UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

Dissertação de Mestrado

# Avaliando a Robustez e Manutenibilidade do Comportamento Excepcional de Aplicações C#

**Eliezio Soares de Sousa Neto**

Natal-RN
Abril 2014

**ELIEZIO SOARES DE SOUSA NETO**

# Avaliando a Robustez e Manutenibilidade do Comportamento Excepcional de Aplicações C#

Dissertação apresentada ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como requisito parcial para a obtenção do grau de Mestre em Sistemas e Computação.

**Prof. Dr. Nélio Cacho**

**Orientador**

Natal-RN
Abril 2014

Catalogação da Publicação na Fonte. UFRN / SISBI / Biblioteca Setorial
Centro de Ciências Exatas e da Terra – CCET.

Sousa Neto, Eliezio Soares de.
Avaliando a robustez e manutenibilidade do comportamento excepcional de aplicações C# / Eliezio Soares de Sousa Neto. - Natal, 2014.
109 f. : il.

Orientador: Prof. Dr. Nélio Cacho.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Programa de Pós-Graduação em Sistemas e Computação.

1. Engenharia de software – Dissertação. 2. Tratamento de exceções – Dissertação. 3. Mecanismos de tratamento de exceções – Dissertação. 4. Robustez - Dissertação. 5. Manutenibilidade - Dissertação. I. Cacho, Nélio. II. Título.

ELIEZIO SOARES DE SOUSA NETO

# Avaliando a Robustez e Manutenibilidade do Comportamento Excepcional em Aplicações C#

Esta Dissertação de Mestrado foi julgada adequada para a obtenção do título de mestre em Sistemas e Computação e aprovado em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

_____

Prof. Dr. Nélio Alessandro de Azevedo Cacho - UFRN
Orientador

_____

Prof. Dr. Martin Alejandro Musicante - UFRN
Coordenador do Programa

**Banca Examinadora**

_____

Presidente: Prof. Dr. Nélio Alessandro de Azevedo Cacho

_____

Examinador: Prof. Dr. Umberto Souza da Costa

_____

Examinador: Prof. Dr. Fernando José Castor de Lima Filho

Abril, 2014

# AGRADECIMENTOS

Como disse o genial Albert Einstein: “se vi mais longe foi por estar de pé sobre ombros de gigantes”. Particularmente possuo os meus próprios gigantes, aos quais desejo agradecer e sem os quais nada do que realizei, existiria.

A Deus que me capacitou e me capacita, que me sustenta e fortalece para que os mais difíceis obstáculos, tornem-se pequenos pedregulhos. Sem sua presença e amor, nada de bom poderia existir em mim. Agradeço pela certeza de que nada, seja morte, seja vida, sejam coisas do passado, ou do futuro, altura, ou profundidade, poderá me separar de seu amor (Rm 8 : 38,39).

Aos meus pais, Misael e Lucimar, que sempre me ensinaram, com suas próprias vidas, que tudo o que se quer, se pode e nada do que queremos pode ser maior do que o que somos. Com eles aprendí a ser homem e que nada na vida pode ser maior do que os valores que aprendi. Vocês são as minhas raízes. Sem vocês eu não existiria.

A minha amada irmã, Mikaely, que cumprindo o seu papel de primogênita, sempre me mostrou as boas coisas do mundo e com cuidado e carinho soube apontar e repreender as más.

A Jéssica, minha amada esposa, que me suportou, no mais profundo sentido da palavra, sempre com palavras e gestos capazes de abrandar os mais agitados e estressados pensamentos; que com seu carinho, suas palavras, seu “ouvido”, foi capaz de me manter focado, mesmo quando mente e corpo pareciam querer desistir; que me motiva com seu companheirismo, com as boas, longas e sinceras conversas, com os sonhos e as lutas compartilhadas.

A meu orientador, Nélio Cacho, que pacientemente me suportou durante todo esse tempo. O meu muito obrigado. Se este trabalho agora se torna realidade devo demais a você, obrigado por me ensinar o que precisava aprender, obrigado pela oportunidade desse tempo precioso no mestrado.

Por fim, a todos, que direta ou indiretamente, contribuíram para a realização deste trabalho.

À minha família que sempre, independente de circunstâncias me apoiou.

A Deus, pela VIDA e pela certeza que me deu de sua existência, " porque dEle e por Ele, e para Ele, são todas as coisas; glória, pois, a Ele eternamente. Amém." (Romanos 11:36)

# RESUMO

As linguagens de programação convencionais fornecem mecanismos de tratamento de exceções internos a fim de prover implementação de tratamento de exceções em sistemas de software robusto e manutenível. A maioria dessas linguagens modernas, tais como C#, Ruby, Python e muitas outras, frequentemente afirmaram ter mecanismos de tratamento de exceção mais apropriados. Elas reduzem as restrições de programação no tratamento de exceção buscando agilizar mudanças no código fonte. Essas linguagens suportam o que chamamos de mecanismos de tratamento de exceções dirigidos a manutenção. Espera-se que a adoção desses mecanismos aumentem a manutenibilidade dos softwares sem impactar na robustez. Entretanto ainda existe pouco conhecimento empírico acerca do impacto que a adoção desses mecanismos tem na robustez de softwares. Este trabalho busca preencher essa lacuna conduzindo um estudo empírico direcionado ao entendimento da relação entre mudanças em programas C# e sua robustez. Em particular, nós avaliamos como mudanças nos códigos normal e excepcional se relacionam com faltas no tratamento de exceção. Nós aplicamos uma análise de impacto de mudança e uma análise de fluxo de controle em 100 versões de 16 programas escritos em C#. A partir dos dados coletados, este trabalho identificou os elementos do trade-off entre manutenibilidade e robustez.

**Palavras-chave:** Sistemas de Software. Qualidade. Tratamento de Exceções. Mecanismos de Tratamento de Exceções. Robustez. Manutenibilidade.

# ABSTRACT

Mainstream programming languages provide built-in exception handling mechanisms to support robust and maintainable implementation of exception handling in software systems. Most of these modern languages, such as C#, Ruby, Python and many others, are often claimed to have more appropriated exception handling mechanisms. They reduce programming constraints on exception handling to favor agile changes in the source code. These languages provide what we call maintenance-driven exception handling mechanisms. It is expected that the adoption of these mechanisms improve software maintainability without hindering software robustness. However, there is still little empirical knowledge about the impact that adopting these mechanisms have on software robustness. This work addresses this gap by conducting an empirical study aimed at understanding the relationship between changes in C# programs and their robustness. In particular, we evaluated how changes in the normal and exceptional code were related to exception handling faults. We applied a change impact analysis and a control flow analysis in 100 versions of 16 C# programs. The results showed that: (i) most of the problems hindering software robustness in those programs are caused by changes in the normal code, (ii) many potential faults were introduced even when improving exception handling in C# code, and (iii) faults are often facilitated by the maintenance-driven flexibility of the exception handling mechanism. Moreover, we present a series of change scenarios that decrease the program robustness.



**Keywords:** Software System. Quality. Exception Handling. Exception Handling Mechanisms. Reliability. Maintainnability.

# LISTA DE FIGURAS

# LISTA DE TABELAS

# LISTA DE ABREVIATURAS E SIGLAS

CCI - Common Compiler Infrastructure

EH - Exception Handling

EHC - Exception Handling Context

GUI - Interface Gráfica com o Usuário

IEEE - Institute of Eletrical and Eletronic Engineers

OO – Orientação a objetos

RAIL - Runtime Assembly Instrumentation Library

V&V – Validação e verificação

# SUMÁRIO

# 1. INTRODUÇÃO

Ao passo que avança a engenharia de software, a busca por desenvolver softwares mais robustos cresce. Situações consideradas “anormais” podem ocorrer durante o fluxo de execução de sistemas de software, tendo o desenvolvedor o papel de criar fluxos alternativos, ou excepcionais, para que o sistema retorne a um estado de normalidade. No entanto, projetar sistemas que conciliem confiabilidade e manutenibilidade é particularmente difícil [Parnas 1975; Yang 2003].

Confiabilidade diz respeito a capacidade de um sistema de software de entregar as suas funcionalidades sob condições normais e erroneas [Randell 1978; Lee 1990; Laprie 2004]. Robustez é um importante fator de confiança que caracteriza a capacidade de um sistema para reagir a falhas de componentes. Um sistema de software robusto precisa reagir à manifestação de erros em seus módulos e evitar falhas no sistema [Lee 1990]. Dessa forma, a robustez de um sistema de software contribui para a confiabilidade do mesmo de forma direta, pois implica em garantias da entrega de suas funcionalidades.

Manutenibilidade, por sua vez, refere-se à facilidade com a qual um produto de software pode ser modificado para correção de defeitos, para atender novos requisitos, para tornar mais fácil futuras manutenções, ou adaptação para mudanças de ambiente [IEEE 1990]. Em [Canfora and Cimitile 2000] há um alerta para a necessidade de se enxergar manutenibilidade como característica de projeto desde as primeiras decisões, pois normalmente é vista como uma demanda apenas após a entrega do software. Manutenção de software é mais do que corrigir erros, especialmente porque ela possui impacto sobre a qualidade e os custos dos sistemas em uso. Vários problemas técnicos e gerenciais contribuem para os custos de manutenção de software. Segundo [Canfora and Cimitile 2000], entre correções e melhorias, manutenção de software consome entre 60% e 80% do custo total do ciclo de vida de um produto de software. Portanto, construir softwares manuteníveis impacta diretamente nos custos totais de um software.

Segundo [Lee 1990], o comportamento tolerante a falhas introduz uma complexidade adicional a sistemas de software tornando mais desafiadora a realização da atividade de manutenção. Dessa forma, configura-se um desafio: desenvolver softwares confiáveis que também sejam fáceis de modificar e evoluir. Sistemas de software confiáveis devem, portanto, ser bem estruturados, a fim de controlar uma complexidade adicional e simultaneamente conservar-se manutenível.

## 1.1 Problema

A fim de incorporar uma clara separação entre comportamento de tratamento de erro e comportamento normal, os mecanismos de tratamento de exceção (*Exception Handling Mechanisms - EHMs*) [Parnas and Wurges 1976] foram originalmente concebidos. Tais mecanismos buscam facilitar a compreensão do programa e manutenção de ambos os comportamentos, normais e excepcionais, apoiando uma separação explícita entre eles. Essa separação tem como objetivo evitar que as mudanças no comportamento normal causem modificações não intencionais no comportamento excepcional e vice-versa [Parnas and Wurges 1976]. O tratamento de exceções é um dos mecanismos mais utilizados para implementar sistemas robustos [Gorbenko et al. 2008]. Ele está embutido na maioria das linguagens de programação, tais como Java, C#, Ruby, Python e C++. Essas linguagens oferecem abstrações para encapsular as condições excepcionais e construções próprias para lidar com elas durante a execução do programa, tanto na detecção quanto no tratamento destas condições. Neste contexto, exceções podem ser lançadas por quaisquer componentes que compõem uma aplicação e há construções próprias para indicar o comportamento a ser executado quando uma exceção é capturada, configurando assim o comportamento excepcional do software.

O projeto de mecanismos de tratamento de exceções em várias linguagens modernas, muitas vezes favorece a flexibilidade, a fim de facilitar a manutenção do software. Em outras o foco está na confiabilidade, provendo mecanismos de tratamento de exceções mais rígidos. Denominamos tais propostas de mecanismos de tratamento de exceções *dirigidos a manutenção* e mecanismos de tratamento de exceções *dirigidos a confiabilidade*, respectivamente.

Linguagens de programação como C# [Williams, M. 2002], C++ [Stroustrup, B. 1994], Ruby[Matsumoto, Yukio and K. Ishituka. 2002] e Python [van Rossum, Guido. 2007], utilizam EHMs menos rígidos que tem o objetivo de facilitar a modificação e reutilização de comportamento normal. Para atingir este objetivo, desenvolvedores são motivados a representar o mínimo possível as propriedades do comportamento excepcional. Esses mecanismos, em particular, não impõem declarações de interfaces excepcionais em assinaturas de métodos tornando mais fácil alterar a estrutura de manipulação de exceção de acordo com as exigências crescentes do projeto. Apesar da ausência de interfaces excepcionais, verificações de confiabilidade ainda são suportadas em tempo de execução.

Portanto, seus criadores afirmam que esses mecanismos permitem que os desenvolvedores encontrem um melhor custo-benefício entre robustez e facilidade de manutenção [Harrold et al. 2010].

No entanto, há pouco conhecimento experimental sobre como os programas que utilizam estes mecanismos de tratamento de exceção *dirigidos a manutenção* evoluem ao longo do tempo. Seu impacto e efeitos colaterais sobre a robustez e facilidade de manutenção em um projeto de software são atualmente desconhecidos. Cenários de programação propensos a falhas em alterações nos códigos normal e excepcional precisam ser melhor compreendidos quando um mecanismo de tratamento de exceção dirigidos a manutenção é empregado. Os estudos empíricos existentes focam apenas na investigação de padrões para lidar com exceções em linguagens de programação sem mecanismos de tratamento de exceções [Yang 2008; Alfredo et al. 2007] ou facilidades de tratamento de exceção dirigidos a confiabilidade suportados, por exemplo, por Java e seus dialetos [Robillard and Murphy 2003][Fu and Ryder 2007][Coelho et al. 2008][Marinescu 2011].

## 1.2 Objetivos

Esta dissertação tem como objetivo principal preencher essa lacuna, apresentando um estudo empírico com o objetivo de compreender a relação entre as mudanças no comportamento normal e excepcional e seu impacto sobre a robustez. Além disso, o trabalho apresenta os seguintes objetivos específicos:

- Identificar as principais construções de linguagem de programação que caracterizem os mecanismos *dirigidos a manutenção* e mecanismos de tratamento de exceções *dirigidos a confiabilidade*;
- Identificar cenários de mudança no código normal e excepcional;
- Identificar como cenários de mudança no código normal afetam a robustez;
- Identificar como cenários de mudanças no código excepcional afetam a robutez;

## 1.3 Organização do Texto

Além deste capitulo introdutório, esta dissertação apresenta mais 5 capítulos, organizados como descrito a seguir: o capítulo 2 apresenta a fundamentação teórica, o

capítulo 3 discute as abordagens existentes para mecanismos de tratamento de exceções e no capítulo 4 é apresentada a descrição do estudo e o processo de coleta de dados. O capítulo 5 apresenta os dados e os analisa detalhadamente.

# 2. FUNDAMENTOS

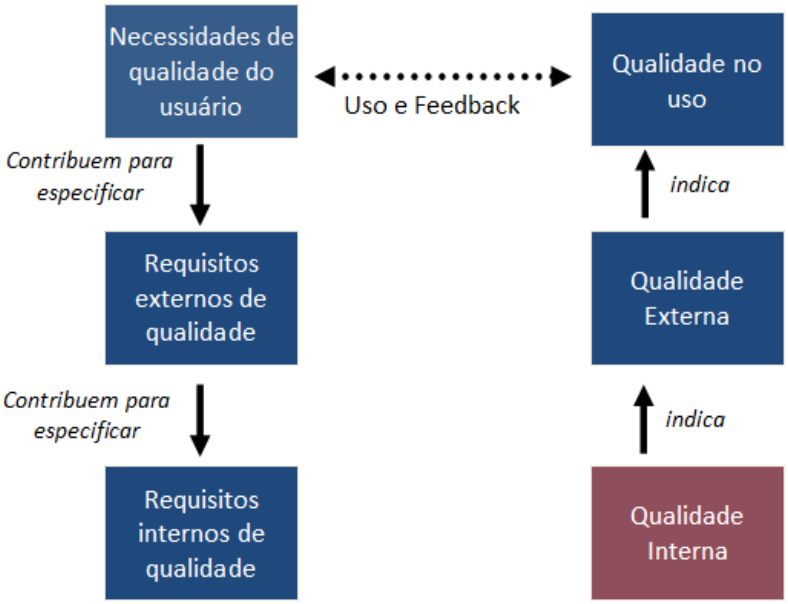
Este capítulo apresenta os temas abordados nesta dissertação. Na seção 2.1 são apresentados conceitos de Qualidade de Software. Em seguid*a* a seção 2.2 apresenta conceitos de Análise Estática. A seção 2.3 aborda detalhes da plataforma .NET. Os conceitos de defeito, erro e falha são desenvolvidos na seção 2.4 enquanto na seção 2.5 os mecanismos de tratamento de exceção são detalhados em seus mecanismos e desafios internos.

## 2.1 Qualidade de Software

A qualidade dos produtos de software é um fator essencial para o sucesso das empresas que desenvolvem software. Alguns tipos de software, tais como, sistemas de tempo real, software para equipamentos médicos, software controlador de tráfego aéreo, ou qualquer software que lide com aplicações críticas, onde uma falha produza consequências graves, necessitam de atenção especial ao requisito qualidade. Existem várias definições sobre qualidade. Garvin (1984) reuniu várias dessas definições e as categorizou em cinco abordagens de qualidade, a saber: (i) transcendental, pela qual qualidade é algo que se percebe, mas não se define; (ii) do usuário, pela qual qualidade é atingir os objetivos especificados pelo consumidor; (iii) da manufatura, qualidade é seguir precisamente as especificações do projeto; (iv) do produto, vê qualidade como um conjunto mensurável e preciso de características, que são requeridas para satisfazer o consumidor; e (v) do valor, pela qual qualidade está relacionada a quanto o cliente está disposto a pagar pelo produto. Slack, Chambers e Johnston (2002) conciliaram essas abordagens em sua definição de qualidade: *Qualidade é a consistente conformidade com as expectativas dos consumidores*.

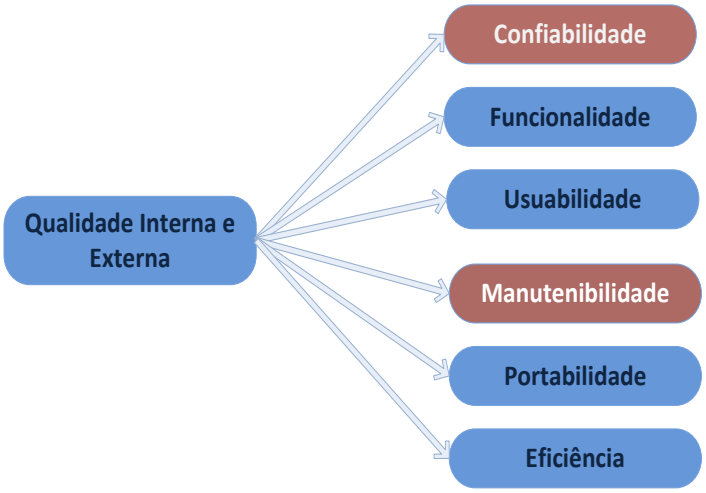
Essa definição pode ser relacionada com o modelo de qualidade do padrão ISO/IEC 9126-1:2001 [ISO, 2001] na forma de que na qualidade interna dos produtos, a palavra *conformidade* indica que é preciso seguir processos e especificações durante a fase de produção, e essa qualidade interna irá indicar a qualidade externa, que está relacionada ao produto em execução, que por sua vez levará a qualidade no uso, indicando se as expectativas dos consumidores foram ou não atendidas. A Figura 1 ilustra a relação entre essas visões de qualidade, bem como que cada visão gera necessidades e requisitos específicos que são refinados ao longo do ciclo de vida de produção e uso dos produtos de software.

**Figura 1 -** Diferentes visões de qualidade. Adaptado [ISO. 2001]




Para cada visão de qualidade descrita na Figura 1, ISO (2001) definiu um conjunto de características e suas respectivas sub-características que podem ser utilizadas para medir o nível de qualidade do software. A Figura 2 ilustra as características das visões interna e externa e a Tabela 1 as descreve.

**Figura 2 -** Características de qualidade. Adaptado [ISO. 2001]



| Característica | Descrição |
|---|---|
| Confiabilidade | Medida da capacidade de um software de manter seu nível de desempenho dentro de condições estabelecidas por um dado período de tempo |
| Funcionalidade | Conjunto das funções especificadas e suas propriedades. As funções devem satisfazer as necessidades do usuário |
| Usabilidade | Medida de esforço de uso de um software por um usuário de um determinado perfil |
| Manutenibilidade | Esforço necessário para fazer alterações no software – manter o software |
| Portabilidade | Medida da facilidade do produto de software ser transferido para outro ambiente. |
| Eficiência | Relação entre o nível de desempenho do software e a quantidade de recursos utilizados. |

**Tabela 1 -** Características de qualidade do padrão ISO 9126 (ISO, 2001)

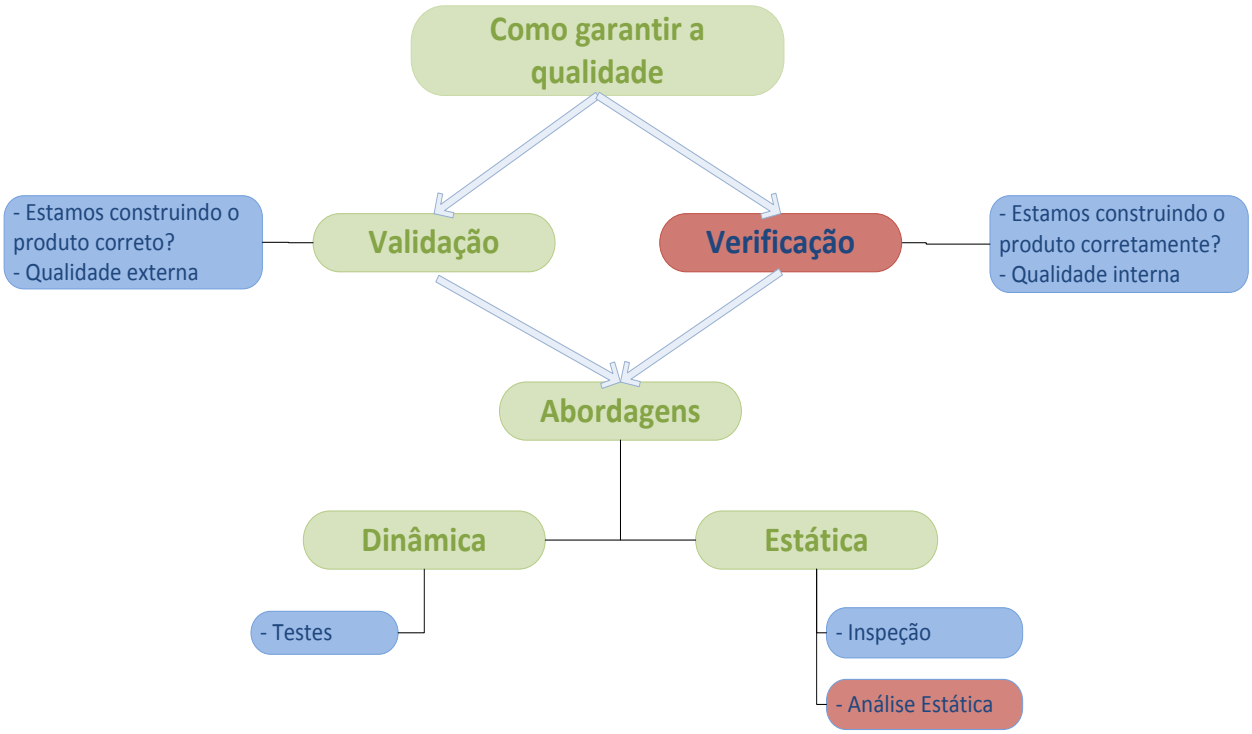
Existem outros modelos de qualidade propostos por acadêmicos e pela indústria. Al-Qutaish (2010) realizou um estudo comparativo entre cinco modelos de qualidade, a saber: McCall's, Boehm's, Dromey's, FURPS e ISO 9126. Embora cada modelo possua nomenclaturas, divisões e enfoques diferentes, o estudo chegou à conclusão que a única característica de qualidade em comum a todos eles foi à confiabilidade, demonstrando a importância desse atributo e a necessidade de processos e ferramentas que assegurem níveis aceitáveis de qualidade.

