIServletAdapter
```

5.4.6 Versão 6

Na versão 6 de HW as mudanças realizadas tiveram como objetivo generalizar o mecanismo de persistência para melhorar o reuso e a extensibilidade. Para isso, o padrão de projeto *Factory* foi implementado. A aplicação do padrão fez surgir o pacote `healthwatcher.data.factories`, preservando demais pacotes advindos de versões anteriores. Essa mudança teve impacto direto sobre classes das camadas *Business* e *Data* representadas respectivamente por `BuL` e `DaL`. As novas classes criadas durante essas mudanças (`ArrayRepositoryFactory`, `RDBRepositoryFactory`, `AbstractRepositoryFactory` e `RepositoryFactory`) foram adicionadas ao novo pacote, porém, nenhuma alteração nos mapeamentos da especificação se fez necessária. Como consequência, o módulo `DaL` passa a ser composto por 25 classes, como mostrado na Listagem 5.16.

Listagem 5.16 – Classes Mapeadas para o Módulo DaL na Versão 06 de HW.

```
-DaL module implementation classes:
  -healthwatcher.data.mem.SpecialityRepositoryArray
  -healthwatcher.data.IDiseaseRepository
  -healthwatcher.data.rdb.ComplaintRepositoryRDB
  -healthwatcher.data.IHealthUnitRepository
  -lib.persistence.PersistenceMechanism
  -lib.persistence.IPersistenceMechanism
  -healthwatcher.data.factories.AbstractRepositoryFactory
  -healthwatcher.data.mem.SymptomRepositoryArray
  -healthwatcher.data.mem.EmployeeRepositoryArray
  -healthwatcher.data.factories.ArrayRepositoryFactory
  -healthwatcher.data.factories.RDBRepositoryFactory
  -healthwatcher.data.mem.DiseaseTypeRepositoryArray
  -healthwatcher.data.ISymptomRepository
  -healthwatcher.data.rdb.DiseaseTypeRepositoryRDB
  -healthwatcher.data.mem.ComplaintRepositoryArray
  -healthwatcher.data.IComplaintRepository
  -healthwatcher.data.rdb.AddressRepositoryRDB
  -healthwatcher.data.rdb.HealthUnitRepositoryRDB
  -healthwatcher.data.ISpecialityRepository
  -healthwatcher.data.rdb.EmployeeRepositoryRDB
  -healthwatcher.data.rdb.SpecialityRepositoryRDB
  -healthwatcher.data.IEmployeeRepository
  -healthwatcher.data.factories.RepositoryFactory
  -healthwatcher.data.mem.HealthUnitRepositoryArray
  -healthwatcher.data.IAddressRepository
```

5.4.7 Versão 7

As alterações feitas no sistema que resultaram a versão 7 de HW tiveram como objetivo remover as dependências entre os objetos de requisição e resposta do servlet para facilitar o processo de adicionar novas interfaces gráficas. Para isso, foi realizada a implementação do padrão de projeto *Adapter*, aos mesmos moldes da implementação realizada na versão 5 de HW. Essas mudanças não geraram acréscimo de pacotes, mas de novas classes aos pacotes `healthwatcher.view.servlets` e `healthwatcher.view.command`. As mudanças em questão tiveram impacto direto na camada *View* representada por `ViL`. Nessa versão, nenhum alteração nos mapeamentos da especificação se faz necessário, porém as classes `ServletRequestAdapter` e `ServletResponseAdapter` do pacote `healthwatcher.view.servlets` e as classes `CommandRequest` e `CommandResponse` do pacote `healthwatcher.view.command` passam a ser mapeadas para o módulo `ViL`. Com isso, o módulo `ViL` passa a ser composto por 29 classes, como mostrado na Listagem 5.17.

Listagem 5.17 – Classes Mapeadas para o Módulo ViL na Versão 7 de HW.

```
-ViL module implementation classes:
  -healthwatcher.view.command.UpdateHealthUnitSearch
  -healthwatcher.view.command.UpdateComplaintList
  -healthwatcher.view.command.ConfigRMI
  -healthwatcher.view.command.InsertSpecialComplaint
  -healthwatcher.view.command.InsertAnimalComplaint
  -healthwatcher.view.command.LoginMenu
  -healthwatcher.view.command.CommandRequest
  -healthwatcher.view.command.SearchHealthUnitsBySpecialty
  -healthwatcher.view.command.UpdateHealthUnitData
  -healthwatcher.view.servlets.ServletRequestAdapter
  -healthwatcher.view.command.UpdateEmployeeData
  -healthwatcher.view.command.SearchComplaintData
  -healthwatcher.view.servlets.HWServlet
  -healthwatcher.view.servlets.ServletResponseAdapter
  -healthwatcher.view.command.Login
  -healthwatcher.view.command.CommandResponse
  -healthwatcher.view.command.GetDataForSearchByDiseaseType
  -healthwatcher.view.command.UpdateHealthUnitList
  -healthwatcher.view.servlets.ServletWebServer
  -healthwatcher.view.command.Command
  -healthwatcher.view.command.InsertFoodComplaint
  -healthwatcher.view.command.UpdateEmployeeSearch
  -healthwatcher.view.command.UpdateComplaintData
  -healthwatcher.view.command.UpdateComplaintSearch
  -healthwatcher.view.command.InsertEmployee
  -healthwatcher.view.command.GetDataForSearchByHealthUnit
  -healthwatcher.view.command.GetDataForSearchBySpeciality
  -healthwatcher.view.command.SearchDiseaseData
  -healthwatcher.view.command.SearchSpecialtiesByHealthUnit
```

5.4.8 Versão 8

Na versão 8 de HW as mudanças realizadas tiveram como objetivo generalizar o mecanismo de distribuição para melhorar o reuso e a extensibilidade. Para isso, uma nova implementação do padrão de projeto *Factory* foi realizada. Essas mudanças tiveram como reflexo o acréscimo de um novo pacote chamado `healthwatcher.business.factories`, três novas classes `FacadeFactory`, `RMIFacadeFactory` e `AbstractFacadeFactory` pertencentes ao novo pacote e mais uma classe, `HWServer`, pertencente ao pacote `healthwatcher.business`, já existente. Embora as novas classes estejam dentro de pacotes que possuem a palavra "`business`", elas fazem parte da camada *Distribution*. Desse modo, foi necessário fazer uma alteração no mapeamento do módulo `DiL` para incluí-las (Listagem 5.18).

Listagem 5.18 – Mapeamento para o Módulo DiL na Versão 8 de HW.