Atividades de validação e verificação (V&V) são utilizadas para esse fim, elas visam assegurar que tanto o modo como o software está sendo desenvolvido quanto o produto em si estejam em conformidade com a especificação e com as expectativas dos usuários. Boehm (1979) definiu que uma atividade de validação visa garantir que o produto atenda as expectativas dos usuários, já a atividade de verificação visa assegurar que o produto esteja em conformidade com as especificações. Relacionando essas atividades com ISO (2001) conclue-se que a atividade de validação está ligada a qualidade externa do produto, e a atividade de verificação está ligada a qualidade interna.

Essas atividades possuem duas abordagens: (i) estática, realizada através de inspeções manuais nos requisitos, diagramas de projeto e código-fonte, bem como com análises automatizadas do código-fonte; (ii) dinâmica, realizada através de testes de software, envolvendo a execução do software com dados de testes, afim de observar seu comportamento operacional, desempenho e confiabilidade. A abordagem estática pode ser utilizada em qualquer fase do ciclo de vida do software, e serve para verificar a correspondência entre o software e sua especificação, podendo não demonstrar se o software será útil operacionalmente. Para tal, é necessário utilizar a abordagem dinâmica, pois ela é

baseada em cenários de testes e análises de comportamento. A Figura 3 ilustra o relacionamento entre os conceitos explanados anteriormente sobre garantia da qualidade.

**Figura 3 -** Como garantir a qualidade




## 2.2 Análise Estática

Revisões e inspeções são tarefas humanas que requerem muita atenção para obter êxito. Além disso alguns cenários de testes podem ser difíceis de serem criados. Nesse contexto, uma técnica bastante utilizada durante a realização de atividades de validação e verificação é a análise estática [AYEWAH; PUGH, 2008, 2010; BESSEY et al, 2010].

> Qualidade de software está se tornando mais importante com a crescente dependência de sistemas de software. Existem diferentes formas de garantir qualidade de software, incluindo revisões de código e testes rigorosos. Defeitos de software, ou erro, podem custar às empresas quantias significativas de dinheiro, especialmente quando eles levam a falha de software .
> As ferramentas de análise estática proporcionam um meio para analisar o código sem a necessidade de executá-lo, ajudando a garantir maior qualidade de software em todo o processo de desenvolvimento. [Johnson et al. 2013]

A análise estática realizada por ferramentas automatizadas que verificam o software sem executá-lo. Landi (1992) a definiu como o processo de extrair informação sobre a semântica do código em tempo de compilação. Tal extração pode ser feita diretamente no código-fonte ou no código objeto (no caso da plataforma .NET, denominado *ILCode*). A análise direta do código-fonte irá refletir exatamente o que o programador escreveu. Alguns compiladores otimizam o código e outros podem inserir informações ocultas, como o compilador do JavaServer Pages, logo o código objeto poderá não refletir o código-fonte [CHESS; WEST, 2007]. Por outro lado, a análise do código objeto é consideravelmente mais rápida, o que é determinante em projetos com dezenas de milhares de linhas de código.

As ferramentas de análise estáticas podem ser classificadas em quatro grupos de funcionalidades [PFLEEGER; ATLEE, 2009]: (i) análise de código, faz verificação de erros de sintaxe, procura por construções com tendência a erros e itens não declarados; (ii) verificação estrutural, mostra relações entre os elementos, aponta possíveis fluxos do programa, detecta loops e partes do código não utilizadas; (iii) análise de dados, verificação da atribuição de valores, revisão das estruturas de dados e verificação da validade das operações; e (iv) verificação de sequencia, verificação da sequencia dos eventos e checagem da ordem das operações.

Além dessa classificação por funcionalidade, as ferramentas podem ser categorizadas de acordo com os problemas que elas se propõem a resolver. A Tabela 2 lista os diversos tipos de ferramentas de acordo com essa classificação [CHESS; WEST, 2007].

| Tipo | Descrição |
|---|---|
| Verificador de tipo | Visa verificar e fazer cumprir as restrições de tipo, por exemplo, associar uma expressão do tipo *int* a uma variável do tipo *short*. |
| Verificador de estilo | Verifica a conformidade do código-fonte com um estilo de programação pré-determinado, checando aspectos como espaço em branco, comentários, nomenclatura das variáveis, funções obsoletas, etc. |
| Entendimento do programa | Utilizada pelos ambientes integrados de desenvolvimento (*IDEs*) para ajudar o programador a entender o software. Pode ter recursos do tipo: localização de chamadas a um método, localização da declaração de uma variável global etc. |
| Verificação do programa | De acordo com uma especificação e um pedaço de código, tenta provar que o código é uma implementação fiel da especificação. |
| Localização de *bugs* | Aponta locais onde o programa poderá se comportar de uma forma diferente da intenção do programador. Tal comportamento é derivado de um conjunto de regras que descrevem padrões de código que geralmente irão resultar em *bugs*. |

**Tabela 2 -** Tipos de ferramentas de análise estática

Essas ferramentas possuem um aspecto intrínseco de imprecisão, elas apontam e advertem sobre possíveis defeitos e riscos que possam vir a ocorrer, ou seja, podem ocorrer casos de falso positivo e falso negativo durante as análises. Falso positivo ocorre quando a ferramenta encontra um problema que na verdade não existe. Já falso negativo ocorre quando o problema existe e não é reportado. Falsos positivos são indesejáveis, pois geram informações irrelevantes, que podem se misturar a problemas reais. Porém, do ponto de vista da confiabilidade, os falsos negativos são mais preocupantes pois a partir do resultado gerado pela ferramenta, não há como saber se ocorreu falso negativo ou não, gerando assim uma falsa sensação de segurança. A análise estática será tão mais precisa quanto menos falsos positivos e falsos negativos produzir, e será confiável se não produzir falsos negativos considerando as propriedades analisadas [CHESS; WEST, 2007].

Além disso, análise estática é um problema computacionalmente indecidível [LANDI, 1992], pois existem situações que são impossíveis de prever quando um algoritmo analisa outro algoritmo, por exemplo, o momento em que o algoritmo que está sendo analisado termina. Esse problema do término de um algoritmo foi identificado por Alan Turing na década de trinta e ele o denominou de *Halting Problem.* Segundo ele, a única forma garantida de saber o que um algoritmo irá fazer é executá-lo. Isso significa que se de fato um algoritmo não termina, a decisão sobre o término dele nunca será alcançada [CHESS; WEST, 2007]. Além desse problema, em 1953 foi exposto o teorema de *Rice*, que diz que análise estática não pode determinar perfeitamente qualquer propriedade não trivial de um programa [ARNOLD L. ROSENBERG, 2009].

Segundo [Johnson et al. 2013], há lacunas entre a prática dos desenvolvedores de software e os recursos fornecidos pelas ferramentas de análise estática atuais. Tais ferramentas geram alertas como resultado de suas análises, no entanto não oferecem informações suficientes acerca das medidas que poderiam ser adotadas pelos desenvolvedores. Grande parte dessas ferramentas não se integra com o fluxo de trabalho dos desenvolvedores, por exemplo, se um desenvolvedor submete seu código a um repositório várias vezes ao dia é mais provável que ele utilize uma ferramenta que execute uma análise a cada submissão de seu código. Essas dentre outras dificuldades confirmam, segundo [Johnson et al. 2013], que falsos positivos e a sobrecarga de atividades sobre os desenvolvedores, levam esses à insatisfação com as ferramentas de análise estática atuais.

Apesar desses problemas, na prática as ferramentas de análise estática produzem resultados úteis e ajudam na melhoria da qualidade dos produtos de software (AYEWAH; PUGH, 2008, 2010; BESSEY, 2010). Essas imperfeições não as impedem de ter um valor significativo e nem são um fator limitador de seu uso (CHESS; WEST, 2007). Dessa forma, alguns tipos de falhas foram catalogados e indicam formas de utilização comum das ferramentas de análise estática. A Tabela 3 lista alguns desses tipos de falhas.

| Tipo | Pode acontecer quando | Como detectar |
|---|---|---|
| *Race Conditions* | Mais de uma *thread* acessa uma mesma variável, sendo pelo menos um dos acessos de escrita e nenhum mecanismo previne esse acesso simultâneo. | Examinar todas as possíveis ordens de comandos das *threads* envolvidas. |
| *Array Bounds* | Ocorre acesso à posição inexistente do array, ou seja, quando o índice é menor que a primeira posição ou maior que a última. | Examinar se variáveis inteiras assumem valores menores, iguais ou maiores que o tamanho do array e se são usadas como índice, através da varredura de todos os caminhos possíveis. |
| *Buffer Overflow* | Ocorre ao copiar dados para um buffer que possui tamanho menor do que a quantidade de dados de entrada, podendo causar comportamento inesperado no sistema ou furo imperceptível na segurança. | Pode ser detectado de forma semelhante ao *Array Bounds*. |
| *Exception Handling* (vide item 2.4) | Não há tratamento de uma exceção lançada. | Examinar exceções lançadas sem bloco `try/catch`. |

**Tabela 3 -** Formas de utilização de ferramentas de análise estática

## 2.3 Plataforma .NET

Em meados de 2002, a Microsoft lançou oficialmente uma plataforma única para desenvolvimento e execução de sistemas, denominada Microsoft.NET, ou .NET *Framework.* A idéia principal é qualquer sistema desoftware escrito para .NET seja executado por qualquer dispositivo que possua o *framework* de tal plataforma, seja ele um computador com Windows ou Linux instalado, um smartphone ou até mesmo um celular. Esse ambiente é composto de ferramentas, *framework* para execução, linguagens de programação e biblioteca de classes que suportam a construção de aplicações desktop, sistemas distribuídos, componentes, páginas dinâmicas para Web e XML Web Services.

Capers Jones (1998) escreveu em seu livro que, pelo menos, um terço das aplicações de software foram implementadas utilizando duas linguagens de programação diferentes. Seguindo essa tendência, a plataforma .NET foi construída para suportar o paradigma multi-linguagem, ou seja, é possível trabalhar com várias linguagens no mesmo projeto e interagir entra elas. Isso ocorre devido ao fato do código ser compilado para uma linguagem intermediaria (*IL Intermediate Language),* e ser somente essa linguagem que o ambiente de execução conhece. A arquitetura dessa plataforma é dividida em camadas, que organiza o caminho que inicia na escrita do código e finaliza com a execução do software. A Figura 4 ilustra esse fluxo e exibe os componentes da arquitetura, a saber:

- **Linguagem de programação**

Não é toda linguagem de programação que pode ser compilada para a linguagem intermediária (Intermediate Language - IL). Ela deve ser compatível com duas especificações, a saber: (i) *CLS - Common Language Specification*, conjunto de recomendações que as linguagens devem seguir e que garante a interoperabilidade entre elas, (ii) *CTS Common Type System,* definição padrão de todos os tipos de dados disponíveis para a *IL*.

- **Metadata (metadados)**

Todo código compilado em .NET é auto-explicativo, ou seja, toda informação necessária para sua execução é armazenada dentro dele próprio na forma de *METATAGS*, fazendo com que o ambiente de execução não precise procurar essas informações no registro do sistema operacional. Algumas dessas informações são listadas a seguir: (i) descrição dos tipos utilizados (classes, estruturas, enumerações, etc), (ii) descrição dos membros (propriedades, métodos, eventos etc), (iii) descrição das referências a componentes externos (assembly) e (iv) versionamento e integridade do código na seção *manifest*.

- **Assembly**

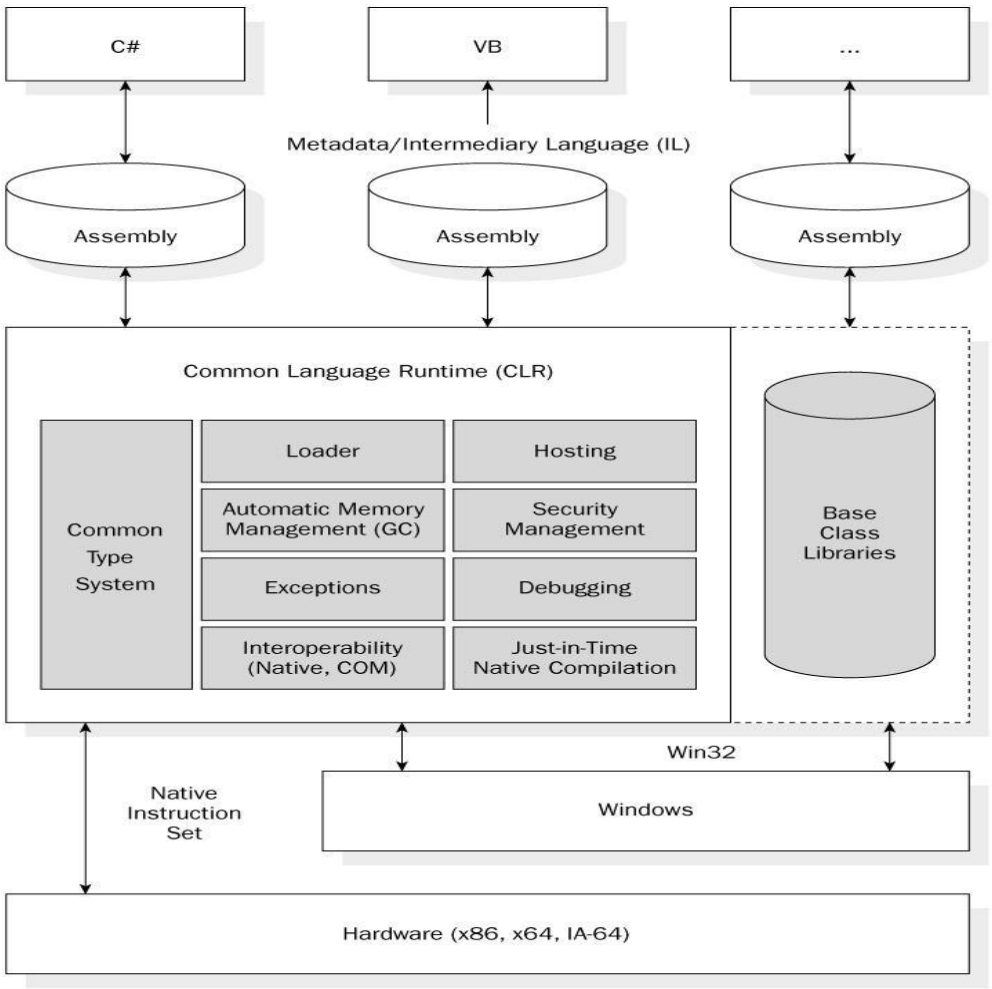
É a unidade de código física resultante da compilação para *IL*. Pode ser representado por um arquivo executável com extensão .EXE, ou por uma biblioteca de ligação dinâmica com extensão .DLL. Todo *assembly* é auto-explicativo através de seus metadados, conforme explicado no item anterior.

- **CLR – Common Language Runtime**

É o ambiente de execução das aplicações .NET, que funciona como uma máquina virtual que gerencia o relacionamento entre o programa e o sistema operacional. Entre suas responsabilidades estão: gerenciamento de memória, mecanismos de segurança e tratamento de exceções, integração com outras plataformas, por exemplo: COM, depuração, e a

compilação *Just-In-Timer (JIT)*. A *JIT* interpreta a *IL* do *assembly* e gera a linguagem de máquina na arquitetura do processador. Existem três tipos de *JIT*: (i) *Pre-JIT*, compila de uma só vez todo o código da aplicação que esta sendo executada e o armazena no cache para uso posterior, (ii) *Econo-JIT*, utilizado em dispositivos móveis onde a memória é um recurso precioso, sendo assim, o código é compilado sob demanda e a memória alocada que não esta em uso é liberada quando o dispositivo assim o requer, (iii) *Normal-JIT*, compila o código sob demanda e joga o código resultante em cache, de forma que esse código não precise ser recompilado quando houver uma nova invocação do mesmo método.

**Figura 4 -** Arquitetura da plataforma .NET




- **Biblioteca de classes (Base Class Library)**

Conjunto de classes para os mais variados propósitos, tais como: acesso a base de dados, gerenciamento de arquivos, gerenciamento de memória, serviços de rede, interface gráfica etc. Este é o principal recurso que possibilita o desenvolvedor criar os sistemas. Sua estrutura é organizada na forma de *namespaces*, baseada em uma hierarquia de nomes. É interessante ressaltar que a *base class library* é desenvolvida para a *IL*, consolidando o aspecto multi-linguagem da plataforma .NET.

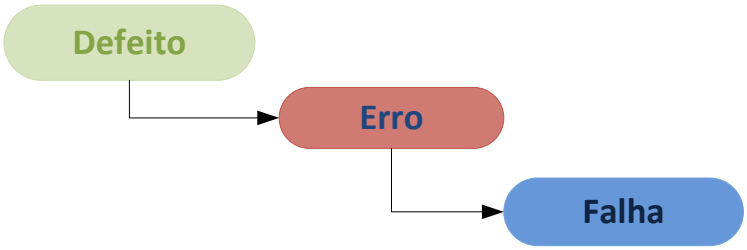
Por ter código compilável para uma linguagem intermediária, que será interpretado pelo ambiente gerenciado *(CLR)* através do processo *JIT*, a plataforma .NET permite ao desenvolvedor manipular e reescrever *assemblies* dinamicamente e em tempo de execução. Tal recurso é disponibilizado através da funcionalidade *Reflection*, que permite ao desenvolvedor [ROBINSON *et al*, 2004]: (i) listar os membros de uma classe, (ii) instanciar um novo objeto, (iii) executar os métodos de um objeto, (iv) pesquisar informações sobre uma classe ou assembly, (v) inspecionar os atributos personalizados de uma classe, (vi) criar e compilar um novo assembly e (vii) inserir ou remover instruções *IL* em métodos de uma classe. As principais classes que habilitam o uso do *Reflection* são: `System.Type`, `System.Reflection.Assembly` e `System.Reflection.Emit`.

## 2.4 Defeito, Erro e Falha

Um sistema de software consiste de um conjunto de componentes que interagem de acordo com o especificado em um projeto de software, a fim de atender as demandas do ambiente (LEE; ANDERSON, 1990). O projeto define como os componentes irão interagir e estabelece as conexões entre os componentes e o ambiente. Para ser confiável, um sistema de software precisa entregar o serviço esperado mesmo na presença de condições anormais. Para tal, ele recebe requisições de serviços e produz respostas que devem estar de acordo com as especificações ficando assim em um estado consistente. Quando por algum motivo esse estado sofre uma alteração que o torne inconsistente, é dito que ocorreu um erro [LEE; ANDERSON, 1990; LAPRIE; RANDELL, 2004]. Uma falha [LEE; ANDERSON, 1990; LAPRIE; RANDELL, 2004] é a manifestação de um ou mais erros e é de percepção externa, ou seja, pelos usuários do sistema. O problema que pode ter originado o erro, que foi manifestado através de uma falha é denominado defeito, que pode ter origem interna ou externa aos limites do software (LAPRIE; RANDELL, 2004), ou seja, problemas internos no

código-fonte ou através de interações errôneas, falhas de hardware, etc. A Figura 5 exibe a sequência da ocorrência do defeito até a manifestação da falha.

**Figura 5 -** Sequência entre defeito, erro e falha.




## 2.5 Tratamento de Exceções

O desenvolvimento de um software robusto requer a adição de código para detecção e tratamento de erros e a abordagem dos mecanismos de tratamento de exceções é uma das mais utilizadas pelas linguagens de programação modernas. Esta seção descreve os conceitos envolvidos em tais mecanismos.

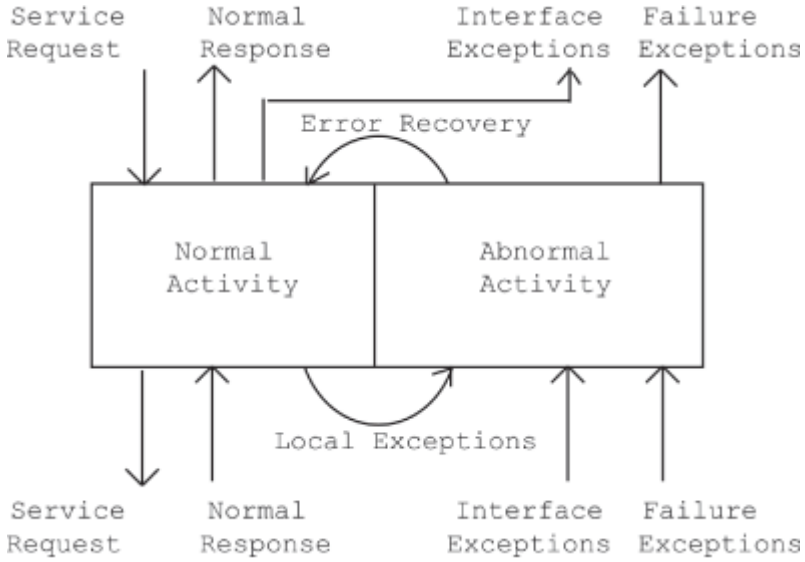
### 2.5.1 Exceções

A atividade de um sistema pode ser dividida em: normal e excepcional. A atividade normal corresponde à entrega do serviço que o responsável pela requisição espera, já a atividade excepcional provê as medidas necessárias para lidar com as falhas que causaram a exceção, portanto, se o sistema não conseguir responder uma requisição de serviço, ele irá retornar uma exceção. As exceções podem ser divididas em três categorias [LEE. ANDERSON., 1990]: (i) exceções de interface, que são sinalizadas em resposta a uma requisição que não está em conformidade com a interface especificada pelo componente; (ii) exceções de defeitos, que são sinalizadas quando o componente determina que por alguma razão não pode prover o serviço requisitado; e (iii) exceções internas, que são exceções geradas pelo próprio componente a fim de invocar suas medidas internas de tolerância a falhas. As exceções são geradas por um componente, porém podem ser sinalizadas entre componentes. Sendo assim, as exceções de interface e de falha podem se tornar exceções externas desde que as ações que irão lidar com ela estejam fora do componente que as

sinalizou. A estrutura e o relacionamento entre essas categorias de exceções foram organizados por Lee e Anderson (1990) em um componente ideal de tolerência a falhas (ver Figura 6).