```
1 diL = ArCatch.element().module("DiL").matching("(lib.distribution.rmi.[a-zA-Z_$
      ][a-zA-Z0-9_$]*|healthwatcher.view.(IFacade|RMIServletAdapter)|healthwatcher
      .business.(factories.[a-zA-Z_$][a-zA-Z0-9_$]*|HealthWatcherFacade|
      IFacadeRMITargetAdapter|RMIFacadeAdapter|HWServer))*").build();
```

Com a alteração do mapeamento, o módulo `DiL` passou a ser composto por 12 classes de implementação, como mostrado na Listagem 5.19.

Listagem 5.19 – Classes Mapeadas para o Módulo DiL na Versão 8 de HW.

```
1  -DiL module implementation classes:
2      -healthwatcher.business.HWServer
3      -healthwatcher.view.IFacade
4      -healthwatcher.business.IFacadeRMITargetAdapter
5      -lib.distribution.rmi.IteratorRMISourceAdapter
6      -lib.distribution.rmi.IIteratorRMITargetAdapter
7      -healthwatcher.business.factories.FacadeFactory
8      -healthwatcher.business.factories.RMIFacadeFactory
9      -healthwatcher.business.HealthWatcherFacade
10     -lib.distribution.rmi.IteratorRMITargetAdapter
11     -healthwatcher.business.RMIFacadeAdapter
12     -healthwatcher.business.factories.AbstractFacadeFactory
13     -healthwatcher.view.RMIServletAdapter
```

5.4.9 Versão 9

Na versão 9 de HW as alterações realizadas dizem respeito a implementação de uma nova funcionalidade. Essas mudanças implicam no acréscimo de um novo conjunto de classes voltadas à sintomas de saúde e tem impacto direto nas camadas *Business*, *View* e *Data*. Entre as mudanças, as principais foram a implementação das classes `SymptomRecord` e `SymptomRepositoryRDB` adicionadas, respectivamente, aos pacotes `healthwatcher.business.complaint` e `healthwatcher.data.rdb`; e das classes `InsertDiseaseType`, `InsertSymptom`, `InsertHealthUnit`, `InsertMedicalSpeciality`, `GetDataForSearchBySpeciality`, `UpdateSymptomData`, `UpdateMedicalSpecialityData`,

`UpdateSymptomList`, `UpdateMedicalSpecialitySearch`, `UpdateMedicalSpecialityList` e `UpdateSymptomSearch` adicionadas ao pacote `healthwatcher.view.command`. Nenhuma alteração nos mapeamentos fez-se necessária, porém, com a adição de novas classes os módulos `ViL`, `BuL` e `DaL` passaram a incorporar outras classes, como mostrado nas Listagens 5.20, 5.21 e 5.22.

Listagem 5.20 – Classes Mapeadas para o Módulo ViL na Versão 9 de HW.

```
-ViL module implementation classes:
  -healthwatcher.view.command.UpdateHealthUnitSearch
  -healthwatcher.view.command.UpdateComplaintList
  -healthwatcher.view.command.ConfigRMI
  -healthwatcher.view.command.InsertSpecialComplaint
  -healthwatcher.view.command.InsertAnimalComplaint
  -healthwatcher.view.command.LoginMenu
  -healthwatcher.view.command.CommandRequest
  -healthwatcher.view.command.SearchHealthUnitsBySpecialty
  -healthwatcher.view.command.UpdateHealthUnitData
  -healthwatcher.view.servlets.ServletRequestAdapter
  -healthwatcher.view.command.UpdateEmployeeData
  -healthwatcher.view.command.SearchComplaintData
  -healthwatcher.view.servlets.HWServlet
  -healthwatcher.view.servlets.ServletResponseAdapter
  -healthwatcher.view.command.UpdateMedicalSpecialityList
  -healthwatcher.view.command.Login
  -healthwatcher.view.command.UpdateSymptomSearch
  -healthwatcher.view.command.UpdateMedicalSpecialitySearch
  -healthwatcher.view.command.InsertSymptom
  -healthwatcher.view.command.CommandResponse
  -healthwatcher.view.command.GetDataForSearchByDiseaseType
  -healthwatcher.view.command.InsertMedicalSpeciality
  -healthwatcher.view.command.UpdateHealthUnitList
  -healthwatcher.view.servlets.ServletWebServer
  -healthwatcher.view.command.Command
  -healthwatcher.view.command.InsertHealthUnit
  -healthwatcher.view.command.UpdateMedicalSpecialityData
  -healthwatcher.view.command.InsertFoodComplaint
  -healthwatcher.view.command.InsertDiseaseType
  -healthwatcher.view.command.UpdateEmployeeSearch
  -healthwatcher.view.command.UpdateComplaintData
  -healthwatcher.view.command.UpdateSymptomList
  -healthwatcher.view.command.UpdateComplaintSearch
  -healthwatcher.view.command.InsertEmployee
  -healthwatcher.view.command.GetDataForSearchByHealthUnit
  -healthwatcher.view.command.UpdateSymptomData
  -healthwatcher.view.command.SearchDiseaseData
  -healthwatcher.view.command.GetDataForSearchBySpeciality
  -healthwatcher.view.command.SearchSpecialtiesByHealthUnit
```

Listagem 5.21 – Classes Mapeadas para o Módulo BuL na Versão 9 de HW.

```
-BuL module implementation classes:
  -healthwatcher.business.complaint.ComplaintRecord
  -healthwatcher.business.healthguide.HealthUnitRecord
  -healthwatcher.business.healthguide.MedicalSpecialityRecord
  -healthwatcher.business.complaint.DiseaseRecord
  -healthwatcher.business.complaint.SymptomRecord
  -healthwatcher.business.employee.EmployeeRecord
```

Listagem 5.22 – Classes Mapeadas para o Módulo DaL na Versão 9 de HW.

```
-DaL module implementation classes:
  -healthwatcher.data.mem.SpecialityRepositoryArray
  -healthwatcher.data.IDiseaseRepository
  -healthwatcher.data.rdb.ComplaintRepositoryRDB
  -healthwatcher.data.rdb.SymptomRepositoryRDB
  -healthwatcher.data.IHealthUnitRepository
  -lib.persistence.PersistenceMechanism
  -lib.persistence.IPersistenceMechanism
  -healthwatcher.data.factories.AbstractRepositoryFactory
  -healthwatcher.data.mem.SymptomRepositoryArray
  -healthwatcher.data.mem.EmployeeRepositoryArray
  -healthwatcher.data.factories.ArrayRepositoryFactory
  -healthwatcher.data.factories.RDBRepositoryFactory
  -healthwatcher.data.mem.DiseaseTypeRepositoryArray
  -healthwatcher.data.ISymptomRepository
  -healthwatcher.data.rdb.DiseaseTypeRepositoryRDB
  -healthwatcher.data.mem.ComplaintRepositoryArray
  -healthwatcher.data.IComplaintRepository
  -healthwatcher.data.rdb.AddressRepositoryRDB
  -healthwatcher.data.rdb.HealthUnitRepositoryRDB
  -healthwatcher.data.ISpecialityRepository
  -healthwatcher.data.rdb.EmployeeRepositoryRDB
  -healthwatcher.data.rdb.SpecialityRepositoryRDB
  -healthwatcher.data.IEmployeeRepository
  -healthwatcher.data.factories.RepositoryFactory
  -healthwatcher.data.mem.HealthUnitRepositoryArray
  -healthwatcher.data.IAddressRepository
```

5.4.10 Versão 10

Na versão 10 de HW as mudanças realizadas foram feitas na forma de refatorações com o objetivo de modularizar o tratamento de exceção e incluir um comportamento de recuperação de erros mais eficaz para os tratadores. Nessa alteração um novo pacote, `lib.logging`, foi adicionado ao sistema. As mudanças realizadas tiveram impacto nas classes das seguintes camadas: *Business*, *View* e *Data*. Três novas classes de exceção, `FacadeUnavailableException`, `SQLPersistenceMechanismException` e `ConnectionPersistenceMechanismException`, foram adicionadas em `lib.exceptions`. A primeira classe é uma exceção relacionada com a camada de *Distribution* e as duas últimas relacionadas com a camada *Data*. Desse modo, uma alteração nos mapeamentos

dos módulos de exceção `DiLEx` e `DaLEx` fez-se necessária. As alterações realizadas nos mapeamentos são mostradas nas Listagens 5.23 e 5.24.

Listagem 5.23 – Mapeamento para o Módulo de Exceção DiLEx na Versão 10 de HW.

```
diLEx = ArCatch.element().exception("DiLEx").matching("(java.rmi.RemoteException
    |lib.exceptions.(CommunicationException|FacadeUnavailableException))*").
    build();
```

Listagem 5.24 – Mapeamento para o Módulo de Exceção DaLEx na Versão 10 de HW.

```
daLEx = ArCatch.element().exception("DaLEx").matching("lib.exceptions.(
    Persistence|ObjectNot|Repository|Transaction|SQLPersistenceMechanism|
    ConnectionPersistenceMechanism)[a-zA-Z_$][a-zA-Z0-9_$]*").build();
```

Com a alteração nos mapeamentos, `DiLEx` passou a ser mapeado para 3 classes de exceção (Listagem 5.25) e `DaLEx` passou a ser mapeado para 7 classes de exceção (Listagem 5.26).

Listagem 5.25 – Classes de Exceção Mapeadas para DiLEx na Versão 10 de HW.

```
-DiLEx exception implementation classes:
  -lib.exceptions.CommunicationException
  -lib.exceptions.FacadeUnavailableException
  -java.rmi.RemoteException
```

Listagem 5.26 – Classes de Exceção Mapeadas para DaLEx na Versão 10 de HW.