Ao receber uma requisição de serviço o componente pode responder da seguinte forma: (i) resposta normal, se houver sucesso no processamento; (ii) lançamento de uma exceção de interface, se a assinatura do serviço for inválida; (iii) exceção interna, se ocorrer falha no processamento e o fluxo for desviado para a atividade excepcional do componente, e (iv) exceção de defeito, quando a atividade excepcional não conseguir voltar o componente para um estado consistente.

**Figura 6 -** Componente ideal de tolerância a falhas. (LEE; ANDERSON, 1990)




### 2.5.2 Modelos de Tratamentos de Exceções

A primeira definição de um modelo de tratamento de exceções foi feita por Goodenough (1975). Tal autor considera que exceções são meios de comunição entre a entidade que invoca a execução de uma operação e algumas condições que podem ocorrer no programa, tais como: erros durante a execução de uma rotina, resultados retornados por uma função, e determinados eventos (não necessariamente falhas) que ocorrem durante a execução do programa.

O gerenciamento das exceções que ocorrem em um sistema é definido como mecanismo de tratamento de exceção (*exception handling*). Tal mecanismo deve contemplar as seguintes atividades: (i) detecção de uma ocorrência de exceção, (ii) desvio do fluxo normal do

programa para o fluxo excepcional, (iii) localização do código de tratamento da exceção, e (iv) execução do código que irá lidar com a exceção. Após o tratamento da exceção, a aplicação deve retormar o fluxo normal.

O processo de instanciamento de uma exceção consiste na sinalização de exceções baseadas em declarações implícitas ou explícitas [GOODENOUGH 1975]. Declarações implícitas são fornecedidas por eventos nativos do hardware que executa o programa, tais como, divisão por zero, ponteiro nulo e overflow. Por outro lado, as declarações explícitas são definidas e sinalizadas pelos próprios desenvolvedores da aplicação. Os mecanismos de tratamento de exceções devem prover construções que permitam tal sinalização. O elemento que detecta o estado excepcional e gera a exceção é chamado de sinalizador de exceção (*exception signaler)*. Após a detecção da exceção, o sistema precisa desviar o fluxo normal do sistema para poder lidar com o problema. O processo interrompe a execução do programa e procura pelo bloco de código associado à situação excepcional. Tal bloco de código é chamado de tratador de exceção *(exception handler)* e é responsável por executar as medidas necessárias para o tratamento do problema. Dependendo do mecanismo utilizado, o tratador de exceção pode ser associado a uma instrução, um bloco, um método, um objeto, uma classe ou uma exceção [GARCIA *et al*. 2001]. A região protegida do código onde o tratador é implementado é chamada de contexto de tratamento de exceções (*Exception handling contexts - EHC)* e é utilizada para limitar o escopo de execução do tratador. Durante o processo de busca do tratador, a exceção pode ser propagada do nível em que se encontra para os níveis mais externos até que um tratador compatível seja localizado.

O controle do fluxo do programa depois que o tratador é executado é determinado pelo modelo de tratamento de exceções. Três modelos são referenciados pela literatura [ROBILLARD and MURPHY 2003]: (i) *termination model*, o escopo que gerou a exceção é destruído, e, se um tratador for encontrado e executado, o fluxo retoma a primeira unidade de código posterior ao tratador; (ii) *resumption model*, após a exceção ser tratada, o fluxo continua do ponto que a exceção foi gerada; (iii) *retry model*, quando a exceção é tratada, o bloco que gerou a exceção é finalizado e então repetido.

## 2.6 Mecanismos de Tratamento de Exceções

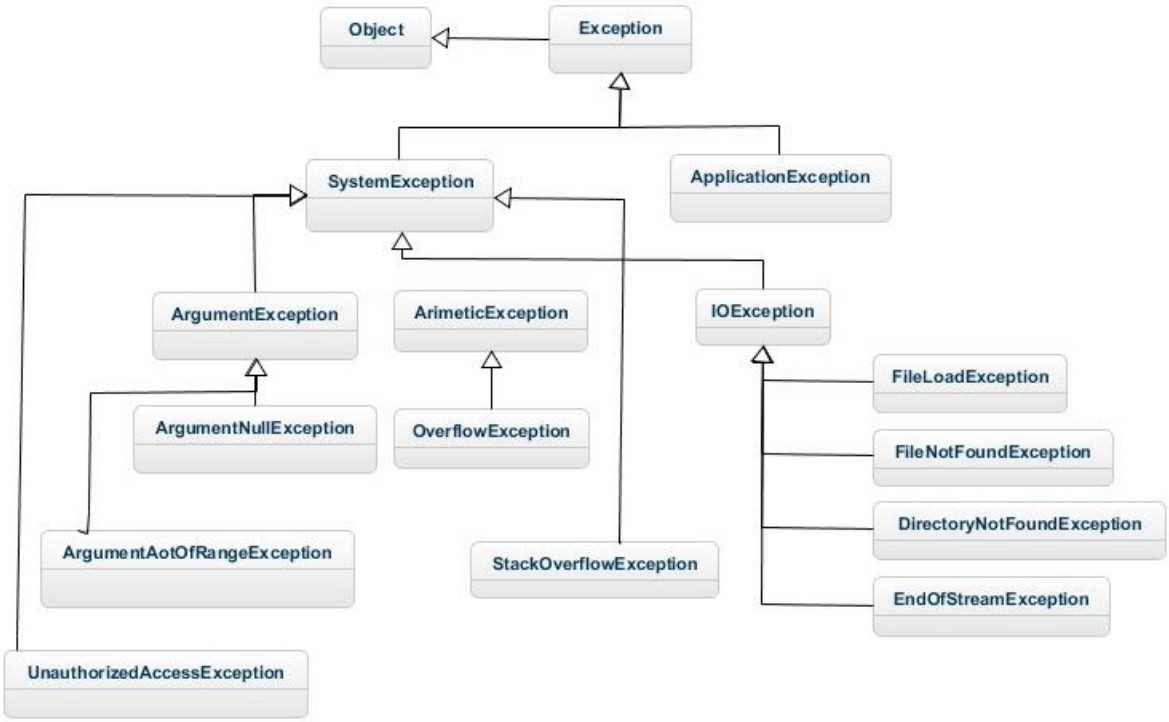
Grande parte das linguagens de programação modernas suportam os mecanismos de tratamento de exceção de forma nativa, através do *termination model* . A seguir serão descritos os modelos de tratamento de exceção de algumas linguagens de programação, como: C#, F#, J#, VB.NET, Java e C++. Essas linguagens são representativas, pois variam de linguagens orientadas a objetos (C#, VB.NET, J#, Java e C++) a uma linguagem funcional (F#). A hieraquia de classes e alguns aspectos relacionados ao projeto de tratamento de exceções, abordados por Garcia *et al* (2001), serão descritos nas subseções seguintes.

### 2.6.1 Hieraquia de classes

Na maioria das linguagens de programação OO, uma exceção é uma instância de uma classe que herda de um tipo excepcional, por exemplo, em .NET as exceções são subtipos de *System.Exception* [Rustan; Schulte, 2004], em Java as exceções são subtipos de *java.lang.Throwable* [Robillard and Murphy 2000]. Tal objeto contém informações que ajudam a rastrear a origem do problema [Robinson *et al*, 2004]. Embora o desenvolvedor possa criar suas próprias classes de exceção, as linguagens possuem número grande de classes pré-definidas para os mais variados tipos de erros. A Figura 7 e a Figura 8 ilustram a hierarquia de algumas dessas classes do .NET Framework e da plataforma Java, respectivamente. A classe genérica `System.Exception` (Figura 7) descende de `System.Object` que é a classe base de todas as classes do .NET. Nessa hierarquia é importante ressaltar as duas classes a seguir:

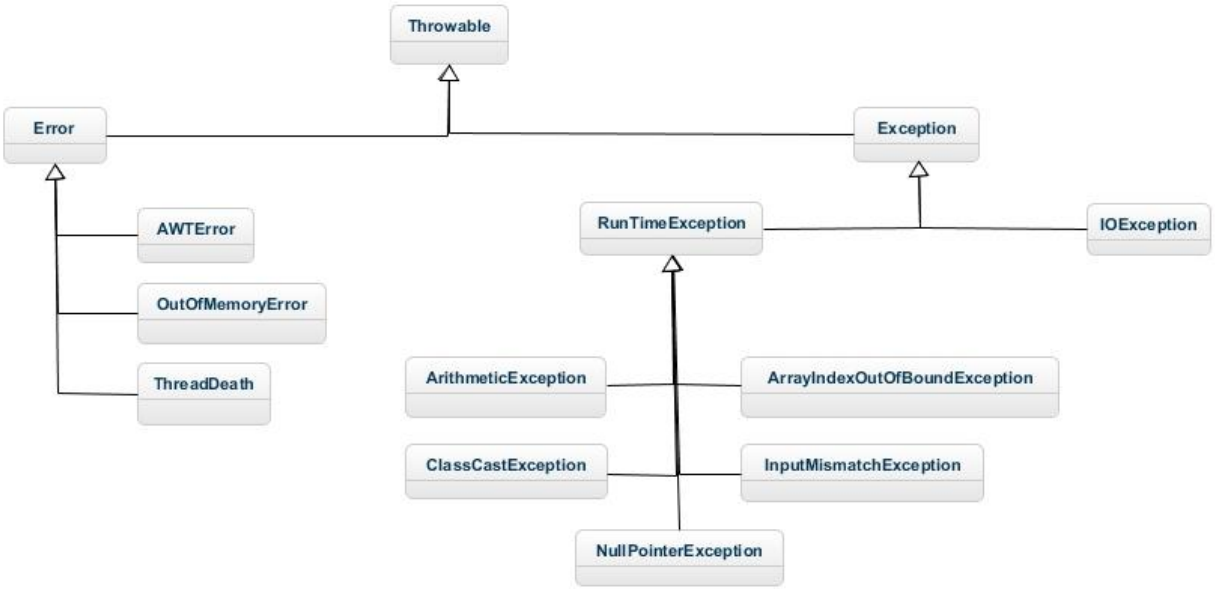
- `System.SystemException` – Esta classe é para exceções que são lançadas pelo .NET Runtime. Por exemplo, `StackOverflowException` será lançada pelo .NET Runtime na detecção do estouro da pilha. Por outro lado, o desenvolvedor pode lançar a exceção `ArgumentException` ou suas subclasses em seu próprio código, caso se detecte que um método foi chamado com argumentos inválidos;
- `System.ApplicationException` – Esta é a classe base para todas as classes de exceção definidas pelo desenvolvedor, ou seja a definição de exceções próprias do domínio da aplicação deverão descender desta classe.

**Figura 7 -** Hierarquia de exceções no .NET *Framework* [Robinson *et al*, 2004]




Essa representação permite a ocorrência de situações nas quais uma exceção de um subtipo é capturada por um supertipo. Essa situação é denominada *subsumption* [Robillard; Murphy, 2003], ou seja, um objeto de um subtipo pode ser atribuído a uma variável declarada como sendo de seu supertipo. Nesse caso, quando uma exceção é sinalizada e não existe um tratador específico, ela pode ser capturada por um tratador que trata um supertipo da mesma. Por exemplo, com base na Figura 8, se uma exceção do tipo ArithmeticException for lançada e um tratador do tipo RunTimeException capturar essa exceção ocorre uma subsunção pois o tipo ArithmeticException herda de RunTimeException e por isso chamamos esse fluxo de *subsumption flow*.

**Figura 8 -** Hierarquia das principais exceções na plataforma Java.




| Propriedade | Descrição |
|---|---|
| Message | Texto que descreve a exceção |
| HelpLink | Link para um arquivo de ajuda sobre a exceção |
| Source | Nome da aplicação ou do objeto que causou a exceção |
| StackTrace | Prover detalhes da pilha de chamadas dos métodos, a fim de auxiliar no rastreamento do método que lançou a exceção. |
| TargetSite | Objeto .NET *Reflection* que descreve o método que lançou a exceção |
| InnerException | Objeto Sytem.Exception que causou a exceção |

**Tabela 4 -** Atributos da classe System.Exception.

### 2.6.2 Interface de Exceção

A interface de exceção é utilizada em algumas linguagens de programação para explicitar que exceções podem ser lançadas por um método [Lang; Stewart, 1998]. Tal recurso é útil na medida em que o desenvolvedor pode pensar seu código para, ao chamar um método, estar preparado para as condições excepcionais que podem ocorrer durante a execução do programa. Sendo assim, no âmbito das linguagens de programação as exceções

são divididas em: (i) exceções checadas, que precisam ser declaradas na assinatura do método; e (ii) exceções não-checadas, que não precisam ser declaradas.

Interface de exceção não é suportada em C#, F# e VB.NET. Para essas linguagens, os desenvolvedores precisam examinar a implementação e documentação do método para identificar que exceções podem ser propagadas. Cabral, Sacramento e Marques (2007) fizeram uma extensa análise sobre o uso da documentação das exceções em aplicações .NET. Os autores concluíram que 87% das exceções lançadas não foram documentadas. Sendo assim, nas linguagens citadas acima, a documentação sobre as exceções provê um suporte limitado que ajuda os desenvolvedores a determinar se uma chamada a um método está lançando uma exceção.

C++, por sua vez, adota a abordagem opcional para interface de exceção. Essa abordagem oferece a palavra reservada `throw` para definir interfaces de exceção para os métodos. Métodos com a cláusula `throw` só podem propagar exceções listadas na interface. Entretanto, se nenhuma exceção for listada na assinatura do método, ele poderá lançar qualquer exceção.

Uma abordagem híbrida é utilizada por Java e J# através do suporte a exceções checadas e não-checadas. O compilador força as exceções checadas a serem associadas a um tratador ou definidas explicitamente na interface de exceção. Por outro lado, exceções não-checadas não obrigam que o programador nem associe a um tratador nem especifique a interface de exceção. Para separar exceções entre checadas e não-checadas, Java e J# definem uma hierarquia de tipos de exceções que caracteriza três grupos semânticos e funcionais: *errors*, exceções de *runtime* e exceções *checked* (exceções checadas ou verificadas). *Errors* e exceções de *runtime* não são checadas pelo compilador e não precisam ser declaradas na interface de exceção.

### 2.6.3 Vinculação de Tratadores

Contexto de tratamento de exceções *(EHC)* são regiões de código do programa ás quais os tratadores são vinculados. Um EHC pode ter um conjunto de tratadores associados, dentre os quais um é escolhido em tempo de execução quando uma exceção é lançada dentro do contexto. Existem diversos tipos de contexto de tratamento de exceções [Garcia *et al*, 2001], tais como declaração, bloco, método, objeto, classe ou exceção. Todas as linguagens analisadas suportam somente o EHC do tipo bloco (*block*). Um bloco é definido por um

construtor especial chamado *try* que delimita o conjunto de instruções associadas ao tratador. O bloco *try* inclui uma instrução composta, chamada de cláusula `try`, e uma lista de tratadores de exceção associados. A Tabela 5 descreve como os tratadores de exceção são especificados em algumas linguagens que implementam mecanismos de tratamento de exceções.

| Linguagem | Bloco EHC |
|---|---|
| J# | try {S} catch (E1 x) {T}<br>catch (E2 x) {T} |
| C# | try {S} catch (E1 x) {T}<br>catch (E2 x) {T} |
| C++ | try {S} catch (E1 x) {T}<br>catch (E2 x) {T} |
| VB.NET | try {S} Catch x As E1[When c] T<br>Catch x As E2 T End Try |
| F# | try S With \| :? E1 -> T<br>\| :? E2 -> T<br>\| :? C -> T |
| Java | try {S} catch (E1 x) {T}<br>catch (E2 x) {T} |

**Tabela 5 -** Vinculação de tratadores de diferentes linguagens de programação

### 2.6.4 Ligação de Tratador

Ligação de tratador determina como o mecanismo de tratamento de exceções procura o tratador. [Garcia *et al*. 2001] e [Lang. 1998] elencam três abordagens para ligação: *static*, *dynamic* e *semi-dynamic*. Todas as linguagens analisadas neste trabalho suportam a abordagem *semi-dynamic* [Garcia *et al*. 2001]. Nessa abordagem o mecanismo de tratamento de exceções executa uma procura por tratadores levando em conta os tratadores vinculados estaticamente ao EHC. Na sintaxe descrita na Tabela 5, o código em *S* pode gerar uma exceção. Quando uma exceção é lançada, cada padrão *x* é comparado com o tipo de exceção *En*, e para a primeira comparação bem sucedida o tratador respectivo é executado. Se nenhum padrão for encontrado, a exceção será propagada dinamicamente através da pilha de chamadas até um tratador ser localizado. Se nenhum tratador for localizado, o mecanismo de tratamento de exceções propaga uma exceção genérica ou finaliza o programa.

Adicionalmente, linguagens como F# e VB.NET suportam filtros de exceção que possibilita ao desenvolvedor construir uma cláusula `catch` condicional. A Figura 9 ilustra a

utilização de um filtro de exceção através do construtor `When` em VB.NET. Nesta figura, o tratador somente será executado quando a exceção *e* for um subtipo de `IOException` e se a condição (*filename* <> *String.Empty*) for verdadeira. Se a condição for falsa, o sistema irá continuar a procurar por tratadores através da pilha de chamadas.

**Figura 9 -** Exemplo de uso de filtro em VB.NET

```
Dim namefile As String = "c:\ex.txt"

 Try
   Dim fs As New IO.FileStream(namefile,FileMode.CreateNew)
 Catch e As IOException
    When namefile <> String.Empty
      Console.Write("The file already exists!" & vbCrLf)
      Throw New IOException("That file already exists on the hard
drive!" & vbCrLf)
 Catch e As Exception
      Console.Write("The message was : " & e.Message & vbCrLf)
      Console.Write("The stacktrace was : " & e.StackTrace & vbCrLf)
  End Try
```

# 3. ABORDAGENS DE TRATAMENTO DE EXCEÇÕES

A Engenharia de Software e as linguagens de programação têm evoluído no sentido de aumentar e facilitar o reuso de componentes de software. Entretanto, projetar sistemas manuteníveis é um processo árduo quando confiança é uma exigência. Segundo [RANDELL B., LEE, P. and TRELEAVEN, P. C. 1978], confiabilidade diz respeito à capacidade do software de prover suas funcionalidades, seja em condições normais ou errôneas.

Confiança possui como característica essencial a robustez, que é a capacidade de reação, do software, a falhas de componentes [LAPRIE, J. C. and RANDELL, B. 2004]. Um software robusto deve tratar os erros dos seus respectivos módulos impedindo que esses erros se tornem falhas de sistema [LEE, P. A. and ANDERSON, T. 1990]. Códigos referentes a comportamentos excepcionais normalmente acrescentam complexidade aos sistemas de software, tornando-os mais difíceis de manter [CRISTIAN, F. 1989; ROBILLARD, M. P. and MURPHY, G. C. 2003; R. MILLER and A. TRIPATHI. 1997].

Depois de analisar o estado da arte concluímos que os modelos de tratamento de exceções impõem um "trade-off" entre manutenibilidade ou confiabilidade. Os mecanismos de tratamento de exceção existentes tornam difícil a implementação de um tratamento de erro robusto e modular na presença de mudanças. Decisões de projeto específicas, tais como interfaces de exceção obrigatórias, normalmente favorecem a confiabilidade em detrimento da manutenibilidade. Há características, no entanto, que produzem o efeito contrário, tais como não suportar interface de exceção obrigatória. Essa característica favorece a manutenibilidade em detrimento de confiança. As sessões 3.1 e 3.2 analisam os elementos deste trade-off imposto pelas principais características dos mecanismos de tratamento de exceções.

## 3.1 Mecanismos de Tratamento de Exceção Dirigidos a Manutenção

A manutenibilidade do código de tratamento de erros é em grande parte dependente do suporte fornecido por um mecanismo de tratamento de exceções à modularidade e reusabilidade. A separação de interesses é o princípio fundamental para alcançar um tratamento de exceção modular e reutilizável. De fato, um fator de motivação fundamental para a concepção de mecanismos de tratamento de exceção foi apoiar a separação explícita entre o comportamento normal e excepcional. O objetivo era permitir a modificação

independente e reutilização do código normal e excepcional em um sistema de software. De acordo com [Parnas. 1976], quanto maior o grau de separação de comportamento excepcional suportada, maior o grau de reutilização e variabilidade alcançado. Essa separação permite estender e remover tratadores de exceção sem interferir na execução do comportamento normal.

Inúmeros artigos, como [F. Castor Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira. 2006; M. LIPPERT and C. V. LOPES. 2000; R. DE LEMOS and A. B. ROMANOVSKY. 2001] apontam para a necessidade de separação entre código normal e código excepcional. O que se pode perceber é que, com o objetivo de prover essa separação de uma maneira clara, os mecanismos de tratamento de exceção foram projetados para facilitar a compreensão e a manutenção de ambos os comportamentos, promovendo uma separação explícita entre eles [Goodenough, J. B. 1975]. Essa separação visa evitar que alterações no código normal causem alterações não intencionais no comportamento excepcional e vice-versa.

Reuso do comportamento normal também é fortemente dependente do grau de separação de interesses. Reduzir o embaraço e espalhamento de código excepcional pode resultar em uma dependência mais fraca entre os módulos. Dependências mais fracas, por sua vez, permitem que o código normal possa ser reutilizado de uma aplicação para outra, independentemente das diferenças entre as estratégias de tratamento de erros. As subseções seguintes descrevem como algumas decisões de projeto de mecanismos de tratamento de exceção promovem ou impedem a separação de interesses, a reutilização e a variabilidade tanto de código de manipulação de erro quanto código normal. Adicionalmente, para cada decisão de projeto, definimos a melhor escolha para apoiar o tratamento de exceção dirigido a manutenção.