```
-DaLEx exception implementation classes:
  -lib.exceptions.ObjectNotFoundException
  -lib.exceptions.RepositoryException
  -lib.exceptions.PersistenceMechanismException
  -lib.exceptions.ObjectNotValidException
  -lib.exceptions.ConnectionPersistenceMechanismException
  -lib.exceptions.SQLPersistenceMechanismException
  -lib.exceptions.TransactionException
```

5.5 Resultados e Discussão

A Tabela 7 resume os resultados obtidos na verificação de conformidade de todas as 10 versões do sistema Health Watcher (HW). Cada versão de HW foi verificada contra o mesmo conjunto de 16 regras de design de tratamento de exceção apresentadas na Tabela 6 da Seção 5.3. Como pode ser observado na tabela, todas as versões analisadas estavam em total conformidade com 6 regras de design (R02, R04, R07, R11, R12 e R14) e em não conformidade com outras 7 regras design (R01, R05, R06, R08, R09, R10, e R16). Por outro lado, três regras, R03, R13 e R15, possuem um comportamento diferente. As regras R03 e R13 começam sendo satisfeitas nas versões iniciais e depois passam a ser violadas a partir das versões 10 e 4, respectivamente. De forma contrária, a regra R15 é violada em todas as 9 versões iniciais do sistema e passa a ser satisfeita apenas na última versão.

Em resumo, as versões v1–v3 possuem 50% de grau de conformidade e as versões v4–v10 possuem 44% de grau de conformidade.

Tabela 7 – Verificação das regras de design para todas as versões de Health Watcher.

ID	**Versões de Health Watcher**									
	01	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**10**
R01	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
R02	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
R03	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
R04	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
R05	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
R06	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
R07	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
R08	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
R09	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
R10	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
R11	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
R12	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
R13	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗
R14	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
R15	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓
R16	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗

Analisando o relatório de conformidade de ArCatch.Checker (Listagem 5.27), a regra R03 é violada na versão 10 pelo fato de que após a modularização do código responsável pelo tratamento de exceção, como mencionado na Tabela 5, o método `initFacade()` da classe `HWServlet` começa a sinalizar a exceção `CommunicationException`. Já R013 começa a ser violada na versão 4, por causa da implementação do padrão *Observer*, onde o método `notify()` da classe `Subject` começa a sinalizar as exceções `ObjectNotFoundException`, `RepositoryException`, `ObjectNotValidException` e `TransactionException`, como descrito na Listagem 5.28.

Listagem 5.27 – Exemplo do Relatório da Regra R03 (HW v10).

```
1 -Rule Violations
2  -The method [healthwatcher.view.servlets.HWServlet.initFacade()] is signaling
      the exception [lib.exceptions.CommunicationException]
```

Listagem 5.28 – Exemplo do Relatório da Regra R13 (HW v4).

```
1 -Rule Violations
2   -The method [lib.patterns.observer.Observer.notify(lib.patterns.observer.
        Subject)] is signaling the exception [lib.exceptions.
        ObjectNotFoundException]
3   -The method [lib.patterns.observer.Observer.notify(lib.patterns.observer.
        Subject)] is signaling the exception [lib.exceptions.RepositoryException]
4   -The method [lib.patterns.observer.Observer.notify(lib.patterns.observer.
        Subject)] is signaling the exception [lib.exceptions.
        ObjectNotValidException]
5   -The method [lib.patterns.observer.Observer.notify(lib.patterns.observer.
        Subject)] is signaling the exception [lib.exceptions.TransactionException]
```

A regra R15 é violada da versão 1 até a versão 9 e começa ser satisfeita na versão 10. As violações ocorriam devido ao fato do módulo `DaL` sinalizar a exceção `SQLException` (Listagem 5.29). Porém, depois da refatoração do tratamento de exceção ocorrido na versão 10, essa violação deixou de ocorrer.

Listagem 5.29 – Exemplo do Relatório da Regra R15 (HW v1–v9).

```
1 -Rule Violations
2   -The method [healthwatcher.data.rdb.ComplaintRepositoryRDB.accessComplaint(
        java.sql.ResultSet,healthwatcher.model.complaint.Complaint)] is signaling
        the exception [java.sql.SQLException]
```

Em sequência, são detalhadas as demais regras de design detectadas como violadas na verificação de conformidade realizada. O conjunto completo de relatórios gerados a partir da verificação de conformidade de todas as 10 versões de HW pode ser encontrado no repositório online do experimento[5].

A R01 é violada ao longo de todas as versões de HW. O motivo desta violação deve-se ao fato de que existem vários métodos das classes mapeadas para o módulo `ViL` que não tratam ao menos uma classe de exceção mapeada para o módulo de exceção `DiLEx`. A seguir, a Listagem 5.30 mostra um trecho do relatório de verificação de conformidade gerado para esta violação.

Listagem 5.30 – Exemplo do Relatório da Regra R01 (HW v1).

```
1 (...)
2 -Rule Violations
3   -The method [healthwatcher.view.servlets.ServletUpdateHealthUnitData.doPost(
        javax.servlet.http.HttpServletRequest,javax.servlet.http.
        HttpServletResponse)] is not handling at least one exception in [DiLEx]
4   -The method [healthwatcher.view.servlets.ServletInsertEmployee.doPost(javax.
        servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)]
        is not handling at least one exception in [DiLEx]
5 (...)
```

A regra R05, de forma semelhante à R01, é violada ao longo de todas as versões de HW. Porém, o motivo desta violação deve-se ao fato de existem métodos de classes que não

[5] https://github.com/juarezmeneses/ArCatchExperiment

estão mapeadas para o módulo `DiL` sinalizando alguma classe de exceção mapeada para o módulo de exceção `DiLEx`. Um ponto importante, é que a partir da versão 5 de HW, mais uma fonte de violação da regra é adicionada, devido à implementação do padrão *Observer* (linha 4 da Listagem 5.31). A seguir, a Listagem 5.31 mostra um trecho do relatório gerado para esta violação.

Listagem 5.31 – Exemplo do Relatório da Regra R05 (HW v5).

```
1 -Rule Violations
2     -The method [lib.util.IteratorDsk.hasNext()] is signaling the exception [lib
          .exceptions.CommunicationException]
3   -The method [lib.util.IteratorDsk.remove()] is signaling the exception [lib.
        exceptions.CommunicationException]
4   -The method [lib.patterns.observer.Observer.notify(lib.patterns.observer.
        Subject)] is signaling the exception [java.rmi.RemoteException]
```

A regra R06 também é violada ao longo de todas as versões de HW. Porém, o motivo desta violação deve-se ao fato de que existem vários métodos das classes mapeadas para o módulo `DiL` que não tratam ao menos uma classe de exceção mapeada para o módulo de exceção `BuLEx`, contrariando a inteção da regra. A seguir, a Listagem 5.32 mostra um trecho do relatório gerado para esta violação.

Listagem 5.32 – Exemplo do Relatório da Regra R06 (HW v1).