### 3.1.1 Interface de Exceção

Os meios convencionais para expressar as interfaces de exceção são geralmente prejudiciais à reutilização e variabilidade. O suporte à interface de exceção foi originalmente apresentado para melhorar a confiabilidade, permitindo a comunicação de erro entre os módulos. A fim de alcançar este objetivo, os projetistas de mecanismos de tratamento de exceções decidiram estender a interface do comportamento normal (assinatura do método),

com a definição de uma interface de comportamento excepcional. Como consequência, isso cria diversos problemas novos documentados na literatura.

Em primeiro lugar, a interface de excepção deve obedecer regras de conformidade da linguagem de programação alvo. Isto, por sua vez, limita a reutilização de software por meio de restrição de especialização conceitual do comportamento normal. Miller e Tripathi apontam que o comportamento normal e excepcional geralmente evoluem de maneiras diferentes. O primeiro evolui mantendo as semelhanças com a superclasse, enquanto o segundo pode exigir a introdução de novas exceções que não são semanticamente compatíveis com os supertipos. Para esses casos, pode ser impossível reutilizar simultaneamente um módulo e obedecer as regras de conformidade de exceção.

Segundo, interface de exceção dificulta a variabilidade e manutenção do software. Se uma nova exceção é adicionada à interface de exceção de um método no final da cadeia de chamadas, as interfaces de exceção de todos os métodos através dos quais a nova exceção será propagada também necessitará ser atualizada. Para programas com longas cadeias de chamadas de métodos, essa é uma tarefa demorada e propensa a erros, causando graves problemas para a evolução da arquitetura do sistema. Segundo [Reimer and Srinivasan 2003], outro problema é lidar com um indesejado fenômeno denominado "*swallowed exceptions*" (exceções engolidas), no qual o programador insere um tratador vazio, apenas para satisfazer a checagem do compilador que exige tratamento para a exceção declarada na interface de exceção. Conclui-se, portanto, que qualquer mecanismo de tratamento de exceção que utilize a abordagem de declaração de exceções nas assinaturas dos métodos terá sua manutenibilidade afetada e possivelmente o surgimento de "*swallowed exceptions*".

### 3.1.2 Vinculação de Tratadores

Certamente, a forma como tratadores, ou manipuladores, são ligados aos EHCs tem um efeito direto sobre a separação de interesses e afeta o grau de manutenibilidade de um software. Soluções convecionais para vincular tratadores a declarações e blocos possuem o efeito colateral de gerar entrelaçamento de código excepcional e normal no corpo do método. Adicionalmente, o código ilustrado na Figura 10 apresenta outra situação indesejável na qual dois métodos implementam estratégias de tratamento de exceção idênticas (Linhas 8-10 e 21-23). Em linguagens como Java, C++ e C# muitas vezes não é possível implementar um único manipulador e associá-lo com mais de um método, para evitar a duplicação de código. A fim

de implementar um sistema de modo que o seu comportamento normal e seu código de tratamento de erro possa ser reutilizado de forma independente, esse último deve ser definido por meio de métodos separados invocados pelos tratadores de exceção. Essa solução não é a ideal, uma vez que impõe uma sobrecarga de implementação (devido à implementação de blocos try-catch) e exige que o código real de manipulação de erro seja construído fora dos tratadores de exceção.

Algumas linguagens de programação de "pesquisa", tais como Guide e Extended Ada, promovem a reutilização de código normal entre os aplicativos e a separação de interesses entre os códigos de tratamento de erro e normal, apoiando EHCs alternativos, tais como métodos, classes, objetos e exceções. Isso é claramente desejável quando todo o código de tratamento de exceções está dentro de tal contexto, por exemplo, na mesma classe certas exceções são, sempre, tratadas da mesma forma. No entanto, essas linguagens não fornecem mecanismos simples de reutilização de código excepcional dentro do mesmo sistema. Por exemplo, não há nenhuma maneira direta para criar um tratador para uma determinada exceção e associá-lo a um subconjunto dos métodos de uma classe sem: (i) duplicar o código do tratador, ou (ii) incorrer na sobrecarga de tratadores implementados para simplesmente delegar as tarefas de manuseio de um método especializado. Portanto, nesse sentido, essas linguagens não são tão diferentes de Java, C# e C++. Por isso uma abordagem mais desejável seria a de simplesmente "desligar" a estratégia de tratamento de erro ligada ao método e "plugar" a nova utilizando mecanismos da linguagem fornecidos especificamente para tratar exceções. No entanto, isso não é atualmente possível nas linguagens de programação convencionais.

**Figura 10 -** Atributos modularidade e reuso (Código do Mobile Media)

```
1  class RecordStoreMechanism implements PersistenceMechanism{
2    public void addRecord(byte[] data)
3      throws PersistenceMechanismException
4      {
5      try{
6          . . .
7          mediaInfoRS.addRecord(data, offset, data.length);
8      }catch(RecordStoreException e){
9          throw new RecordStoreMechanismException(e)
10     }
11   }
12 }
13
14 class FileSystemMechanism implements PersistenceMechanism{
15   public void addRecord(byte[] data)
16     throws PersistenceMechanismException
17     {
18      try{
19         . . .
20         dos.write(data,offset,data.length);
21      }catch(IOException e){
22         throw new FileSystemMechanismException(e)
23      }
24     }
25 }
26
27 public abstract class AlbumData {
28    . . .
29    public void addNewMediaToAlbum(. . .)
30      throws PersistenceMechanismException{
31         mediaAccessor.addMediaData(. . .);
32    }
33 }
34
35 class BasicMediaCommands{
36   public boolean handleCommand(Command command){
37     if (label.equals("Save Item")){
38       try {
39          . . .
40          getAlbumData().addNewMediaToAlbum(. . .);
41       }catch (PersistenceMechanismException e) {
42         Alert alert = null;
43         if (e instanceof RecordStoreMechanismException){
44            //abort the operation
45         }
46         else if (e instanceof FileSystemMechanismException){
47            //retry the operation
48         }
49       }
50     }
51   }
52 }
```

### 3.1.3 Ligação de Tratadores

Reuso e variabilidade de tratadores inevitavelmente requer um certo nível de dinamismo do mecanismo de ligação do tratador. Normalmente tratadores são reutilizados pela captura de exceções relacionadas à execução de métodos de níveis mais baixos na pilha de chamadas. Se um tratador é estaticamente ligado a um EHC próximo, ele nunca capturará exceções propagadas por métodos na parte inferior da pilha de chamadas, a menos que seja executada uma propagação explícita. Portanto essa abordagem estática não é adequada visto que impede

a reutilização de código de tratamento de erro. Em contraste a essa abordagem, a abordagem dinâmica dá suporte ao reuso de código excepcional por meio da definição de tratadores simples que podem tratar exceções propagadas a partir de métodos de baixo nível. A Figura 10, mostra, na linha 41, a definição de um tratador que utiliza a abordagem de ligação dinâmica para capturar, por subsunção, todas as exceções lançadas por todos os mecanismos de persistência. Essa estratégia permite que um programador reutilize código através da descrição sucinta de um tratador para um conjunto de exceções relacionadas. Adicionalmente, abordagens dinamicas podem capturar informações em tempo de execução para adaptar o comportamento do tratamento de erro. A Figura 10, na linha 41, exemplifica um caso no qual duas exceções podem ser capturadas por *subsumption*: *RecordStoreMechanismException* e *FileSystemMechanismException*. Portanto, o tipo da exceção capturada é levado em consideração para decidir se a operação *handleCommand* deve ser abortada ou continuada.

### 3.1.4 Propagação de Exceções

Estratégias para ligação de tratadores e propagação de exceção estão fortemente relacionadas ao modo como exceções são propagadas ao longo da cadeia de chamadas. Ligação estática de tratador depende da propagação explícita, enquanto a ligação dinâmica exige propagação automática. O ponto fraco da propagação explícita é que ela requer a inserção de tratadores ao longo da cadeia de chamadas em vez de garantir que uma exceção fará esse caminho para o topo da pilha. Desnecessariamente, a propagação explícita acarreta um aumento de métodos emaranhados. Em alguns casos isso gera a impossibilidade de reutilização do comportamento normal sem que sejam alterados os códigos normal e excepcional. Adicionalmente isso aumenta o acoplamento de métodos intermediários com classes de exceção quando não deveriam saber acerca delas. Por outro lado, propagação automática não requer essa definição custosa de tratadores, especialmente considerando a magnitude do esforço necessário para aplicar todo o sistema de propagação de exceção. No máximo, propagação automática exigirá a declaração da exceção propagada na interface de exceção. Por exemplo, na Figura 10, linha 31, um subtipo de *PersistenceMechanismException* é automaticamente propagado do método *addNewMediaToAlbum* para o método *HandleCommand.*

## 3.2 Mecanismos de Tratamento de Exceção Dirigidos a Robustez

Os mais sérios problemas relacionados à confiabilidade do comportamento de tratamento de exceção nos softwares têm raiz no fato dos mecanismos de tratamento de exceção não considerarem adequadamente exceções globais. Elas são baseadas na suposição implícita de que é suficiente especificar os lugares onde o programa lança exceções e os lugares onde o mesmo as trata. A principal consequência dessa limitação nos mecanismos de tratamento de exceção existentes é que exceções globais introduzem controle de fluxo implícito. Se um programador modifica uma exceção relacionada ao código, o controle de fluxo em partes aparentemente não relacionadas do programa pode mudar de formas inesperadas. Isso cria duas complicações diretas:

- Pode se tornar difícil descobrir onde as exceções lançadas dentro de um determinado contexto será tratada.
- Pode também tornar-se difícil o rastreamento do local de lançamento de uma exceção tratada, na perspectiva de leitura do código.

Em outras palavras, mecanismos de tratamento de exceção tradicionais proveem construções para lançamento, propagação e tratamento de exceções. Entretanto, não é fornecido um suporte tão explícito para as tarefas de definir e entender os caminhos (in) desejáveis das exceções globais, do local de lançamento para o tratador final.

Um controle de fluxo implicito ocorre sempre que um desenvolvedor não pode observar diretamente o que o código fará durante o tempo de execução. Uma vez que o comportamento do tratamento de exceção está em evidência, o controle de fluxo implícito torna mais difícil de analisar explicitamente: (i) Se uma exceção está sendo tratada e onde, (ii) quais alternativas de tratadores serão ligados a uma exceção específica, (iii) a lista de tratadores relacionados a uma exceção, e (iv) quais componentes serão afetados por um controle de fluxo de exceção.

### 3.2.1 Interface de Exceção

Ao contrário do fluxo de controle implícito, interface de exceção deve ser obrigatória. Todas as exceções propagadas por um método devem ser rigorosamente especificadas na assinatura do método. Desenvolvedores não devem ter que examinar a implementação do método para identificar quais exceções estão sendo propagadas. Para cumprir essa meta, interfaces de exceção devem ser completas e precisas. Uma interface de exceção “completa”

significa que sua definição inclui todas as exceções que possam atravessar as fronteiras de um dado método. Por exemplo: uma interface de exceção pode também ser considerada completa se ela inclui somente uma exceção genérica (a raiz de uma árvore de herança de exceção, por exemplo), a qual é supertipo de todas as exceções lançadas pelo método. Por outro lado, esse exemplo de interface de exceção não pode ser considerado preciso. Uma interface de exceção "precisa" significa que sua definição inclui uma lista exaustiva de todos os tipos atuais de exceções capazes de percorrer o método. Completude e precisão transmitem o mesmo significado para mecanismos de tratamento de exceção não orientados a objetos, nos quais a representação de exceção é definida pelo significado dos simbolos visto que não existe noção de especialização de exceções. A distinção entre completude e precisão é somente observável quando as exceções estão representadas como classes.

O método *addNewMediaToAlbum*, por exemplo, possui uma completa, porém imprecisa, interface de exceção. Ela é completa porque *PersistenceMechanismException* representa todas as exceções que podem ser propagadas pelo método, no entanto é imprecisa pois as exceções *FileSystemMechanismException* e *RecordStoreMechanismException*, propagadas conforme as regras, não estão explicitamente declaradas. Conformidade de exceção pode levar a interfaces de exceção imprecisas, obscurecendo ainda mais o controle de fluxo implícito.

Apesar das interfaces de exceção obrigatórias ajudarem a revelar o controle de fluxo implicito, elas ainda não tornam possível, explicitamente e precisamente, ligar o local de lançamento de uma exceção com seu potencial, ou desejável, tratador. Elas apenas indicam potenciais caminhos de propagação das exceções. Segundo Robillard e Murphy [Robillard 2003], mesmo quando se toma um cuidado extremo nas fases iniciais do desenvolvimento, os caminhos de propagação de exceção são criados de uma forma muito difícil de predizer e controlar. Além disso, a ajuda limitada das interfaces de exceção somente se aplicam a algumas exceções de um programa. Em Java, por exemplo, as regras para uma interface de exceção não se aplicam para as exceções não checadas (*unchecked exceptions*).

### 3.2.2 Vinculação de Tratadores

As políticas de vinculação de tratadores podem exibir um impacto negativo no controle de fluxo excepcional. Isso decorre do fato de que o maior escopo do tratador, tais como blocos, métodos, objetos e classes não tornam explícito qual declaração está efetivamente

sinalizando uma exceção. Como essa informação não é facilmente obtida devido as lacunas de documentação ou conformidade de exceção, desenvolvedores não podem determinar se um tratador vinculado foi projetado para somente uma ou várias declarações. Conforme [Reimer 2003, Robillard 2003, Sinha 2004], a perda de informações acerca do local de lançamento pode inconscientemente levar a duas indesejáveis, porém comuns, situações de programação: um único tratador para múltiplas exceções não relacionadas [Reimer 2003] e tratadores que nunca serão utilizados [Robillard 2003, Sinha 2004].

Na primeira situação, uma manutenção de software pode inserir uma nova declaração de lançamento de exceção dentro de um EHC no qual um tratador genérico está vinculado. A intenção do programador é fazer a nova exceção ser capturada por um tratador específico no método chamador. Uma vez que o compilador não reclama, o programador não muda a rotina de tratamento e, como resultado, a exceção pode ser eventualmente tratada de uma forma inadequada por um tratador genérico associado ao EHC. A segunda situação normalmente ocorre quando o programador remove a declaração à qual um tratador estava ligado. Em tais casos, por razões prudentes [Robillard 2003], o tratador não é removido e então é deixado como código morto. Do mesmo modo, como na primeira situação, o problema surge quando uma nova declaração que sinaliza tipos de exceção similares aos da declaração removida, é inserida ao EHC do tratador morto. Novamente, o tratador pode acabar lidando com uma exceção para a qual ele não foi projetado.

Essas deficiências podem ser minimizadas vinculando tratadores a declarações. Tratadores vinculados a declarações tratam condições excepcionais uma de cada vez. Esse, por sua vez, torna mais fácil rastrear que exceção está sendo tratada e propagada.

### 3.2.3 Ligação de Tratadores

O uso da busca dinâmica ou semi-dinâmica de tratadores contribui para tornar o controle implícito de fluxo mais difícil de compreender. Ligação dinamica ou semi-dinamica depende de que a busca encontre algum tratador para uma dada exceção na pilha de chamadas. Como a pilha de chamadas está disponível apenas em tempo de execução, desenvolvedores dificilmente sabem onde uma exceção será tratada dado que os tratadores podem estar um, dois ou mais níveis acima na pilha de chamadas. Nesse contexto, alguns autores tem apontado que a ligação estática é a melhor abordagem para melhorar a compreensão e confiabilidade.

Essa abordagem revela mais facilmente os fluxos de controle implicitos, uma vez que torna-se mais simples identificar qual tratador irá tratar uma ocorrência de exceção.

Um outro problema pode ocorrer quando o mecanismo de ligação do tratador manipula o tipo do objeto da exceção para encontrar um tratador que coincida. Exceções podem ser ligadas por *subsumption* na busca. Entretanto, tratadores podem acabar capturando exceções acidentalmente [Romanovsky 2001, Robillard 2003, Reimer 2003]. [Miller 1997] detalha várias situações onde a evolução de um programa compromete a efetividade do comportamento do tratamento de exceção. Todas essas situações estão baseadas no fato comum de que um módulo pode ser alterado para lançar novas exceções enquanto outros módulos inalterados têm que lidar com elas utilizando tratadores existentes. Suponhamos, por exemplo, que *addRecord* (Figura 10) pode lançar a exceção *FileSystemAlreadyOpenException* para indicar um arquivo já aberto. Como essa exceção é propagada do mecanismo de acesso ao sistema de arquivos, ela é definida como subtipo de *FileSystemMechanismException.* Entretanto, na Figura 10, se *FileSystemAlreadyOpenException* é lançada, uma ação de continuação, não intencional e inadequada, é executada, pois é capturada por *subsumption*. Como consequencia, o tratamento de exceção por *subsumption* torna quase impossível garantir que exceções são especificamente tratadas, uma vez que novas exceções podem ser introduzidas e então sobrecargar as já existentes.

### 3.2.4 Propagação de Exceção

Propagação automática é frequentemente afirmada, por muitos autores, como sendo a causa mais comum de controle de fluxo implícito. Isso ocorre porque a propagação automática permite transparentemente saltar ao longo de muitos níveis na cadeia de chamadas sem qualquer intervenção manual. Como uma consequencia, [Robillard 2003] aponta que pensar sobre o fluxo de controle implícito pode tornar-se uma tarefa árdua para os desenvolvedores. Eles serão forçados a analisar os efeitos da propagação automática de acordo com diferentes perspectivas, ou seja, amplitude e profundidade.

Na perspectiva de amplitude, os desenvolvedores terão que identificar o conjuto de todos os tipos de exceção que um método pode tratar. Nesse sentido, a perspectiva de profundidade deve ser empregada para determinar, para um tipo específico de exceção, todos os caminhos possíveis de um lançamento até seu tratador final. Infelizmente, essas duas perspectivas são

muitas vezes dificeis de obter devido a várias razões. Primeiro, desenvolvedores precisam levar em consideração herança e pensar em todos os possíveis métodos que possam ser invocados em um local específico no código para obter cada uma das perspectivas. Segundo, a determinação da perspectiva de profundidade normalmente requer um grande esforço, uma vez que estudos como [Cabral 2007, Fu 2007] mostram que em muitos casos exceções podem ser propagadas de 2 a 17 níveis pela cadeia de chamadas antes de alcançar seu tratador final. Finalmente, desenvolvedores não podem depender de interfaces de exceção, uma vez que as exceções podem também ser propagadas por *subsumption* ou trocadas por exceções não checadas através de tratadores ao longo da cadeia.

Considerados tais fatos, acredita-se que a propagação explícita tem uma influencia positiva em revelar fluxos de controle implícitos. Como exceções não são transparentemente propagadas, desenvolvedores são forçados a capturar e repropagar exceções ao longo da cadeia de chamadas. Portanto, a propagação explícita facilita o entendimento sobre os fluxos de controle implícitos uma vez que os desenvolvedores estão cientes dos possíveis caminhos de propagação desde a implementação.

## 3.3 Discussão

Conforme citado nas seções anteriores, manutenibilidade e confiabilidade são requisitos primordiais para satisfação de qualidade em projetos de mecanismos de tratamento de exceções. No entanto, concluímos que os modelos de tratamento de exceção disponíveis impõem um *trade-off* entre manutenabilidade e confiabilidade que precisa ser resolvido pelo desenvolvedor. Os mecanismos de tratamento de exceção existentes tornam difícil a implementação de um tratamento de erro robusto e modular na presença de mudanças. Decisões de projeto específicas, tais como interfaces de exceção obrigatórias normalmente favorecem a confiabilidade em detrimento da manutenibilidade. Há características, no entanto, que produzem o efeito contrário, tais como não suportar interface de exceção obrigatória. Essa característica favorece a manutenibilidade em detrimento da confiança.

As seções 3.1 e 3.2 discutiram características e decisões de projeto dos mecanismos de tratamento de exceções e seus efeitos e impactos em manutenibilidade e robustez de sistemas de software. Com base nessa análise, identificamos que para uma maximização da manutenibilidade as seguintes decisões devem ser adotadas:

- Interfaces de exceção não devem ser suportadas;
- Tratadores devem ser vinculados a métodos, classes, ou tipos excepcionais;
- Tratadores devem ser ligados a contextos de tratamento de exceções dinamicamente;
- Exceções devem ser propagadas automaticamente;

Para um mecanismo dirigido a confiabilidade, identificamos as seguintes decisões de projeto:

- Interfaces de exceção devem ser precisas e obrigatórias;
- Tratadores devem ser vinculados a declarações;
- Tratadores devem ser ligados a contextos de tratamento de exceções estaticamente;
- Exceções devem ser propagadas explicitamente;

Analisaremos a partir do capítulo 4, o impacto desse trade-off no desenvolvimento de softwares reais e como tal trade-off se apresenta em aplicações implementadas com mecanismos de tratamento de exceção dirigidos a manutenção. A escolha por avaliar mecanismos de tratamento de exceção dirigidos a manutenção se dá pelo pouco conhecimento empírico sobre como programas utilizando esses mecanismos evoluem ao longo do tempo. Seus impactos e efeitos colaterais em robustez e manutenibilidade são desconhecidos.

Cenários de programação propensos a falhas durante a alteração de código normal e excepcional precisam ser mais compreendidos quando tratamentos de exceção dirigidos a manutenção são empregados. Os estudos empíricos existentes focam somente na investigação de padrões para tratar exceções em linguagens de programação sem mecanismos de tratamento de exceções nativos [Yang 2008] [Harrold et al. 2010], ou características dirigidas a confiabilidade para mecanismos suportados, por exemplo, em Java e seus dialetos [Robillard and Murphy 2003][Fu and Ryder 2007][Coelho et al. 2008][Marinescu 2011].

Este trabalho visa preencher essa lacuna realizando um estudo empírico a cerca das relações entre alterações (manutenção/evolução) no programa e a robustez nas aplicações desenvolvidas sobre a plataforma .NET.

# 4. DESCRIÇÃO DO ESTUDO

Este capítulo apresenta a definição, o planejamento e a execução do estudo seguindo o formato proposto por Wohlin (2000). No item 4.1 é apresentada a definição do estudo, no item 4.2 seu planejamento.