```
1 -Rule Violations
2     -The method [healthwatcher.business.HealthWatcherFacadeInit.
          getSpecialityList()] is not handling at least one exception in [BuLEx]
3     -The method [healthwatcher.view.IFacade.updateComplaint(healthwatcher.model.
          complaint.Complaint)] is not handling at least one exception in [BuLEx]
```

A regra R08, é violada ao longo de todas as versões de HW. Nesse caso, conforme o trecho do relatório de verificação de conformidade mostrado na Listagem 5.33, existem métodos de classes que não foram mapeadas para o módulo `BuL` levantando classes de exceção mapeadas para o módulo de exceção `BuLEx`, contrariando a intenção da regra.

Listagem 5.33 – Exemplo do Relatório da Regra R08 (HW v1).

```
1 -Rule Violations
2     -The method [healthwatcher.data.rdb.AddressRepositoryRDB.insert(
          healthwatcher.model.address.Address)] is raising the exception [lib.
          exceptions.ObjectAlreadyInsertedException]
```

A regra R09, semelhante à regra anterior, é violada ao longo de todas as versões de HW. O motivo dessa violação é o fato de que existem alguns métodos de classes que não estão mapeadas para o módulo `BuL` sinalizando alguma classe de exceção mapeada para o módulo de exceção `BuLEx`. A Listagem 5.34 mostra um trecho do relatório gerado para esta violação.

Listagem 5.34 – Exemplo do Relatório da Regra R09 (HW v1).

```
1 -Rule Violations
2     -The method [healthwatcher.data.IDiseaseRepository.insert(healthwatcher.
          model.complaint.DiseaseType)] is signaling the exception [lib.exceptions
          .ObjectAlreadyInsertedException]
3   -The method [healthwatcher.data.rdb.ComplaintRepositoryRDB.insert(
        healthwatcher.model.complaint.Complaint)] is signaling the exception [lib.
        exceptions.ObjectAlreadyInsertedException]
```

A regra R10, também é violada ao longo de todas as versões de HW. Porém, essa violação deve-se ao fato de que existem vários métodos das classes mapeadas para o módulo `BuL` que não tratam ao menos uma classe de exceção mapeada para o módulo de exceção `DaLEx`, o que viola a intenção da regra. A seguir, a Listagem 5.35 mostra um trecho do relatório gerado para esta violação.

Listagem 5.35 – Exemplo do Relatório da Regra R10 (HW v1).

```
1 -Rule Violations
2     -The method [healthwatcher.business.complaint.ComplaintRecord.validate(
          healthwatcher.model.complaint.Complaint)] is not handling at least one
          exception in [DaLEx]
3   -The method [healthwatcher.business.healthguide.MedicalSpecialityRecord.
        getListaEspecialidade()] is not handling at least one exception in [DaLEx]
```

Por fim, a regra R16 é violada desde a primeira versão pelo fato de que três exceções diferentes (`ObjectNotFoundException`, `RepositoryException` e `ObjectNotValidException`) estão fluindo através dos módulos `DaL` `BuL` e `DiL`. Na verdade, a camada `BuL` não está em conformidade com as estratégias *catch-and-handle* e *catch-and-remap*. A Listagem 5.36 mostra parte do relatório de conformidade da ArCatch para a verificação da regra R16.

Listagem 5.36 – Exemplo do Relatório da Regra R16 (HW v1).