## 4.1 Definição do Estudo

Conforme descreve a seção 3.1, mecanismos de tratamento de exceção dirigidos à manutenção, priorizam a manutenibilidade do comportamento normal e excepcional. No entanto, há um conhecimento limitado sobre o impacto que o uso da abordagem dirigida à manutenção pode causar na robustez do sistema. Portanto o objetivo deste estudo é **analisar** a robustez dos sistemas, **com o propósito de** avaliar, **no que diz respeito** a relação entre mudanças no comportamento normal e excepcional e seus impactos na robustez dos sistemas **do ponto de vista** do programador, **no contexto de** sistemas implementados com mecanismo de tratamento de exceções dirigido a manutenção, neste estudo, sistemas desenvolvidos em C# e de código aberto.

## 4.2 Planejamento do Estudo

### 4.2.1 Seleção do Contexto

O estudo foi realizado com softwares que possuem código aberto e foram desenvolvidos na plataforma .NET, consequentemente utilizando mecanismos de tratamento de exceções dirigidos a manutenção. Dessa forma o contexto deste estudo é o seguinte:

- Processo: Off-line;
- Participantes: Softwares desenvolvidos em C# e de código aberto;
- Realidade: Problema real;
- Generalidade: Específico;

### 4.2.2 Formulação de Hipóteses

Para este estudo, três grupos de hipóteses foram elaboradas: (i) as hipóteses para o efeito do número de mudanças no comportamento normal das aplicações no número de mudanças no código excepcional (ver Tabela 6); (ii) as hipóteses para o efeito do número de mudanças no código responsável pelo comportamento normal no nível de robustez das aplicações (ver Tabela 7); (iii) as hipóteses para o efeito do número de mudanças no nível de robustez das aplicações (ver Tabela 8).

**Tabela 6 -** Hipóteses para o efeito do número de mudanças no comportamento normal das aplicações no número de mudanças no código excepcional

| | |
|---|---|
| H0: | Não há relação significante entre o número de mudanças no código que implementa o comportamento normal e o número de mudanças no código excepcional (blocos catch). |
| Ha: | Há relação significante entre o número de mudanças no código que implementa o comportamento normal e o número de mudanças no código excepcional (blocos catch). |

**Tabela 7 -** Hipóteses para o efeito do número de mudanças no código responsável pelo comportamento normal no nível de robustez das aplicações

| | |
|---|---|
| H0: | Não existe relação significante entre o número de mudanças no código que implementa o comportamento normal e o número de fluxos excepcionais não tratados (*uncaught flows*). |
| Ha: | Há relação significante entre o número de mudanças no código que implementa o comportamento normal e o número de fluxos excepcionais não tratados (*uncaught flows*). |

**Tabela 8 -** Hipóteses para o efeito do número de mudanças no código excepcional (blocos catch) no nível de robustez das aplicações

| | |
|---|---|
| H0: | Não há relação significante entre o número de mudanças no código excepcional (blocos catch) e o número de fluxos excepcionais não tratados (*uncaught flows*). |
| Ha: | Há relação significante entre o número de mudanças no código excepcional (blocos catch) e o número de fluxos excepcionais não tratados (*uncaught flows*). |

O primeiro grupo de hipóteses tem por objetivo investigar se o número de mudanças no código normal induz a mudanças no código excepcional, ou seja, quando são realizadas alterações no código normal o código excepcional também é alterado. O segundo grupo de hipóteses, por sua vez, visa constatar se as mudanças no comportamento normal de um sistema podem induzir a uma alteração no nível de robustez da aplicação[1]. O último grupo de

[1] Neste estudo consideramos o número de fluxos excepcionais não tratados como indicador negativo de robustez.

hipóteses, investiga se as alterações no comportamento excepcional pode induzir a uma alteração no nível de robustez da aplicação[1].

### 4.2.3 Seleção de Variáveis

As variáveis do estudo podem ser dependentes ou independentes. As variáveis independentes são as informações de entrada, as informações que afetam o resultado deste estudo. As variáveis dependentes são as informações que representam o resultado da análise. No item 4.2.3.1 definimos as variáveis independentes e no 4.2.3.2 definimos as variáveis dependentes.

#### *4.2.3.1 Variáveis Independentes*

Foram consideradas, neste estudo, variáveis independentes contabilizadas a partir da análise do código fonte das aplicações e da análise estática de seus binários. Três grupos de variáveis foram então coletados: (i) Estruturais; (ii) de Mudança; (iii) de Robustez.

As variáveis estruturais representam a estrutura básica do código normal e excepcional:

- NMod: Conta o número de módulos (classes e interfaces) de cada versão de um sistema.
- NMethod: Conta o número de métodos de cada versão de um sistema.
- NTry: Conta o número de blocos try de cada versão de um sistema.
- NCatch: Conta o número de blocos catch de cada versão de um sistema.

As variáveis de mudança foram definidas considerando o número de classes adicionadas, removidas ou modificadas, número de métodos adicionados, removidos ou modificados e assim por diante. Os elementos considerados vão de um nível mais abrangente, como classes e métodos, a níveis mais específicos como linhas de código (LOC – *Lines Of* Code), blocos try e blocos catch. Neste estudo, apenas os seguintes fragmentos de código são considerados código excepcional: (i) Declarações throw, utilizadas para iniciar um fluxo excepcional, sinalizando a ocorrência de uma exceção; e (ii) Blocos catch, utilizados para tratar ou relançar uma exceção. Abaixo são detalhadas as variáveis de mudança:

- **EHLocAdded, EHLocChanged e EHLocRemoved**: Essas métricas contam, respectivamente o número de linhas de código excepcional adicionadas,

modificadas e removidas para um dado par de versões. No exemplo apresentado na Figura 8, três linhas de código são adicionadas e duas linhas modificadas. Nenhuma linha é removida. Portanto, o valor de EHLocAdded é 3, enquanto EHLocChanged é dois.

- **TryBlockAdded e TryBlockChanged:** Contam, respectivamente, o número de blocos try adicionados e modificados para um par de versões. No exemplo apresentado na Figura 8, um bloco try é adicionado e outro é modificado (Quando linhas de código são adicionadas ou removidas de um bloco try nós consideramos uma modificação no bloco try).
- **CatchBlockAdded e CatchBlockChanged:** Contam, respectivamente, o número de blocos catch adicionados e modificados para um dado par de versões. No exemplo apresentado na Figura 8, um bloco catch é adicionado e outro é modificado.
- **ClassChurned:** Conta o número de classes que teve um bloco catch adicionado ou removido. No exemplo apresentado na Figura 8, alguns blocos catch são adicionados e modificados na mesma classe, logo o valor da métrica ClassChurned é um.
- **MethodChurned:** Conta o número de métodos que teve um bloco catch adicionado ou removido. No exemplo apresentado na Figura 8, alguns blocos catch são adicionados e modificados em dois métodos diferentes, logo o valor da métrica ClassChurned é dois.
- **NormalChurnedLOC:** Conta a soma das linhas de código normal adicionadas e removidas. No exemplo apresentado na Figura 8, uma linha de código normal é modificada (bloco try um) e seis linhas de código são adicionadas, logo o valor da métrica NormalChurnedLOC é sete.

As variáveis de robustez seguem a abordagem típicamente adotada pela comunidade de tratamento de exceções que utiliza informações dos fluxos excepcionais como um indicativo de robustez. Segundo [Coelho et al. 2008], o objetivo dos mecanismos de tratamento de exceções é tornar os programas mais confiáveis e robustos. [Robillard and Murphy 2003] afirma:

> Um sistema robusto terá uma ou mais políticas de tratamento de exceção, cujos objetivos são evitar acidentes limitando o fluxo de exceções inesperadas e permitir

uma melhor recuperação descrevendo condições excepcionais de forma significativa para os módulos clientes.

Um fluxo excepcional é um caminho no grafo de chamadas de um programa que conecta o sinalizador e o tratador de uma exceção. Caso não exista um tratador para uma exceção específica, o caminho do fluxo excepcional inicia no sinalizador e termina no ponto de entrada do programa. Portanto o fluxo excepcional é composto por três partes principais: (i) A sinalização da exceção; (ii) O fluxo da exceção pelos elementos de um sistema e (iii) O trecho de código pelo qual a exceção é tratada, ou escapa sem tratamento do escopo do sistema.

Utilizamos três variáveis a fim de caracterizar robustez através das informações dos fluxos excepcionais de um sistema. Elas foram escolhidas por serem capazes de revelar a presença de potenciais falhas em um sistema. São elas:

- ***SpecializedFlow*:** Conta o número de fluxos de exceção nos quais a exceção lançada é capturada por um tratador específico, ou seja, com um mesmo tipo.
- ***SubsumptionFlow*:** Conta o número de exceções *subsumption.* Um fluxo excepcional é dito *subsumption* quando a exceção lançada é capturada por um tratador genérico, no qual o seu tipo é supertipo da exceção capturada.
- ***UncaughtFlow*:** Conta o número de fluxos excepcionais que extrapolam a fronteira do sistema sem ser tratado.

*UncaughtFlow* (Fluxo não tratado), por exemplo, é um tipo de fluxo excepcional que pode terminar a execução de um sistema, logo, quanto maior o número de fluxos desse tipo, menor o nível de robustez do sistema. *SubsumptionFlow* indica fluxos que a aplicação tem tratado de uma forma genérica, ou seja, não há uma recuperação específica para um determinado erro, ou um fluxo alternativo específico para uma situação.

#### *4.2.3.2 Variáveis Dependentes*

As variáveis dependentes são: (i) a correlação entre o número de mudanças no comportamento normal e o número de mudanças no comportamento excepcional; (ii) a correlação entre as mudanças no código normal e o número de fluxos excepcionais não tratados; (iii) a correlação entre o número de mudanças no código excepcional e o número de fluxos excepcionais não tratados.

### 4.2.4 Seleção dos Participantes

Analisamos a evolução do comportamento excepcional de 16 programas de código aberto, implementados em C#, detalhados na Tabela 9. Selecionamos esses programas baseados no estudo desenvolvido por [Cabral and Marques 2007a]. Optamos por selecionar os programas a partir desse estudo pelo fato deles cobrirem um amplo aspecto de mecanismos dirigidos à manutenção utilizados em ambientes reais de desenvolvimento de software. Consideramos o máximo de aplicações utilizadas no estudo de [Cabral and Marques 2007a], excluindo, apenas, as que não continham mais do que uma versão disponível para *download.* Para esses casos, substituímos por outros programas que atendessem aos requisitos do estudo e pertencessem a mesma categoria. A Tabela 9 apresenta o nome de cada uma das aplicações analisadas, as versões utilizadas no estudo, a quantidade de pares de versões analisados e a quantidade média de linhas de código (Média LOC – *Lines Of Code*).

Os programas foram classificados utilizando as categorias definidas em [Cabral and Marques 2007a]:

- *Libraries:* Compreende as bibliotecas de software, componentes que proveem uma API específica do domínio
- *Server-Apps:* Aplicações que rodam em servidores, por exemplo: aplicações web, com seus "ASPs" e classes relacionadas.
- *Servers:* Softwares servidores, ou seja, que respondem a requisições através de uma rede. Por exemplo: servidores de arquivo e servidores de e-mail.
- *Stand-Alone:* Essa categoria compreende o conjunto de aplicações desktop.

A coluna "Número de Pares" representa a quantidade de pares de versões analisados, por exemplo: A aplicação SL File Manager teve 4 de suas versões analisadas, a SL0, SL1, SL2 e SL3, a análise se dá em pares, como os seguintes: SL0 – SL1; SL1 – SL2; SL2 – SL3; Ou seja, a SL File Manager teve 3 pares analisados.

| | Nome | Versões | Núm. Pares | Média LOC |
|---|---|---|---|---|
| Libraries | Direct Show Lib | Beta1; 1.0; 1.1; 1.2; 1.3; 1.4; 1.5; 2.0; 2.1 | 8 | 18.580,38 |
| | Dot Net Zip (ZIP) | 1.9.1.7; 1.9.1.8 | 1 | 12.392,00 |
| | Report Net | 0.07.03; 0.07.04; 0.08.00; 0.08.01; 0.09.00; 0.09.01; 0.09.02; 0.09.05 | 7 | 6.258,57 |
| | SmartIRC4Net | 0.2.0; 0.3.0; 0.3.5; 0.4.0 | 3 | 6.117,67 |
| | **SubTotal** | | **19** | **223.298,00** |
| Server-Apps | Blog Engine (Admin) | 1.4.5; 1.5; 1.6.1; 2.0; 2.5 | 4 | 4.656,75 |
| | MVC Music Store | 0.8; 1.0; 2.0; 2.1; 3.0 | 4 | 1.644,50 |
| | PhotoRoom | 1.0; 1.4; 1.5; 1.6; 1.7 | 4 | 1.048,50 |
| | Sharp WebMail | 0.1; 0.2; 0.3; 0.4; 0.5; 0.7; 0.8; 0.9; 0.10; 0.11; 0.12; 0.13 | 11 | 2.597,55 |
| | **SubTotal** | | **23** | **57.972,00** |
| Servers | Neat Upload | 1.0; 1.1; 1.2; 1.3; 1.7 | 4 | 8.635 |
| | RnWood SMTP Server | 2.0 alpha1; 2.0 alpha2; 2.0; 2.1 alpha1 | 3 | 2.638 |
| | SL FileManager | SL0; SL1; SL2; SL3 | 3 | 199,33 |
| | Super Web Socket | Drop 1;Drop 2; Drop 3; Drop 4;Drop 5; Drop 6 | **5** | **1.048,80** |
| | **SubTotal** | | **15** | **37.023,00** |
| Stand-alone | Asc Generator | 0.7.1; 0.7.2; 0.7.3; 0.8.0;0.8.1; 0.8.2; 0.9.0; 0.9.1;0.9.5; 0.9.6 | 9 | 9.676,56 |
| | CircuitDiagram (EComponents) | 0.1; 0.2; 0.3; 0.4; 0.5; 0.6; 1.0; 1.1 | 7 | 2.204,14 |
| | NUnit (NUnitCore.Core) | 2.4.5; 2.4.8; 2.5-2008-04-07; 2.5.0.82; 2.5.0.83; 2.5.0.90; 2.5.0.909; 2.5.0.911; 2.5.0.912; 2.5.1.918; 2.5.2.922 | 11 | 6.862,78 |
| | Sharp Developer (Main.Core) | 1.0.3; 1.1.0; 2.0.1.1710; 2.1.0.2429; 2.2.0.2595; 2.2.1.2648; 3.0.0.3800; 3.1.0.4077; 3.1.0.4545; 3.1.0.4890; 3.1.1.5052; 3.1.1.5327; 3.2.0.5505; 3.2.0.5698; 3.2.1.6466; 4.0.0.5570; 4.0.0.5949; 4.0.0.6361; 4.0.0.6721; 4.0.0.6842; 4.0.0.7043; 4.0.0.7070 | 21 | 5.289,35 |
| | **SubTotal** | | **48** | **270.070,00** |
| **Total** | | | **100** | **588.263,00** |

**Tabela 9 -** Programas selecionados.

As aplicações selecionadas são uma representação de como a abordagem de tratamento de exceção dirigida à manutenção é aplicada no dia a dia de quem emprega esforços na construção de softwares reais. Essa representatividade se justifica por duas razões: A primeira é que esse grupo é composto por projetos de software não triviais e é diverso no que diz respeito ao modo como o tratamento de exceção é empregado. Diversas categorias de tratadores, no que se trata de estrutura [Cabral and Marques 2007a], podem ser encontradas nesse grupo de aplicações. A segunda razão é que os comportamentos excepcionais são significativamente diversos em relação as suas finalidades [Cabral and Marques 2007a], variando de log de erros a ações de recuperação específicas da aplicação (*rollback*, por exemplo).

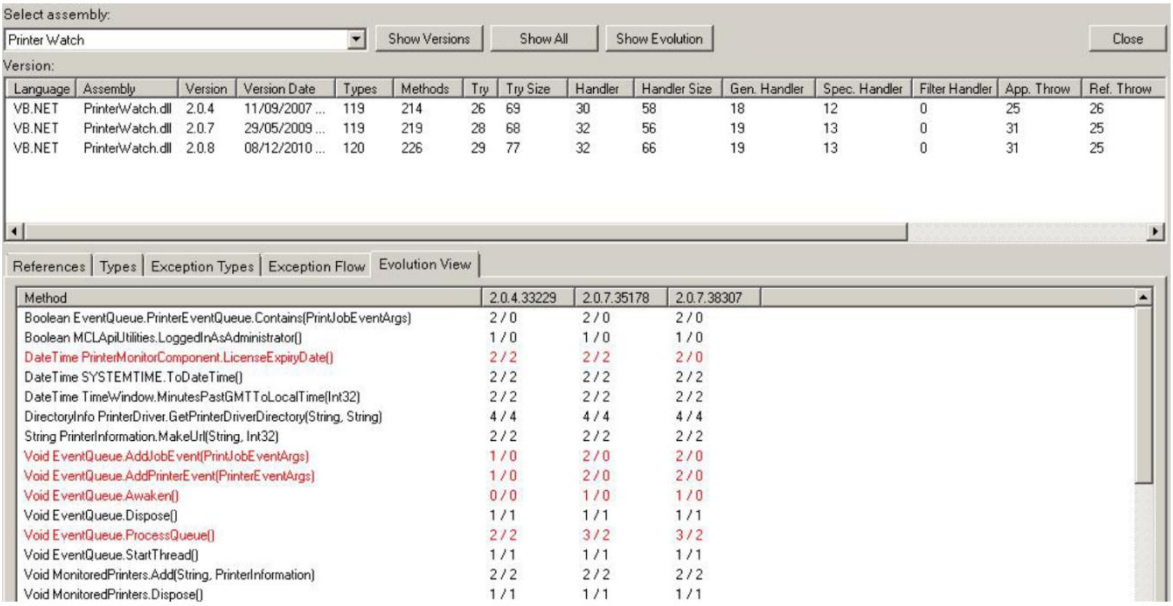
## 4.3 Coleta de Dados

Esta seção apresenta os dados coletados e o procedimento de coleta de cada grupo de variáveis. O item 4.3.1 apresenta os procedimentos de coleta e os dados estruturais, ao passo que o item 4.3.2 detalha os dados de mudança, enquanto o item 4.3.3 trata dos dados de robustez.

### 4.3.1 Dados Estruturais

As variáveis estruturais (Ntry, NCatch, NMod e NMethod) foram contabilizadas através do uso de duas ferramentas distintas. NTry que contabiliza o número de blocos try de uma aplicação e NCatch que contabiliza o número de blocos catch foram contabilizadas pela eFlowMining [Garcia and Cacho 2011], uma ferramenta de análise estática que a partir da análise do binário de um programa .NET, utiliza o grafo de chamadas de um programa para rastrear o fluxo das exceções através dos caminhos desse grafo, coletando métricas sobre o comportamento excepcional. A Figura 11 exemplifica como a eFlowMining extrai dados das aplicações.

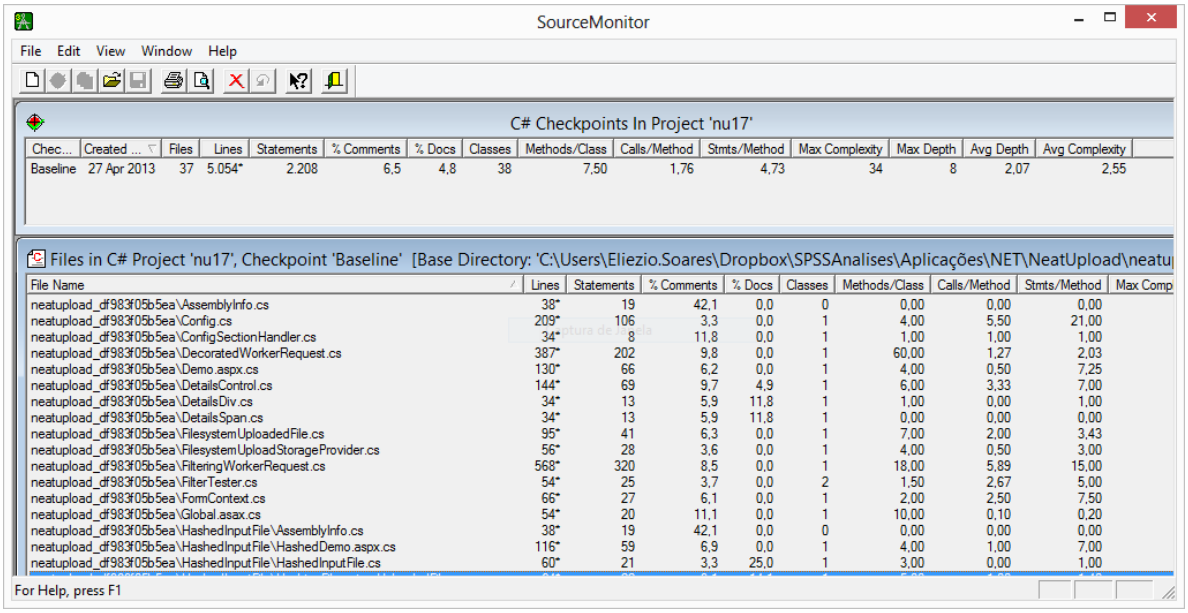
**Figura 11 -** Coleta dos dados NTry e NCatch




As variáveis NMod, que contabiliza o número de módulos (classes e interfaces) de uma aplicação, e NMethod, que registra o número de métodos, foram obtidas através da ferramenta SourceMonitor[2], uma ferramenta que analisa o código fonte das aplicações e extrai números de classes, métodos, interfaces, média de chamadas por método, entre outras informações. A Figura 12 mostra dados extraídos pela SourceMonitor da aplicação NeatUpload.

**Figura 12 -** Coleta dos dados NMod e NMethod

[2] http://www.campwoodsw.com/sourcemonitor.html




### 4.3.2 Dados de Mudança

A análise de mudanças foi realizada com base nos códigos fontes dos programas selecionados, em duas abordagens: (i) Abordagem Automática e (ii) Abordagem Manual. A abordagem automática foi utilizada somente para o cálculo da métrica NormalChurnedLOC, através da ferramenta *Microsoft Line of Code* (LOC) *Counter* (LOCCounter). A ferramenta LOCCounter fornece a soma de todas as linhas de código adicionadas, porém sem distinguir se o código é normal ou excepcional. Ou seja, obtém-se a métrica ChurnedLOC (essa métrica não é considerada no estudo). Com o valor de ChurnedLOC subtrai-se o valor da soma das métricas EHLocAdded e EHLocChanged (NormalChurnedLOC = ChurnedLOC – (EHLocAdded + EHLocChanged)).

Para realizar a abordagem manual no restante das métricas foi utilizado o software DiffMerge[3] que compara diretórios e mostra graficamente as diferenças entre dois arquivos e também os arquivos que existem em apenas uma das pastas. Com esse recurso foi realizada a análise detalhada das alterações em código excepcional de uma versão para a seguinte de um mesmo programa.

Quatro estudantes de mestrado, contaram manualmente as mudanças em cada métrica inspecionando as mudanças sugeridas pela ferramenta DiffMerge. Como resultado dessa inspeção uma planilha foi gerada para cada um dos softwares com a lista de mudanças, contendo o lugar no código onde houve a alteração e o valor da métrica (Ver Figura 13). Por

[3] HTTP://www.sourcegear.com/diffmerge/

exemplo, da versão 1.0 para a versão 1.1 do “Programa A”, uma classe foi adicionada e nessa classe há um “try” e um “catch”. Nesse exemplo, a planilha “Programa A” em sua linha “1.0 – 1.1” teve seus campos “Class/Added”, “Method/Added”, “Try Blocks/Added” e “Catch Blocks/Added” incrementados em uma unidade e o campo ”Number Lines of Code/Added” incrementado com o somatório de linhas de código do bloco “try” e do bloco “catch” (Ver Figura 14).

**Figura 13 -** Planilha de Registro de Mudanças - Detalhada

| | A | O | P | R | S | T | V | W | X |
|---|---|---|---|---|---|---|---|---|---|
| 1 | **Class/Method** | **Try Block** | | **Catch Block** | | | **Number of Lines of Code** | | |
| 2 | | Add | Changed | Add | Changed | | Add | Changed | Removed |
| 3 | | | | | Block | Caught Types | | | |
| 4 | admin_Comments_DataGrid .ProcessSelected(ActionType action) | | | | | | | | **1** |
| 5 | admin_Comments_Settings .ApprovedCnt(object total, object cought) | | | | | | | | **1** |
| 6 | admin_Comments_Settings .Accuracy(object total, object mistakes) | | | | | | | | **1** |
| 7 | admin_Pages_configuration .btnTestSmtp_Click(object sender, EventArgs e) | | | | | | | | **2** |
| 8 | admin_admin.AdminProfile() | | | | | | | | **2** |

A validação dos dados obtidos foi realizada por um aluno de Ph.D, analizando os dados registrados e classificando-os como “verdadeiros positivos” ou “falsos positivos”. Após isso foram realizadas discussões abertas com os pesquisadores para cada “falso positivo” a fim de se obter um consenso e dessa forma confirmar a avaliação.

**Figura 14 -** Planilha de Registro de Mudanças - Resumo

| | A | C | G | O | P | R | S | T | V | W | X |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | Class | Method | Try Blocks | | Catch Block | | | Number Lines of Code | | |
| 2 | Version | Change | Change | Add | Changed | Add | Changed | | Add | Changed | Removed |
| 3 | | | | | | | Block | Num Types | | | |
| 4 | 1.4.5 - 1.5 | 2 | 2 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 1.5 - 1.6.1 | 2 | 0 | 4 | 0 | 4 | 0 | 0 | 5 | 0 | 1 |
| 6 | 1.6.1 - 2.0 | 0 | 4 | 11 | 0 | 11 | 0 | 0 | 33 | 0 | 7 |
| 7 | 2.0 - 2.5 | 3 | 2 | 4 | 2 | 4 | 2 | 0 | 6 | 0 | 5 |
| 8 | Total | 7 | 8 | 19 | 4 | 19 | 2 | 0 | 44 | 0 | 13 |

| | Cenário | Descrição |
|---|---|---|
| Mudanças no código excepcional | Tratador genérico adicionado | Um bloco catch que captura exceções genéricas foi adicionado. |
| | Tratador vazio adicionado | Um bloco catch vazio que captura exceções genéricas foi adicionado. |
| | Tratador genérico adicionado com rethrow | Um bloco catch que captura exceções genéricas e as re-lança foi adicionado. |
| | Tratador genérico removido | Um bloco catch que captura exceções genéricas foi removido. |
| | Tratador especializado adicionado | Um bloco catch que captura exceções especializadas foi adicionado. |
| | Tratador especializado vazio adicionado | Um bloco catch vazio que captura exceções especializadas foi adicionado. |
| | Tratador especializado adicionado com rethrow | Um bloco catch que captura exceções especializadas e as re-lança foi adicionado. |
| | Tratador especializado removido | Um bloco catch que captura exceções especializadas foi removido. |
| | Mudança na política de tratamento de exceção | Mudanças no modo com que um determinado tipo de exceção é tratado. |
| | Mudança no bloco catch em decorrência do código normal | Por causa de alterações no código normal, alguma alteração foi exigida dentro do bloco catch (Por exemplo: alteração de nome de variável). |
| | Nenhuma alteração no bloco catch | Alteração no bloco try, mas nada foi modificado no bloco catch. |
| Mudanças no código normal | Bloco try adicionado a um método existente | Um bloco try foi adicionado a um método já existente. |
| | Novo método adicionado contendo um bloco try | Um novo método foi adicionado contendo um bloco try. |
| | Nova invocação a um método adicionada | Uma nova invocação a método foi adicionada dentro de um bloco try. |
| | Variável modificada | Uma variável foi alterada dentro de um bloco try. |
| | Fluxo de controle modificado | Uma declaração de controle de fluxo foi alterada dentro de um bloco try. |
| | Bloco try rermovido | Um bloco try foi removido de um método existente. |
| | Método contendo bloco try removido | Um método contendo um bloco try foi removido do projeto. |
| | Nenhuma alteração no bloco try | O bloco catch foi alterado sem nenhuma alteração no bloco try. |

**Tabela 10 -** Ocorrências de cenários de mudança de códigos normal e excepcional.

*4.3.2.1 Cenários de Mudança*

Para cada alteração registrada nas planilhas demonstradas pelas Figuras 13 e 14, foram descritos textualmente os cenários com as mudanças observadas dentro dos blocos try (código normal) e dos blocos catch (código excepcional). Um exemplo de descrição textual dos cenários de mudança ocorrido: "Foi adicionada uma nova declaração if no bloco try. No bloco catch foi inserida uma nova invocação a um método".

Uma técnica de codificação foi aplicada às descrições textuais de cada cenário extraindo então, em categorias, as mudanças no código normal e no código excepcional. A Tabela 10 apresenta a categorização desses cenários de mudança.

### 4.3.3 Dados de Robustez

As variáveis de robustez (SpecializedFlow, SubsumptionFlow e UncaughtFlow) foram coletadas através da ferramenta eFlowMining [Garcia and Cacho 2011] (Figura 11), que como citado no item 4.3.1 deste capítulo coleta métricas sobre o comportamento excepcional a partir do grafo de chamadas de um programa.

| Autor | Nome ferramenta | Linguagem alvo | Exibe fluxos graficamente | Registra histórico das análises | Acompanha evolução do fluxo excepcional | Alvo da análise |
|---|---|---|---|---|---|---|
| Chang *et al* (2001) | Não possui | Java | Não | Não | Não | Código Fonte |
| Fahndrich *et al* (1998) | Não possui | ML | Não | Não | Não | Código Fonte |
| Schaefer e Bundy (1993) | Não possui | Ada | Não | Não | Não | Código Objeto |
| Robillard e Murphy (2003) | JEX | Java | Sim | Não | Não | Código Fonte |
| Fu e Ryder (2007) | Não possui | Java | Não | Não | Não | Código Objeto |
| Shah *et al* (2008) | Enhance | Java | Sim | Não | Não | Código fonte |
| Coelho *et al* (2008) | SAFE | Java/AspectJ | Sim | Não | Não | Código Objeto |
| Garcia et al (2011) | eFlowMining | .NET | Sim | Sim | Sim | Bytecode |

**Tabela 11 -** Características das ferramentas de análise estática do fluxo excepcional.

O uso de ferramentas de análise estática é uma abordagem clássica para mitigar problemas criados por fluxos excepcionais implícitos. Existem algumas ferramentas [SCHAEFER; BUNDY, 1993; FAHNDRICH *et al*, 1998; CHANG *et al*., 2001; ROBILLARD; MURPHY, 2003; CABRAL; MARQUES; SILVA, 2005; FU; RYDER, 2007; SHAH; CARSTEN; HARROLD, 2008; COELHO *et al*, 2008] que, baseadas no grafo de chamadas de um programa, rastreiam o fluxo das exceções através dos caminhos desse grafo, além de coletar métricas sobre o comportamento excepcional. Elas fornecem uma ajuda valiosa para os desenvolvedores tentarem entender como o programa se comporta na presença de exceções globais, e como as exceções se comportam.

Com base nos dados da Tabela 11, optamos por utilizar a ferramenta eFlowMining [Garcia and Cacho 2011] para coletar as métricas de robustez. Utilizamos a ferramenta eFlowMining, pois como se pode ver na Tabela 11, ela é capaz de analisar aplicações .NET e a única que nos permite acompanhar a evolução dos fluxos excepcionais nas aplicações, registrando o histórico das análises e versões e apresenta essas informações graficamente.

A Tabela 11 consolida as seguintes características das ferramentas dos trabalhos citados acima: (i) exibe fluxos graficamente, se a ferramenta permite a visualização dos fluxos excepcionais na forma de árvore ou de partes do grafo de chamadas; (ii) registra histórico das análises, se as análises são armazenadas em banco de dados, arquivos XML, ou outro tipo; (iii) acompanha evolução do fluxo excepcional, se a ferramenta compara o fluxo excepcional de versões diferentes do mesmo programa e exibe as diferenças; (iv) alvo da análise, se a análise é feita no código fonte ou código objeto do programa.

Analisando a Tabela 11 percebe-se que a maioria das ferramentas são focadas na plataforma Java, somente o estudo [CABRAL; MARQUES; SILVA, 2005] examinou a plataforma .NET, ele porém não implementou a visualização gráfica dos fluxos excepcionais em forma de árvore. Essa visualização também não foi trabalhada na maioria das ferramentas, somente por [ROBILLARD; MURPHY, 2003] e [SHAH; CARSTEN; HARROLD, 2008]. Além disso, apenas uma ferramenta implementa o registro histórico das análises e o acompanhamento da evolução do fluxo excepcional entre as versões das aplicações. Esses dois requisitos, bem como a visualização gráfica dos fluxos excepcionais, são aspectos que ajudam a aumentar a objetividade das análises. O registro histórico permite que uma análise seja armazenada em um mecanismo de persistência. Tal registro serve para reanalises e acompanhamento da evolução dos fluxos, pois a partir das análises armazenadas é possível comparar os fluxos que surgem/desaparecem entre as versões.

## 4.4 Ameaças a Validade do Estudo

Esta seção trata das ameaças a validade do estudo que podem afetar os resultados demonstrados nas seções seguintes. As ameaças podem ser de quatro tipos: (i) Validade Interna; (ii) Validade da Construção; (iii) Validade Externa; (iv) Validade da Confiabilidade.

As ameaças a validade interna estão relacionadas a fatores desconhecidos que possam ter influenciado nos resultados deste experimento [Goodenough, J. B. 1975]. Para reduzir esse risco, nós selecionamos aplicações das quais os desenvolvedores não tem conhecimento deste estudo, evitando, dessa forma, que eles pudessem alterar suas práticas de programação. Adicionalmente, nossos resultados podem ter sido influenciados pela performance em termos de precisão das ferramentas utilizadas. Nós tentamos limitar o número de falsos positivos através de uma validação manual. Essas validações amenizam os riscos a validade interna, mas não os elimina completamente, da mesma que as ferramentas que lidam com problemas indecidíveis não podem evitar uma margem de imprecisão.

As ameaças a construção do experimento estão em como se relacionam os conceitos e teorias por tras do experimento e o que é medido e afetado [Wohlin, C., et al. ]. Para reduzir esses riscos nós usamos métricas que são amplamente utilizadas para quantificar a extensão das mudanças na evolução do software [Gosling, J. et al. 1996], [Fu and Ryder 2007], e para medir a robustez do software [Garcia and Cacho 2011], [Cabral and Marques 2007b], [Balter, R., et al., 1994], [Cristian, F. 1979].

Ameaças a validade externa do estudo podem surgir do fato de que todos os dados foram coletados de 16 sistemas que podem não ser representativos da prática da indústria. Entretanto, a heterogeneidade desses sistemas ajuda a minimizar esse risco. Eles são de categorias distintas e amplamente utilizados e extensivamente avaliados em pesquisas anteriores [Araujo, J., et al. 2012].

As ameaças a validade da confiabilidade dizem respeito a possibilidade de replicação do estudo. O código fonte dos programas analisados estão publicamente disponíveis. A maneira como os dados foram coletados está detalhadamente descrita na seção 4.3 deste trabalho. A despeito disso, seria possível obter os mesmos dados através das ferramentas eFlowMining e SourceMonitor que estão disponíveis.

## 4.5 Trabalhos Relacionados

Dulaigh et al. [2012] mediram e estudaram a quantidade e distribuição das construções de tratamento de exceções em 12 aplicações Java durante sua evolução. A principal meta dos autores era explorar diferentes métricas relacionadas a exceções e visualizações durante a evolução do software. Estudando tais métricas, os autores foram capazes de identificar algumas mudanças no código relacionadas a exceções e mudanças na política de tratamento de exceção dos programas analisados. Os autores, no entanto, não focaram na análise do impacto do tratamento de exceção na robustez dos softwares, como este trabalho se propõe.

Em [Cabral and Marques 2007b] foi analisado o código de tratamento de exceções de 32 aplicações de código aberto de diferentes domínios e para as plataformas Java e .Net. O foco desse estudo foi categorizar os blocos catch em termos das ações de tratamento implementadas em vez de entender como o tratamento de exceção foi utilizado pelos desenvolvedores de software. A principal conclusão desse trabalho é que a maior parte dos tratadores de exceção implementados são simplistas, implementando apenas uma das seguintes ações: registro de log; notificação de usuário; finalização da aplicação. [Cabral and Marques 2007b] analisou apenas uma versão de cada programa, enquanto este trabalho analisa os programas selecionados em sua evolução.

O trabalho de [Coelho et al. 2008] avaliou o impacto do tratamento de exceção na robustez dos softwares no contexto de programas orientados a aspectos. Os autores avaliaram a robustez de tres programas de médio porte implementados em AspectJ. Semelhantemente a abordagem proposta neste estudo, os autores investigaram as aplicações selecionadas durante sua evolução e utilizaram a métrica *Uncaught Exception Flows* (Fluxos Excepcionais não tratados). Como resultado, o estudo mostrou que o número de exceções não tratadas cresceu durante a evolução dos programas. O presente trabalho possui algumas importantes diferenças, uma vez que [Coelho et al. 2008] investigou robustez apenas na perspectiva da evolução dos softwares e não explorou o relacionamento entre as atividades de manutenção e a robustez dos mesmos. Nesse sentido, este trabalho o complementa.

[Marinescu 2011] analisou a propensão a defeitos de classes que tratem ou lancem exceções no contexto de tres versões do Eclipse. O autor categorizou os defeitos em pré release e pós release, onde pré release corresponde aos defeitos que são reportados em um período de no máximo seis meses antes do lançamento, enquanto os defeitos pós releases são aqueles reportados nos seis meses seguintes ao lançamento. Este estudo é similar ao de

[Marinescu 2011] no sentido que buscou-se correlacionar propriedades do código de tratamento de exceção com propriedades de robustez de software. Entretanto, este estudo utiliza propriedades mais granulares do código de tratamento de exceção do que o estudo apresentado em [Marinescu 2011] e também apresenta um conjunto de cenários de tratamento de exceção propensos a falta.

Em um estudo diferente, porém conduzido também no contexto de versões do Eclipse, Sawadpong et al [2012] comparou a densidade de defeito do código de tratamento de exceção e código normal. A densidade de defeito é definida como "o número de defeitos conhecidos causados por exceções dividido pelo número de linhas de código-fonte não comentadas". O principal resultado desse trabalho é que a densidade de defeito do código excepcional é quase três vezes maior do que a densidade de defeito total. O presente estudo tem uma meta similar, mas difere principalmente nas métricas utilizadas para medir robustez, em [Sawadpong et al. 2012] foi utilizada a métrica "densidade de defeito" e neste "uncaught flows" (fluxos excepcionais não tratados). Adicionalmente este estudo não apenas coletou métricas de robustez como também adotou uma abordagem qualitativa para compreender e descrever em detalhes os cenários de mudança onde falhas de exceção relacionadas foram introduzidas. Dessa forma, este trabalho pode ser compreendido com um complemento mais preciso de ambos, [Marinescu 2011] e [Sawadpong et al. 2012].

# 5. Análise dos Dados e Resultados

Este capítulo apresenta os dados coletados em cada grupo de variáveis. O item 5.1 apresenta os dados obtidos e uma análise geral dos mesmos, enquanto o item 5.2 analisa os dados na perspectiva das hipóteses definidas na seção 4.2.2.

| Aplicações | | NTry | NCatch | NMod | NMethod |
|---|---|---|---|---|---|
| Libraries | DirectShowLib | 3,33 | 3,33 | 718,22 | 104,78 |
| | DotNetZip | 56,50 | 61,50 | 192,00 | 2323,00 |
| | ReportNET | 1,00 | 1,00 | 120,25 | 777,88 |
| | SmartIRC4Net | 13,00 | 13,25 | 37,25 | 558,50 |
| | **TOTAL** | **8,33** | **9,30** | **346,04** | **610,70** |
| Server-Apps | BlogEngine | 11,20 | 11,20 | 30,60 | 211,20 |
| | MVC Music Store | 2,40 | 3,20 | 54,20 | 222,40 |
| | PhotoRoom | 2,00 | 2,00 | 12,20 | 87,40 |
| | SharpWebMail | 24,50 | 26,50 | 18,08 | 186,08 |
| | **TOTAL** | **13,78** | **14,81** | **26,00** | **179,19** |
| Servers | NeatUpload | 12,40 | 14,40 | 51,80 | 446,40 |
| | RnWood SMTP Socket | 12,00 | 13,25 | 69,75 | 246,50 |
| | SuperSocket | 1,00 | 1,00 | 38,00 | 158,17 |
| | SuperWebSocket | 38,00 | 41,00 | 54,33 | 400,67 |
| | **TOTAL** | **12,78** | **14,11** | **51,61** | **298,28** |
| Stand Alone | AscGenerator | 63,70 | 70,60 | 92,30 | 1253,00 |
| | CircuitDiagram | 7,87 | 7,87 | 30,25 | 176,50 |
| | NUnit | 22,45 | 22,45 | 245,36 | 886,91 |
| | SharpDevelop | 31,41 | 33,36 | 76,45 | 450,18 |
| | **TOTAL** | **32,12** | **34,31** | **87,18** | **658,86** |

**Tabela 12 -** Dados Estruturais das aplicações nas quatro categorias.

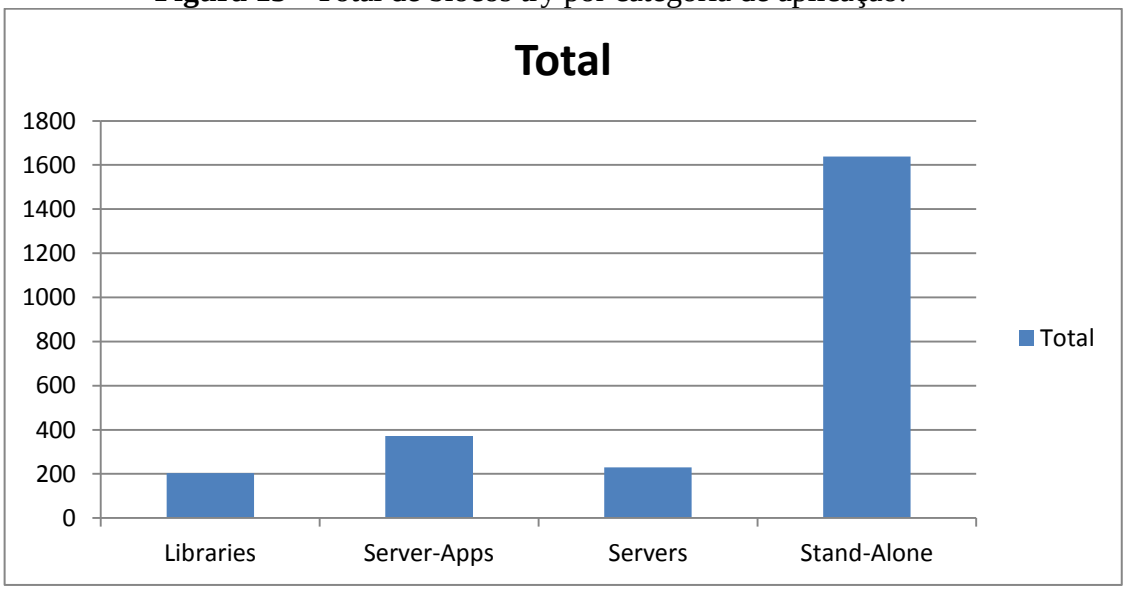
## 5.1 Análise dos Dados Coletados

Esta seção apresenta os dados coletados a partir dos procedimentos descritos e detalhados no item 4.3 deste trabalho. A seguir são apresentados os dados estruturais, na seção 5.1, na seção 5.2 os dados de mudança e na seção 5.3 os dados de robustez.

### 5.1.1 Dados Estruturais

A Tabela 12 contém a média dos dados estruturais das aplicações analisadas. As aplicações estão organizadas em suas respectivas categorias e para cada categoria há um totalizador contendo a média da métrica na categoria relacionada. Por exemplo, a métrica NTry possui uma média de 8,33 para a categoria Libraries, 13,78 para as Server-Apps, 12,78 para as Servers e 32,12 para as Stand-Alone, tais valores sugerem que os desenvolvedores se preocupam significativamente mais com o tratamento de exceções quando estão desenvolvendo aplicações stand-alone e tendem a não escrever código de tratamento de exceções para Libraries.

**Figura 15 -** Total de blocos try por categoria de aplicação.




A Figura 15 apresenta um gráfico de barras com o total de blocos try existentes por categoria de aplicação. As aplicações da categoria libraries (bibliotecas de classes) apresentam o menor número de blocos try (203 no total), seguido do conjunto de aplicações da categoria servers (servidores) que totalizam 230 blocos try. As aplicações de servidores

(server-apps) apresentam um número um pouco maior de blocos try, 372 no total. O maior número de blocos try é das aplicações Stand-Alone que totalizam 1638 blocos, o que sinaliza um maior cuidado dos desenvolvedores quando se trata dessa categoria de aplicação.