```
1 -Rule Violations
2   -The exception [lib.exceptions.ObjectNotFoundException] is flowing through [
        DaL, BuL, DiL]
3   -The exception [lib.exceptions.RepositoryException] is flowing through [DaL,
        BuL, DiL]
4   -The exception [lib.exceptions.ObjectNotValidException] is flowing through [
        DaL, BuL, DiL]
```

Para validar ainda mais os resultados, foi realizada uma entrevista com o arquiteto de software de HW. Primeiramente, foi discutido sobre as sete regras de design que são violadas em todas as versões. Assim, após analisar os relatórios das violações, o arquiteto reconheceu que todas as violações representavam desvios claros de sua intenção como arquiteto de software, confirmando a existência de problemas de erosão do tratamento de exceção em HW.

Em segundo lugar, foram discutidos os casos particulares de violação das regras de design R03 e R13. Olhando para o relatório de violação da R03, o arquiteto reconheceu

que tal violação introduzida na versão 10 é um erro cometido por um desenvolvedor e uma possível solução poderia ser criar um bloco try-catch no método `initFacade()` para capturar a exceção `CommunicationException` e, dentro do bloco catch, executar uma operação de `log` ou um redirecionamento de página para apresentar o erro corretamente. Analisando o relatório de violação da R13, o arquiteto argumentou que tal violação não é uma violação propriamente dita, mas um efeito colateral causado pela implementação do padrão de projeto *Observer*. No entanto, ele decidiu que deveria ser corrigida em uma versão futura. Finalmente, olhando para o relatório de violação da R15, o arquiteto não teve dúvidas de que tal violação representa um desvio de sua intenção, que foi corrigido na versão 10.

Nenhuma avaliação sobre o desempenho e a usabilidade de ArCatch foi realizada nesta dissertação de mestrado. No entanto, no cenário da avaliação, ArCatch leva cerca de 50 segundos (em média) para executar a análise do código-fonte e verificar a conformidade de cada regra de design em cada versão de HW. Depois de definir a política de tratamento de exceção, a especificação de todas as regras de design usando ArCatch.Rules leva menos de 1h. O processo de mapeamento foi a parte mais demorada, uma vez que não havia familiaridade prévia do autor desta dissertação com o código-fonte de HW; Foi preciso analisar manualmente o código fonte de cada versão. A análise da primeira versão demorou mais de 5h, enquanto a soma do tempo de análise de todas as outras versões levaram cerca de 5h, portanto, todo o processo de mapeamento levou um pouco mais de 10h.

5.6 Considerações Finais

Este capítulo apresentou a avaliação realizada com ArCatch. Foi detalhado um cenário composto por: manutenções e evoluções de software, em um sistema real chamado Health Watcher. A avaliação buscou demonstrar a expressividade e a viabilidade da ferramenta proposta, na verificação da conformidade arquitetural do tratamento de exceções. Adicionalmente, uma discussão foi conduzida, detalhando os mapeamentos realizados, as regras especificadas e os resultados obtidos na verificação da conformidade realizada para cada uma das 10 versões da aplicação analisada.

Observando os resultados obtidos, é possível afirmar que a ArCatch consegue detectar violações de regras de design do tratamento de exceção, e, portanto, identificar possíveis problemas de erosão do tratamento de exceção, contribuindo para um dos objetivos esperados. Além disso, o relatório de conformidade gerado por ArCatch traz informações que permitem localizar a causa das violações diretamente no código-fonte. Por fim, vale ressaltar que as 16 regras de design elicitadas para a avaliação, demonstram apenas parte do poder de expressão da ArCatch.Rules na definição de regras de design do tratamento de exceção, visto que não são necessários o uso de todas as combinações

possíveis de construtos para a definição de uma política de tratamento de exceção válida e útil ao contexto analisado.

Os resultados obtidos nesse capítulo demonstram a expressividade e viabilidade da solução proposta e o seu alinhamento com as metas MET02 e MET03, além de corroborar com o objetivo geral desta dissertação de mestrado, citados no Capítulo 1. No próximo capítulo são apresentadas as conclusões finais sobre esta dissertação, as limitações e os possíveis direcionamentos para trabalhos futuros.

6 Conclusão

Neste capítulo, são realizadas as considerações finais desta dissertação. A Seção 6.1 apresenta uma visão geral do trabalho. Na Seção 6.2, é feita uma análise do objetivo de pesquisa. Em seguida, a Seção 6.3 descreve as limitações deste trabalho. Por fim, na Seção 6.4, são apresentados os trabalhos futuros.

6.1 Visão Geral do Trabalho

Esta dissertação de mestrado investigou o problema de erosão arquitetural do tratamento de exceção que tem sua origem na negligência dos desenvolvedores com respeito ao código de tratamento de exceção e na falta de meios apropriados para documentar o design do tratamento de exceção e permitir que este seja melhor compartilhado, discutido e compreendido por toda a equipe.

A solução escolhida para lidar com esse problema foi a verificação de conformidade arquitetural. Uma abordagem que preconiza a verificação periódica do grau de aderência da arquitetura descritiva à arquitetura prescritiva do software. Entretanto, como discutido no Capítulo 3, as abordagens existentes de verificação de conformidade arquitetural não proveem suporte adequado e completo para a verificação da conformidade do tratamento de exceção.

Para preencher essa lacuna, esta dissertação de mestrado propôs ArCatch, uma solução de verificação de conformidade arquitetural para o tratamento de exceção. ArCatch permite especificar e verificar regras de design do tratamento de exceção, afim de evitar a erosão arquitetural em projetos de software. ArCatch é composta de duas partes: (i) ArCatch.Rules uma linguagem específica de domínio (DSL) interna usada para criação das regras de design; e (ii) ArCatch.Checker um engenho responsável por realizar as verificações de conformidade das regras descritas em ArCatch.Rules.

Por fim, foi apresentada a abordagem utilizada para avaliar a ArCatch, que consistiu na análise do sistema Health Watcher em uma cenário real de evolução de software.

6.2 Análise do Objetivo

O objetivo desta dissertação foi definido no Capítulo 1 e retomado nesta seção. Para isso, está listado novamente a seguir com o propósito de analisar a sua validade frente aos resultados alcançados:

Objetivo Geral:

> *Desenvolver uma solução de verificação de conformidade arquitetural para combater erosão arquitetural do tratamento de exceção.*

Com base na descrição do projeto e implementação de ArCatch apresentados no Capítulo 4 e na avaliação conduzida no Capítulo 5 desta dissertação, pode-se afirmar que o objetivo principal dessa dissertação foi atingido com sucesso.

6.3 Limitações

Uma das principais limitações de ArCatch está relacionada ao suporte dado apenas para projetos de software escritos na linguagem Java. Embora Java seja uma linguagem de programação popular, projetos de software escritos em linguagens de programação como C# poderiam se beneficiar das funcionalidades providas por ArCatch.

Outra limitação de ArCatch está relacionada com a falta de suporte a múltiplos módulos e exceções na definição de regras. Por exemplo, só uma das seguintes regras pode ser verdadeira: "`only` A `can signal` E" e "`only` B `can signal` E". Para que ambas pudessem ser verdadeiras simultaneamente, ArCatch.Rules deveria permitir a escrita de regras do tipo: "`only` A,B `can signal` E".

Uma terceira limitação está no fator expressividade da linguagem fornecida pela ArCatch. Apesar de ArCatch permitir expressar muitas regras de design, todas as regras são específicas, por concepção, para o tratamento de exceção. Desse modo, seria interessante incorporar outros tipos de relações de dependência à ArCatch.Rules.

Por fim, outra limitação de ArCatch está relacionada com o formato de apresentação do relatório de conformidade gerado por ArCatch. O formato textual exibe todas as informações em um mesmo plano, o que pode dificultar o entendimento. Desse modo, uma possível melhoria seria fazer a geração do relatório em formato hipertexto usando tecnologias como HTML e CSS, tornando a apresentação das informações mais gradual e interativa.

6.4 Trabalhos Futuros

Como um dos trabalhos futuros, é importante realizar uma avaliação de usabilidade de ArCatch com o intuito de identificar quão fácil ela é de ser utilizada por desenvolvedores e arquitetos de software. Além disso, um outro possível desdobramento deste trabalho está relacionado ao suporte a regras anti-desvio, seja por meio da ampliação da linguagem ou da criação de uma nova ferramenta integrada a ArCatch. Outra linha que pode ser seguida é a

de geração de testes de software a partir das regras de design como forma de implementar uma verificação dinâmica de conformidade do tratamento de exceção. Por fim, outro possível trabalho futuro está relacionado ao processo de recomendação de refatorações para reconciliação arquitetural em casos em que regras de design do tratamento de exceção são violadas.

Para finalizar, vale ressaltar que a solução proposta nesta dissertação busca ajudar tanto aos desenvolvedores, quanto aos arquitetos de software, em suas tarefas de implementação, evolução e manutenção de software voltados ao tratamento de exceção, auxiliando na prevenção de possíveis problemas de erosão do tratamento de exceção.

Referências

AVIZIENIS, A. et al. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 1, n. 1, p. 11–33, 2004. ISSN 1545-5971. Citado na página 25.

BARBOSA, E. A.; GARCIA, A.; BARBOS, S. D. J. Categorizing faults in exception handling: A study of open source projects. In: *Software Engineering (SBES), 2014 Brazilian Symposium on*. [S.l.: s.n.], 2014. p. 11–20. Citado 2 vezes nas páginas 16 e 31.

BARBOSA, E. A. et al. Enforcing exception handling policies with a domain-specific language. *IEEE Transactions on Software Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 42, n. 6, p. 559–584, 2016. ISSN 0098-5589. Citado 5 vezes nas páginas 17, 31, 34, 36 e 38.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN 0-201-19930-0. Citado 2 vezes nas páginas 20 e 21.

BRUNET, J.; GUERRERO, D.; FIGUEIREDO, J. 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*. [S.l.: s.n.], 2009. p. 255–258. Citado na página 49.

BUHR, P. A.; MOK, W. Y. R. Advanced exception handling mechanisms. *IEEE Transactions on Software Engineering*, IEEE Press, Piscataway, NJ, USA, v. 26, p. 820–836, September 2000. ISSN 0098-5589. Citado 2 vezes nas páginas 15 e 26.

CABRAL, B.; MARQUES, P. Exception handling: A field study in java and .net. In: *Proceedings of the 21st European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2007. (ECOOP'07), p. 151–175. ISBN 3-540-73588-7, 978-3-540-73588-5. Citado na página 16.

CACHO, N. et al. How does exception handling behavior evolve? an exploratory study in java and c# applications. In: *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*. Washington, DC, USA: IEEE Computer Society, 2014. (ICSME '14), p. 31–40. ISBN 978-1-4799-6146-7. Citado na página 16.

CACHO, N. et al. Trading robustness for maintainability: An empirical study of evolving c# programs. In: *Proceedings of the 36th International Conference on Software Engineering*. [S.l.: s.n.], 2014. (ICSE 2014), p. 584–595. ISBN 978-1-4503-2756-5. Citado na página 15.

CARACCIOLO, A.; LUNGU, M.; NIERSTRASZ, O. A unified approach to architecture conformance checking. In: *Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. [S.l.]: ACM Press, 2015. p. 41–50. Citado 6 vezes nas páginas 15, 16, 23, 34, 35 e 37.

CRISTIAN, F. Exception handling. In: ANDERSON, T. (Ed.). *Dependability of Resilient Computers*. [S.l.]: Blackwell Scientific Publications, 1989. p. 68–97. Citado na página 16.

EBERT, F.; CASTOR, F. A study on developers' perceptions about exception handling bugs. In: *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. [S.l.: s.n.], 2013. p. 448–451. ISSN 1063-6773. Citado 2 vezes nas páginas 16 e 31.

EBERT, F.; CASTOR, F.; SEREBRENIK, A. An exploratory study on exception handling bugs in java programs. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 106, n. C, p. 82–101, ago. 2015. ISSN 0164-1212. Citado 2 vezes nas páginas 16 e 31.

EICHBERG, M. et al. Defining and continuous checking of structural program dependencies. In: *Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008. (ICSE '08), p. 391–400. ISBN 978-1-60558-079-1. Citado 3 vezes nas páginas 16, 34 e 37.

EYCK, J. V. et al. Using code analysis tools for architectural conformance checking. In: *Proceedings of the 6th International Workshop on SHAring and Reusing Architectural Knowledge*. New York, NY, USA: ACM, 2011. (SHARK '11), p. 53–54. ISBN 978-1-4503-0596-9. Citado 2 vezes nas páginas 22 e 24.