**Figura 16 -** Total de blocos catch por categoria de aplicação.

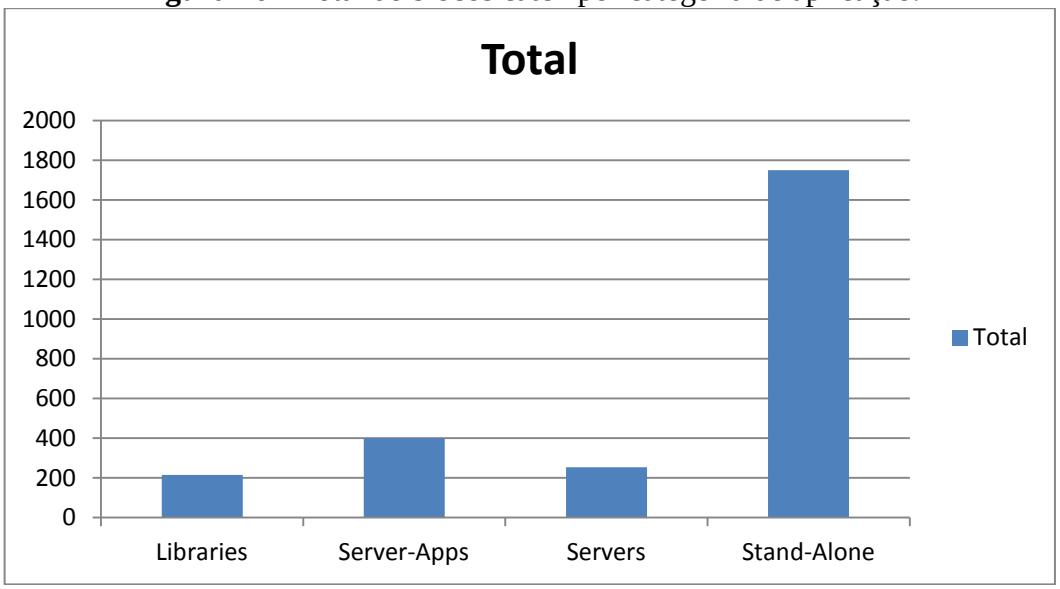
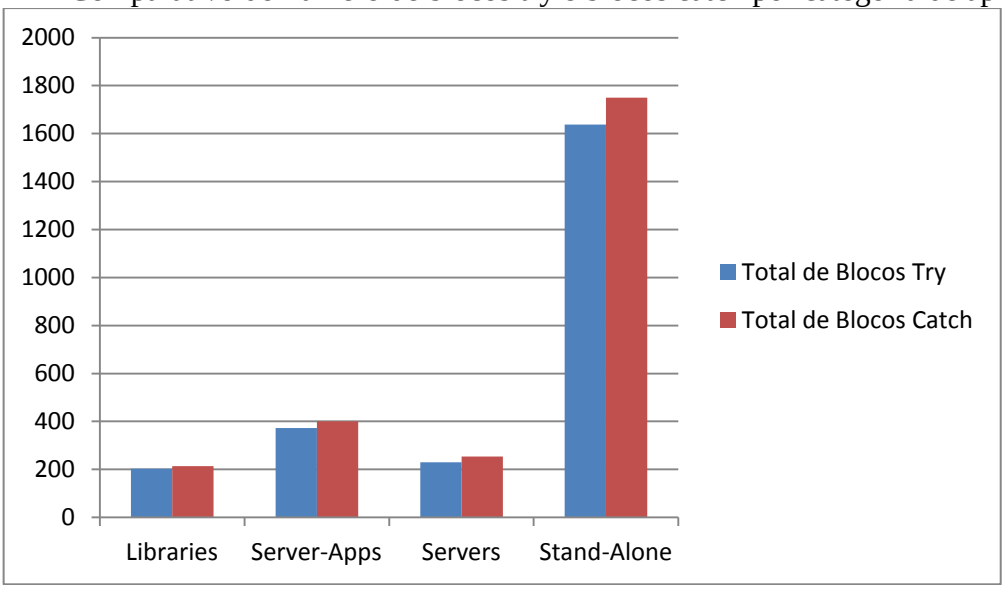


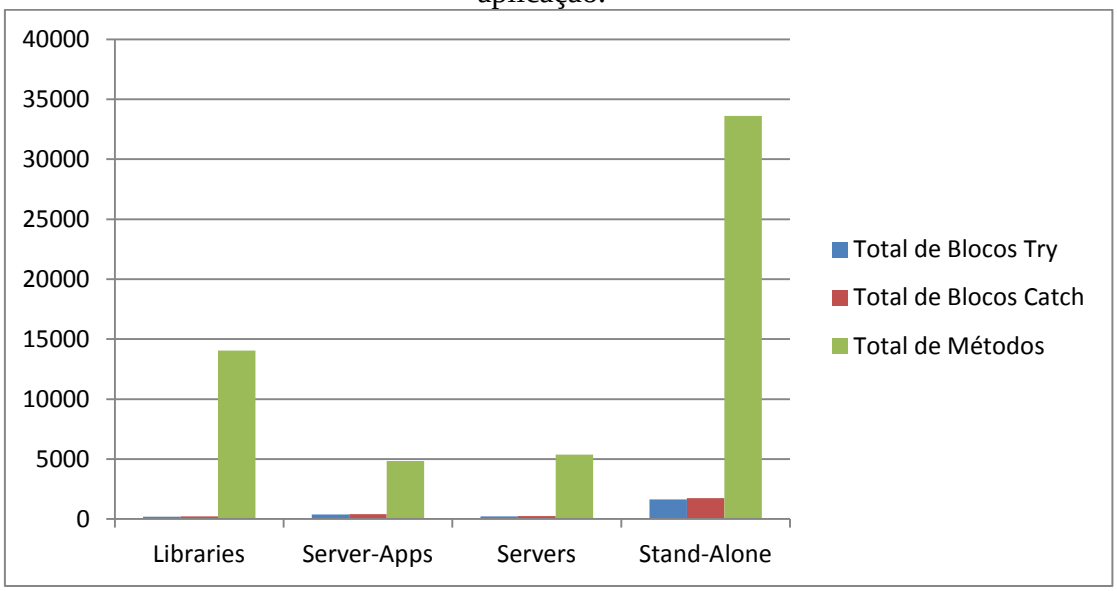

**Figura 17 -** Comparativo do número de blocos try e blocos catch por categoria de aplicação.




O gráfico apresentado na Figura 16 apresenta os totais de blocos catch por categoria de aplicação, são 214 nas aplicações da categoria Libraries, 400 das Server-Apps, 254 das Servers e 1750 das Stand-Alone. A proporção segue a mesma ordem da apresentada na Figura 15 para os blocos try, no entanto a Figura 17 mostra que o número de blocos catch foi maior do que o número de blocos try em todas as categorias de aplicação o que pode ser um bom

sinal, uma vez que existe uma proporção maior que um de blocos catch para blocos try possibilitando um tratamento mais especializado para os fluxos excepcionais.

**Figura 18 -** Comparativo do número de métodos, blocos try e blocos catch por categoria de aplicação.




A Figura 18 apresenta a proporção entre os totais de blocos try e blocos catch com o total de métodos por categoria de aplicação. Analisando o gráfico, percebe-se que há uma disparidade muito grande entre a quantidade de métodos e blocos try e catch. Apesar de a Figura 17 apresentar um dado otimista no que diz respeito a existir um maior número de blocos catch em todas as categorias, na Figura 18 percebe-se que o total de código escrito com o intuito de tornar as aplicações seguras é relativamente muito pequeno.

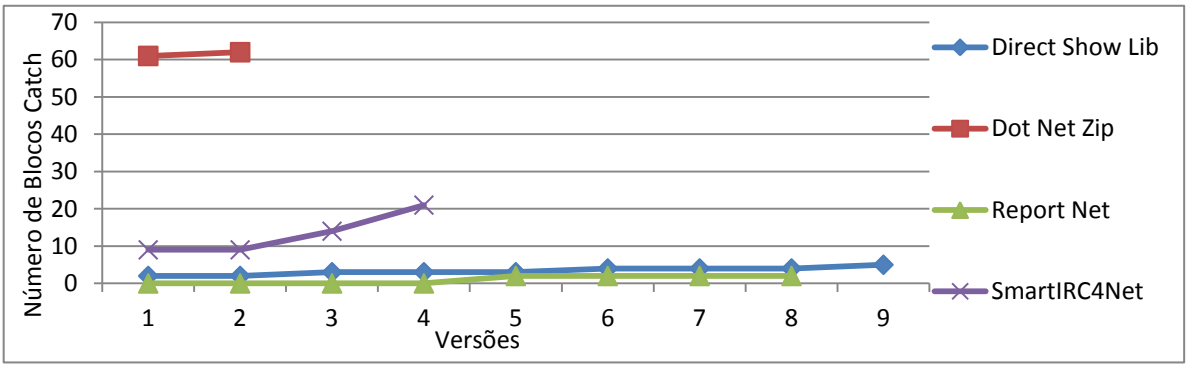
### 5.1.2 Dados de Mudança

Esta seção mostra os dados de mudança coletados através da relação entre métrica e aplicação. Os valores apresentados correspondem a média por versão de cada aplicação, por exemplo, a aplicação SmartIRC4Net possui 3 versões analisadas neste estudo e entre elas foram contabilizadas 102 adições de código excepcionais (EHLocAdded), portanto a média corresponde a 34,00 adições por versão. No fim da tabela há uma coluna denominada "Total" que mostra a média da métrica na categoria correspondente.

| Métricas | Libraries | | | | |
|---|---|---|---|---|---|
| | DirectShow Lib | DotNet Zip | Report Net | Smart IRC4Net | MÉDIA GERAL |
| EHLocAdded | 0,63 | 2,00 | 5,00 | 34,00 | **7,58** |
| EHLocChanged | 0,13 | 2,00 | 7,00 | 2,33 | **3,11** |
| EHLocRemoved | 0 | 7,00 | 1,14 | 5,67 | **1,68** |
| TryBlockAdded | 0,50 | 1,00 | 0,43 | 4,00 | **1,05** |
| TryBlockChanged | 1,13 | 2,00 | 0,43 | 2,00 | **1,05** |
| CatchBlockAdded | 0,38 | 1,00 | 0,29 | 4,33 | **1,00** |
| CatchBlockChanged | 0,25 | 0,00 | 0,00 | 1,00 | **0,26** |
| ClassChurned | 1,1250 | 1,00 | 0,71 | 2,33 | **1,15** |
| MethodChurned | 1,3750 | 2,00 | 0,71 | 4,66 | **1,68** |
| NormalChurnedLOC | 3164,13 | 82,00 | 1024,43 | 2115,33 | **2048,00** |

**Tabela 13 -** Dados de Mudança das aplicações da categoria Libraries.

**Figura 19 -** Evolução da variável NCatch entre as versões das aplicações da categoria Libraries.

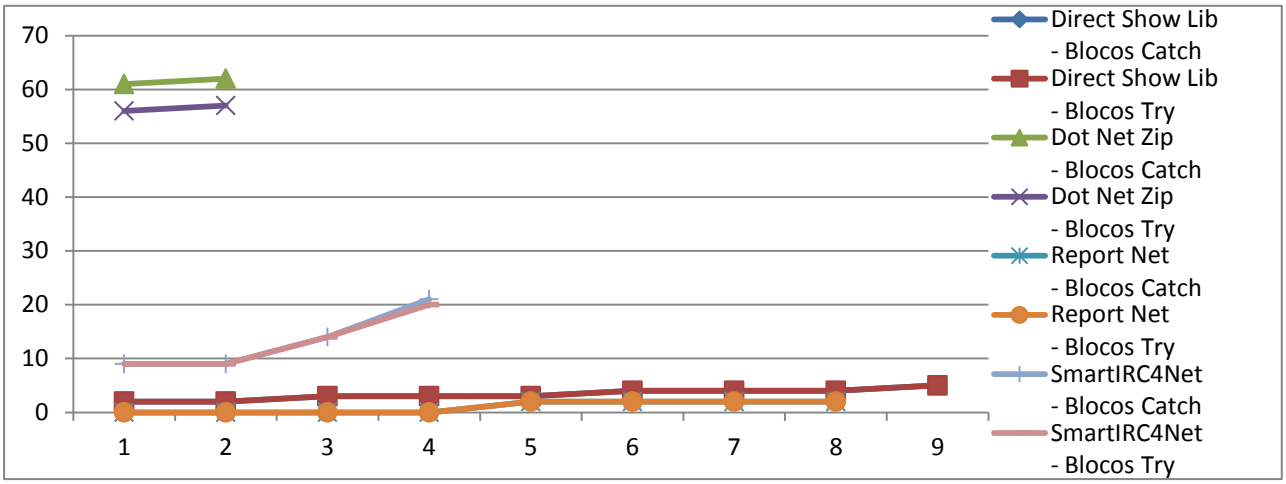



*5.1.2.1 Categoria Libraries*

Na Tabela 13 estão os dados das aplicações da categoria Libraries: (i) DirectShowLib; (ii) DotNetZip; (iii) ReportNet; (iv) SmartIRC4Net. A Figura 19 apresenta um gráfico evolutivo da quantidade de blocos catch dessas aplicações, percebe-se que há um aumento suave na métrica NCatch, mantendo um padrão uniforme desde as versões iniciais, com exceção da aplicação SmartIRC4Net que dobra a quantidade de blocos catch da segunda até a

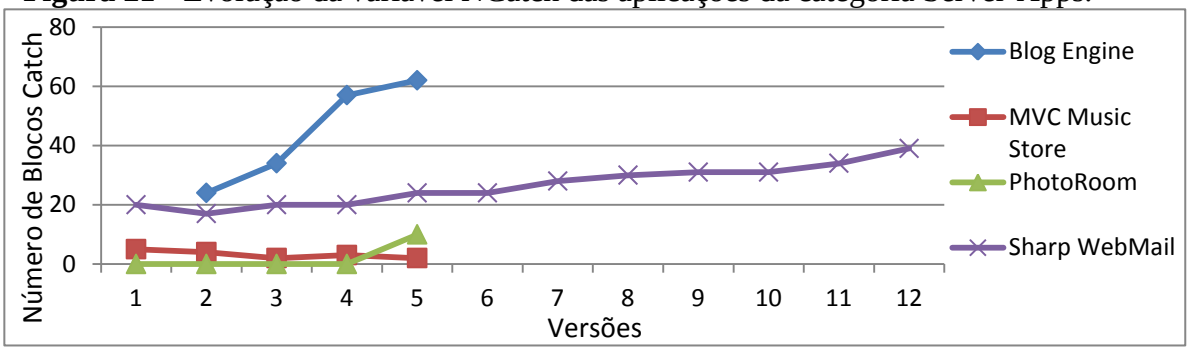
quarta versão. O comportamento evolutivo da métrica NCatch sugere que nessa categoria o esforço de melhoria na robustez dos programas é pequeno, uma vez que não há um padrão de evolução progressiva para essa métrica.

**Figura 20 -** Evolução da variável NCatch e NTry entre as versões das aplicações da categoria Libraries.




A Figura 20 mostra um gráfico da evolução das métricas NTry e NCatch juntas. Percebe-se que a aplicação DotNet Zip possui uma quantidade de Catch maior, porém as duas métricas evoluem na mesma proporção. As outras três aplicações possuem exatamente os mesmos valores de NTry e NCatch, motivo pelo qual as linhas se fundem. A exceção dentre as três aplicações é a aplicação SmartIRC4Net cuja versão 4 apresenta uma ligeira diferenciação no número de NCatch que se apresenta um pouco acima do valor de NTry.

**Figura 21 -** Evolução da variável NCatch das aplicações da categoria Server-Apps.



| Métricas | Server Apps | | | | |
|---|---|---|---|---|---|
| | BlogEngine | MVC Music Store | PhotoRoom | SharpWebMail | MÉDIA GERAL |
| EHLocAdded | 14,25 | 2,25 | 11,75 | 12,00 | **10,65** |
| EHLocChanged | 22,50 | 0,00 | 0,00 | 0,27 | **4,04** |
| EHLocRemoved | 2,00 | 6,00 | 0,00 | 4,09 | **3,35** |
| TryBlockAdded | 4,25 | 0,50 | 2,75 | 2,27 | **2,39** |
| TryBlockChanged | 1,00 | 0,75 | 0,00 | 0,91 | **0,74** |
| CatchBlockAdded | 4,75 | 0,75 | 2,50 | 2,55 | **2,61** |
| CatchBlockChanged | 0,50 | 0,00 | 0,00 | 0,55 | **0,35** |
| ClassChurned | 7,00 | 1,75 | 0,75 | 2,90 | **3,04** |
| MethodChurned | 8,25 | 1,75 | 0,50 | 5,18 | **4,30** |
| NormalChurnedLOC | 1893,00 | 261,00 | 416,25 | 326,82 | **603,30** |

**Tabela 14 -** Dados de Mudança das aplicações da categoria Server-Apps.

*5.1.2.2 Categoria Server-Apps*

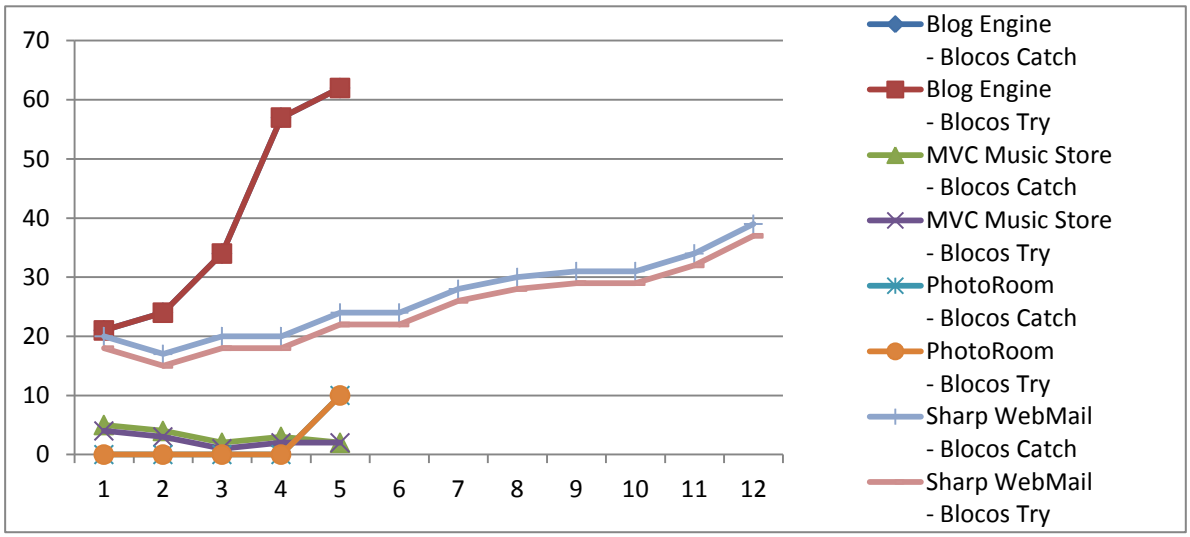
A Tabela 14 mostra os dados das aplicações categorizadas como Server-Apps: (i) BlogEngine; (ii) MVC Music Store; (iii) PhotoRoom; (iv) SharpWebMail. Analisando o gráfico da Figura 21 percebe-se que de modo geral as aplicações dessa categoria tendem a ter um crescimento no número de tratadores, excetuando-se uma queda gradativa na aplicação MVC Music Store.

O gráfico da Figura 22 exibe as linhas evolutivas das métricas NCatch e NTry nas aplicações da categoria Server-Apps. O comportamento dessas métricas também é semelhante. As aplicações BlogEngine e a PhotoRoom possuem exatamente os mesmos valores nas duas métricas em todas as versões. A Sharp WebMail apresenta um crescimento gradual na mesma proporção para as duas métricas, sendo a diferença de valor de 2 pontos a mais na métrica NCatch em todas as verões da aplicação. Por último, a MVCMusic Store apresenta um comportamento semelhante ao da Sharp WebMail, mantendo uma diferença de 1 ponto na métrica NCatch até a última versão, quando os valores se igualam.

A análise dos gráficos da Figura 21 e da Figura 22 mostra, com base no número de blocos try e catch, que houve uma melhoria gradativa no código que trata o comportamento

excepcional através da adição de blocos try e blocos catch, denotando alguma preocupação com a evolução dos softwares de maneira robusta.