FERRARI, F. et al. An exploratory study of fault-proneness in evolving aspect-oriented programs. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010. (ICSE '10), p. 65–74. ISBN 978-1-60558-719-6. Citado na página 54.

FOWLER, M. *Domain Specific Languages*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2010. ISBN 0321712943, 9780321712943. Citado na página 49.

GALLARDO, R. et al. *The Java Tutorial: A Short Course on the Basics*. 6th. ed. [S.l.]: Addison-Wesley Professional, 2014. 864 p. (Java Series). ISBN 0134034082. Citado 3 vezes nas páginas 10, 28 e 29.

GARCIA, A. F. et al. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, v. 59, n. 2, p. 197–222, 2001. ISSN 0164-1212. Citado 4 vezes nas páginas 10, 15, 26 e 27.

GOODENOUGH, J. B. Exception handling: Issues and a proposed notation. *Communications of the ACM*, ACM Press, New York, NY, USA, v. 18, p. 683–696, December 1975. ISSN 0001-0782. Citado 2 vezes nas páginas 15 e 26.

GREENWOOD, P. et al. On the impact of aspectual decompositions on design stability: An empirical study. In: *Proceedings of the 21st European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2007. (ECOOP'07), p. 176–200. Citado 3 vezes nas páginas 11, 54 e 55.

GURGEL, A. et al. Blending and reusing rules for architectural degradation prevention. In: *Proceedings of the 13th International Conference on Modularity*. New York, NY, USA: ACM, 2014. (MODULARITY '14), p. 61–72. ISBN 978-1-4503-2772-5. Citado 5 vezes nas páginas 16, 34, 35, 36 e 54.

GURP, J. van; BOSCH, J. Design erosion: Problems and causes. *Journal of Systems and Software*, v. 61, n. 2, p. 105–119, 2002. ISSN 0164-1212. Citado na página 15.

ISO/IEC 42010, . ISO/IEC Standard for Systems and Software Engineering - Recommended Practice for Architectural Description of Software-Intensive Systems. *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, p. c1–24, July 2007. Citado na página 21.

JENKOV, J. *Java Exception Handling*. 1nd. ed. [S.l.]: Amazon Kindle, 2013. Citado na página 28.

KECHAGIA, M.; SPINELLIS, D. Undocumented and unchecked: Exceptions that spell trouble. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2014. (MSR 2014), p. 312–315. ISBN 978-1-4503-2863-0. Citado na página 16.

KIENZLE, J. On exceptions and the software development life cycle. In: *Proceedings of the 4th International Workshop on Exception Handling*. New York, NY, USA: ACM Press, 2008. (WEH'08), p. 32–38. ISBN 978-1-60558-229-0. Citado na página 26.

LEE, P. A.; ANDERSON, T. *Fault Tolerance: Principles and Practice*. 2nd. ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1990. ISBN 0387820779. Citado na página 26.

MARINESCU, C. Are the classes that use exceptions defect prone? In: *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*. New York, NY, USA: ACM, 2011. (IWPSE-EVOL '11), p. 56–60. ISBN 978-1-4503-0848-9. Citado na página 16.

MARINESCU, C. Should we beware the exceptions? an empirical study on the eclipse project. In: *Proceedings of the 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. Washington, DC, USA: IEEE Computer Society, 2013. (SYNASC '13), p. 250–257. ISBN 978-1-4799-3036-4. Citado na página 16.

MILLER, R.; TRIPATHI, A. Issues with exception handling in object-oriented systems. In: AKSIT, M.; MATSUOKA, S. (Ed.). *ECOOP'97 - Object-Oriented Programming*. [S.l.]: Springer Berlin / Heidelberg, 1997, (Lecture Notes in Computer Science, v. 1241). p. 85–103. Citado na página 26.

MOOR, O. d. et al. Keynote address: .ql for source code analysis. In: *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*. Washington, DC, USA: IEEE Computer Society, 2007. (SCAM '07), p. 3–16. ISBN 0-7695-2880-5. Citado 3 vezes nas páginas 16, 34 e 37.

MURPHY, G. C.; NOTKIN, D.; SULLIVAN, K. 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*. New York, NY, USA: ACM, 1995. (SIGSOFT '95), p. 18–28. ISBN 0-89791-716-2. Citado na página 23.

OIZUMI, W. N. et al. On the relationship of code-anomaly agglomerations and architectural problems. *Journal of Software Engineering Research and Development*, v. 3, n. 1, p. 1–22, 2015. ISSN 2195-1721. Citado na página 54.

PARNAS, D. L.; WüRGES, H. Response to undesired events in software systems. In: *Proceedings of the 2nd International Conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976. (ICSE'76), p. 437–446. Citado 2 vezes nas páginas 15 e 26.

PASSOS, L. et al. Static architecture-conformance checking: An illustrative overview. *Software, IEEE*, v. 27, n. 5, p. 82–89, Sept 2010. ISSN 0740-7459. Citado 3 vezes nas páginas 15, 23 e 24.

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 17, n. 4, p. 40–52, out. 1992. ISSN 0163-5948. Citado 3 vezes nas páginas 15, 20 e 22.

ROCHA, L. S. *CAEHV: Um Método para Vericacão de Modelos do Tratamento de Exceção Sensível ao Contexto em Sistemas Ubiquos*. Tese (Doutorado) — Universidade Federal do Ceará, Fotaleza-CE, 2013. Citado na página 26.

SHAH, H.; GORG, C.; HARROLD, M. J. Understanding exception handling: Viewpoints of novices and experts. *IEEE Transactions on Software Engineering*, IEEE Press, Piscataway, NJ, USA, v. 36, n. 2, p. 150–161, mar. 2010. ISSN 0098-5589. Citado na página 16.

SILVA, L. de; BALASUBRAMANIAM, D. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, Elsevier Science Inc., New York, NY, USA, v. 85, n. 1, p. 132–151, jan 2012. ISSN 0164-1212. Citado 3 vezes nas páginas 15, 22 e 24.

SOARES, S.; LAUREANO, E.; BORBA, P. Implementing distribution and persistence aspects with aspectj. In: *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 2002. (OOPSLA '02), p. 174–190. ISBN 1-58113-471-1. Citado na página 54.

TAYLOR, R. N.; MEDVIDOVIC, N.; DASHOFY, E. M. *Software Architecture: Foundations, Theory, and Practice*. [S.l.]: Wiley Publishing, 2009. ISBN 0470167742, 9780470167748. Citado 2 vezes nas páginas 15 e 21.

TERRA, R.; VALENTE, M. T. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, John Wiley & Sons, Ltd., v. 39, n. 12, p. 1073–1094, 2009. ISSN 1097-024X. Citado 5 vezes nas páginas 16, 22, 34, 35 e 36.

TERRA, R. et al. A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, v. 45, n. 3, p. 315–342, 2015. ISSN 1097-024X. Citado 2 vezes nas páginas 25 e 35.

ZHANG, P.; ELBAUM, S. Amplifying tests to validate exception handling code: An extended study in the mobile application domain. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 23, n. 4, p. 32:1–32:28, set. 2014. ISSN 1049-331X. Citado na página 16.