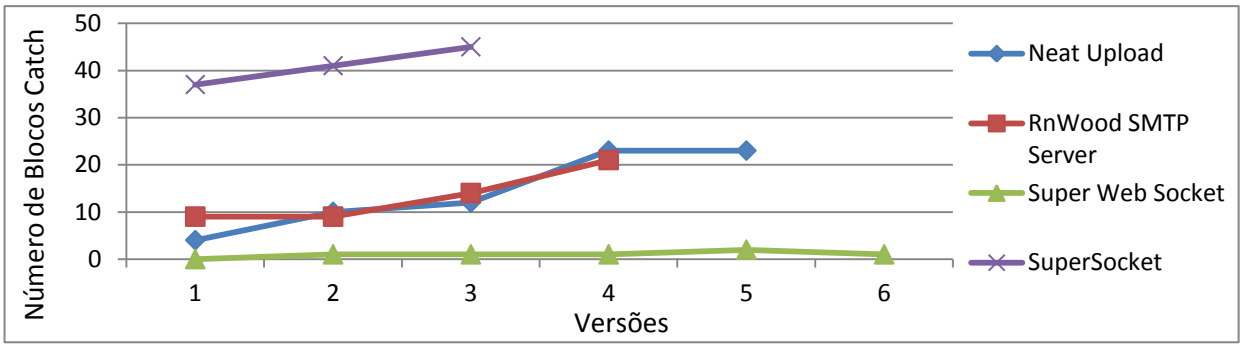
**Figura 22 -** Evolução da variável NCatch e NTry das aplicações da categoria Server-Apps.




| Métricas | Servers | | | | |
|---|---|---|---|---|---|
| | NeatUpload | RnWood SMTP Server | Super Socket | SuperWeb Socket | MÉDIA GERAL |
| EHLocAdded | 6,67 | 4,00 | 26,00 | 1,20 | **7,17** |
| EHLocChanged | 0,00 | 0,50 | 1,00 | 0,00 | **0,25** |
| EHLocRemoved | 0,00 | 1,50 | 8,50 | 0,80 | **2,00** |
| TryBlockAdded | 8,00 | 2,00 | 5,00 | 1,00 | **3,58** |
| TryBlockChanged | 0,00 | 2,00 | 2,50 | 0,20 | **0,83** |
| CatchBlockAdded | 3,67 | 2,50 | 5,00 | 1,00 | **2,58** |
| CatchBlockChanged | 0,00 | 0,50 | 1,00 | 0,00 | **0,25** |
| ClassChurned | 3,66 | 6,00 | | 2,00 | **3,30** |
| MethodChurned | 6,33 | 6,00 | | 2,00 | **4,10** |
| NormalChurnedLOC | 2866,67 | 960,50 | 518,50 | 255,60 | **1069,67** |

**Tabela 15 -** Dados de Mudança das aplicações da categoria Servers.

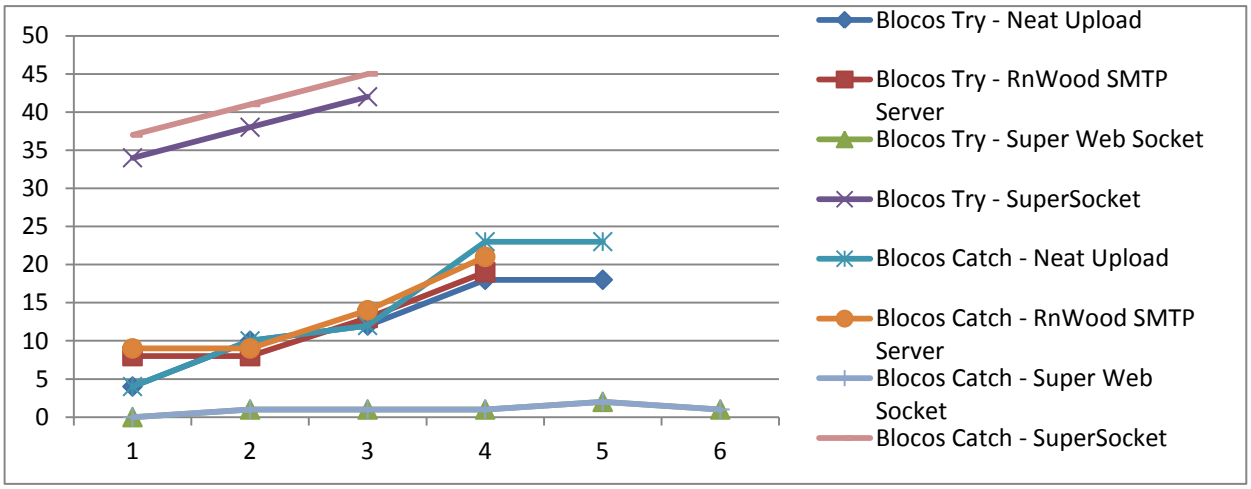
**Figura 23 -** Evolução da variável NCatch das aplicações da categoria Servers.




*5.1.2.3 Categoria Servers*

As aplicações da categoria Servers são apresentadas na Tabela 15. As aplicações são, respectivamente: (i) NeatUpload; (ii) RnWoodSMTP Server; (iii) SuperSocket; (iv) SuperWebSocket. O gráfico evolutivo da métrica NCatch na categoria Server apresentado na Figura 23 mostra uma evolução nessa métrica, exceto na aplicação Super Web Socket, que se mantém com NCatch muito próximo de zero em todas as versões.

**Figura 24 -** Evolução da variável NCatch e NTry das aplicações da categoria Servers.



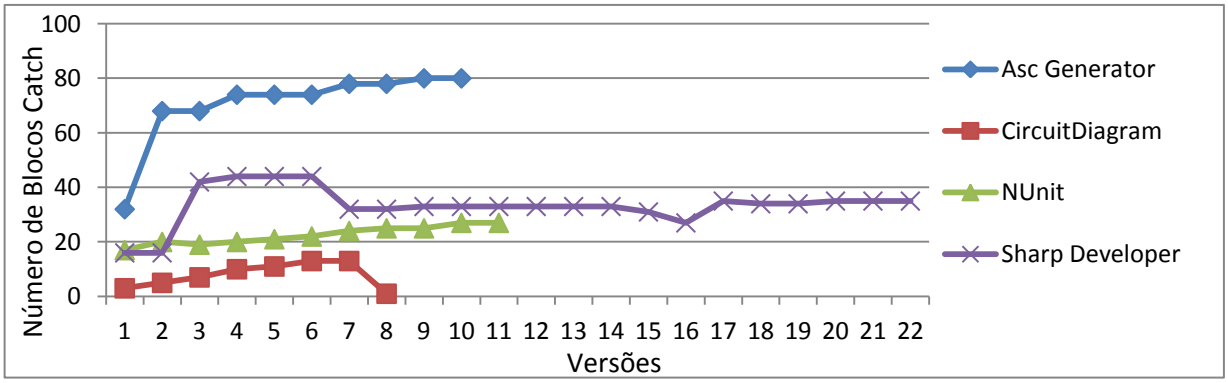

A Figura 24 apresenta um gráfico de linhas contendo a evolução das métricas NTry e NCatch juntas e mais uma vez se pode observar as duas métricas evoluindo juntas, mantendo um padrão da primeira até a última versão das aplicações exceto a Neat Upload que possui valores iguais para as duas métricas nas tres primeiras versões da aplicação e nas duas últimas a NCatch assume um valor 5 pontos maior.

A análise dos gráficos da Figura 23 e da Figura 24 mostra, com base na evolução do número de blocos try e catch, que houve uma melhoria gradativa no código que trata o comportamento excepcional através da adição de blocos try e blocos catch, denotando alguma preocupação com a evolução dos softwares de maneira robusta.

| Métricas | Stand-Alone | | | | |
|---|---|---|---|---|---|
| | Asc Generator | Circuit Diagram | NUnit | SharpDevelop | MÉDIA GERAL |
| EHLocAdded | 8,56 | 5,57 | 15,44 | 13,95 | 11,87 |
| EHLocChanged | 3,56 | 0,14 | 3,33 | 2,55 | 2,53 |
| EHLocRemoved | 1,89 | 6,43 | 2,78 | 13,35 | 7,87 |
| TryBlockAdded | 1,00 | 1,43 | 1,78 | 2,80 | 2,02 |
| TryBlockChanged | 1,89 | 0,14 | 2,11 | 0,30 | 0,96 |
| CatchBlockAdded | 1,33 | 1,43 | 1,89 | 3,00 | 2,20 |
| CatchBlockChanged | 0,67 | 0,00 | 0,89 | 0,70 | 0,62 |
| ClassChurned | 1,22 | 3,57 | 3,77 | 2,60 | 2,71 |
| MethodChurned | 3,11 | 3,57 | 4,88 | 4,95 | 4,35 |
| NormalChurnedLOC | 870,00 | 499,43 | 594,78 | 474,70 | 581,62 |

**Tabela 16 -** Dados de Mudança das aplicações da categoria Stand-Alone.

**Figura 25 -** Evolução da variável NCatch das aplicações da categoria Stand-Alone.

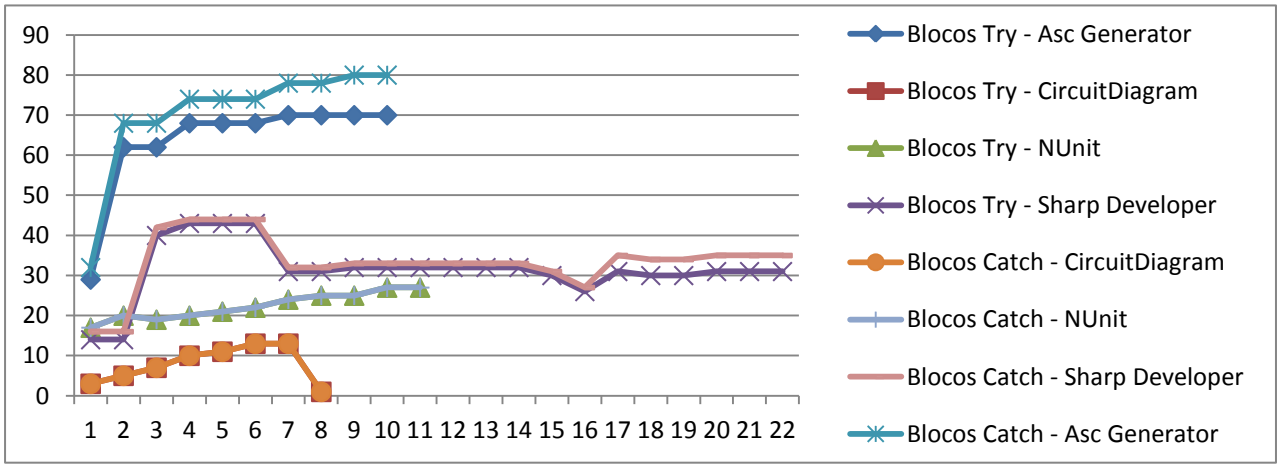



*5.1.2.4 Categoria Stand-Alone*

As médias das métricas de mudança nas aplicações da categoria Stand-Alone são apresentadas na Tabela 16: (i) AscGenerator; (ii) Circuit Diagram; (iii) NUnit; (iv)

SharpDevelop. A Figura 25 exibe um gráfico evolutivo da métrica NCatch em relação as versões de cada aplicação, analisando esses dados percebe-se que o grupo das aplicações Stand-alone é o mais diverso, as aplicações tendem a ter um aumento no número de tratadores, porém há saltos e quedas no decorrer das versões.

**Figura 26 -** Evolução da variável NCatch e NTry das aplicações da categoria Stand-Alone.




A Figura 26 exibe um gráfico de linha com a evolução das métricas NTry e NCatch em relação as versões das aplicações da categoria Stand-Alone. A análise do gráfico demonstra o mesmo comportamento das categorias analisadas nas subseções anteriores. As métricas NTry e NCatch tendem a evoluir nas mesmas proporções e em algumas situações a métrica NCatch alcança níveis ligeiramente superiores.

### 5.1.3 Dados de Robustez

Os dados de robustez estão dispostos na Tabela 17, organizados por aplicação, as quais estão agrupadas por suas respectivas categorias. As colunas apresentam a média da métrica para cada aplicação e ao final de cada categoria, há uma coluna "TOTAL" que representa a média de cada métrica por categoria. Por exemplo: a aplicação DotNetZip possui 2 versões analisadas e uma contagem total de 9 fluxos excepcionais especializados (Specialized Flows), logo possui uma média de 4,50 fluxos dessa natureza. Dentre as 4 aplicações da categoria Libraries, foram analisadas 23 versões, dividindo-se 9 (número de fluxos especializados contabilizados dentre todas essas versões) pela quantidade total de versões da categoria, 23, obtém-se uma média de 0,39 fluxos excepcionais (Specialized Flows) na categoria Libraries.

| Aplicações | | Specialized Flows | Sumbsumption Flows | Uncaught Flows |
|---|---|---|---|---|
| Libraries | DirectShowLib | 0,00 | 2,56 | 91,33 |
| | DotNetZip | 4,50 | 56,50 | 859,00 |
| | ReportNET | 0,00 | 1,00 | 348,00 |
| | SmartIRC4Net | 0,00 | 6,75 | 547,00 |
| | **TOTAL** | **0,39** | **7,43** | **326,61** |
| Server-Apps | BlogEngine | 11,20 | 78,40 | 1701,00 |
| | MVC Music Store | 0,00 | 4,20 | 149,00 |
| | PhotoRoom | 0,00 | 0,60 | 129,80 |
| | SharpWebMail | 0,08 | 1,92 | 113,42 |
| | **TOTAL** | **2,11** | **16,26** | **417,04** |
| Servers | NeatUpload | 0,80 | 22,40 | 230,00 |
| | RnWood SMTP Socket | 0,00 | 0,75 | 32,00 |
| | SuperSocket | 2,00 | 40,33 | 150,67 |
| | SuperWebSocket | 0,00 | 0,00 | 15,00 |
| | **TOTAL** | **0,56** | **13,11** | **101,11** |
| Stand Alone | AscGenerator | 3,00 | 11,50 | 1630,00 |
| | CircuitDiagram | 0,00 | 5,75 | 149,50 |
| | NUnit | 1,00 | 23,09 | 318,18 |
| | SharpDevelop | 4,05 | 12,68 | 748,55 |
| | **TOTAL** | **2,55** | **13,61** | **734,59** |

**Tabela 17 -** Dados de robustez, distribuídos por categoria e aplicação.

A Figura 27 apresenta um gráfico com os totais de possibilidades de fluxos excepcionais por categoria de aplicação. A categoria com a menor incidência de fluxos excepcionais é a das aplicações Servers com 2072 fluxos, após essa está a Libraries com 7692, seguida da Server-Apps com 11756 fluxos. A categoria de aplicações com a maior incidência de fluxos excepcionais é a Stand-Alone com 38288 fluxos.

**Figura 27 -** Totais de fluxos excepcionais por categoria de aplicação.

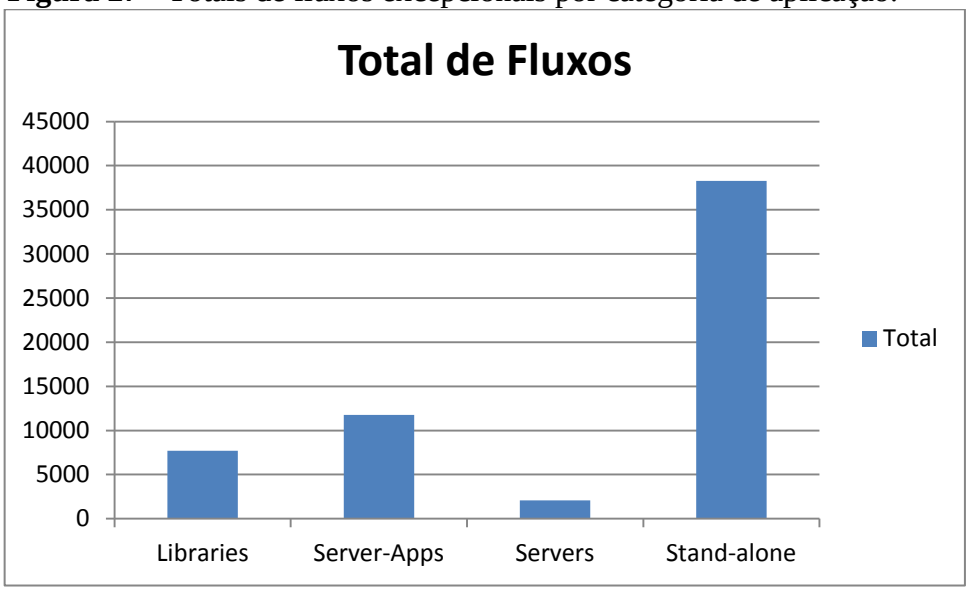
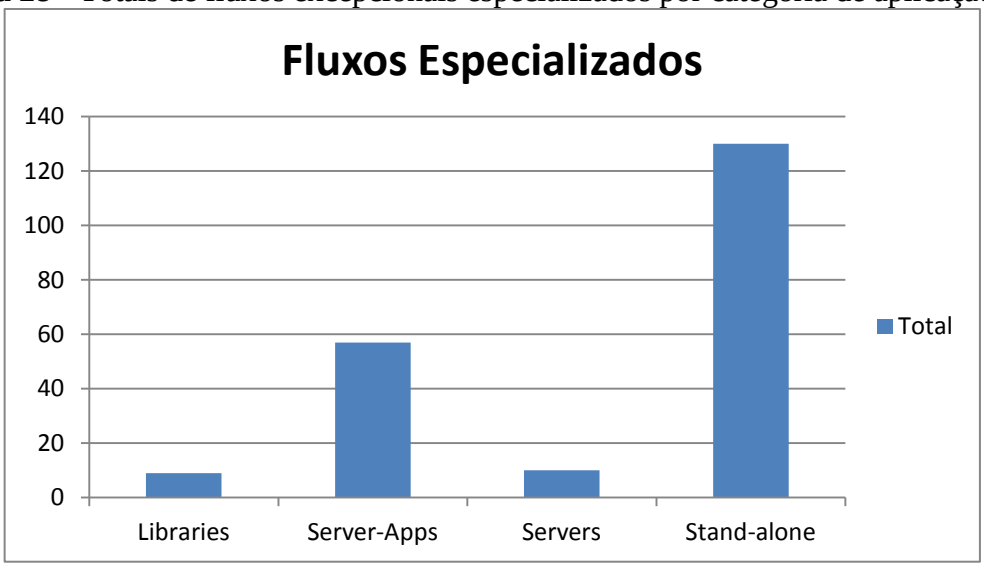
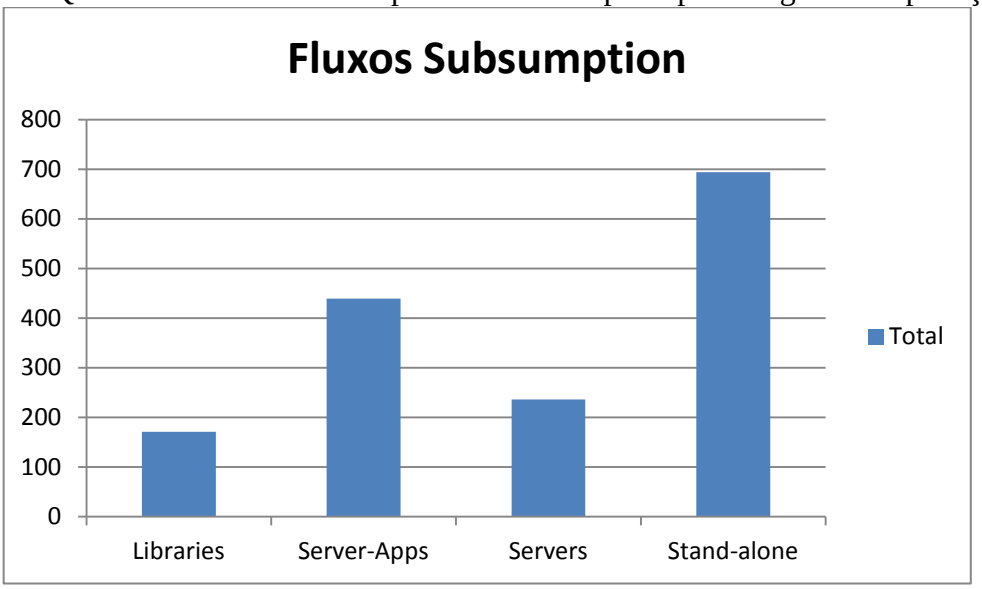



**Figura 28 -** Totais de fluxos excepcionais especializados por categoria de aplicação.



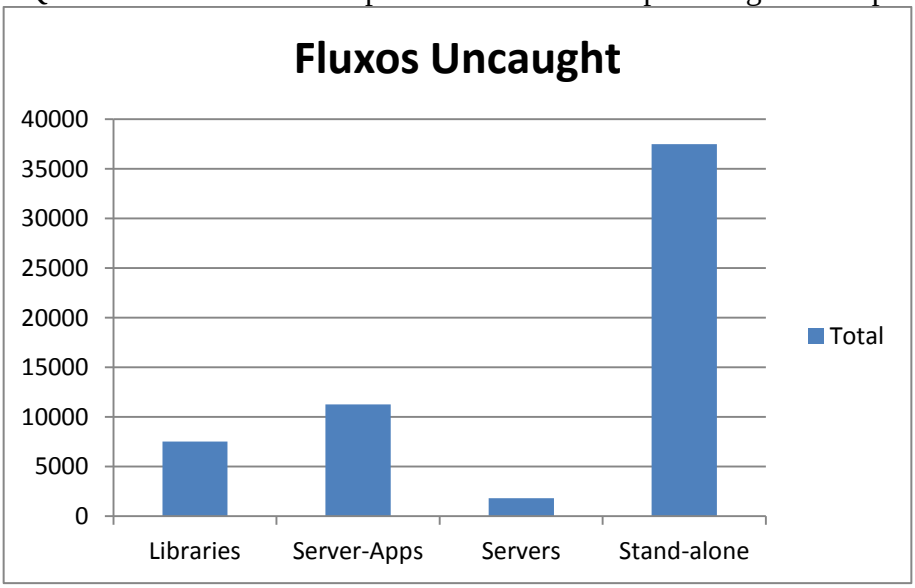

A Figura 28 apresenta um gráfico de barras com a quantidade de fluxos especializados por categoria de aplicação. Esse gráfico permite uma análise, na perspectiva dos fluxos especializados, do gráfico apresentado na Figura 27. A partir desse gráfico pode-se perceber que mesmo possuindo mais que o dobro de fluxos excepcionais da categoria Servers, as aplicações da categoria Libraries possuem o menor número de fluxos especializados. Apenas 0,11% dos fluxos excepcionais são tratados por tratadores específicos nas aplicações da categoria Libraries. Nas aplicações da categoria Server-Apps 0,48% dos fluxos são especializados, o mesmo percentual se repete na categoria Servers. Nas aplicações Stand-Alone esse percentual é de apenas 0,33%.

**Figura 29 -** Quantidade de fluxos excepcionais subsumption por categoria de aplicação.




A Figura 29 apresenta um gráfico com os totais de fluxos tratados com tratadores genéricos, esses dados estão separados por categoria de aplicação. As quantidades estão dispostas de maneira similar aos fluxos especializados, as aplicações da categoria Libraries possuem o menor número de exceções tratadas com tratadores genéricos, seguida pela categoria Servers, depois a Server-Apps e por último, com 694 fluxos desse tipo, as aplicações Stand-Alone. Percentualmente o menor índice de tratamento genérico é da categoria Stand-Alone com 1,81%, seguido pelos 2,22% da Libraries, a Server-Apps vem em terceiro com 3,73% e com o maior percentual está a Servers com 11,38%.

**Figura 30 -** Quantidade de fluxos excepcionais não tratados por categoria de aplicação.



O gráfico apresentado na Figura 30 demonstra as quantidades de fluxos não tratados (uncaught) por categoria de aplicação. A menor quantidade pertence à categoria Servers, que possui também o menor percentual de fluxos não tratados, 87,83%. A segunda menor pertence a categoria Libraries, no entanto seu percentual de 97,65% de exceções não tratadas é maior do que o índice de 95,78% da categoria Server-Apps. O maior índice de exceções não tratadas é da categoria Stand-Alone com 97,84% dos fluxos.

**Figura 31 -** Comparativo entre fluxos uncaught , specialized e subsumption.

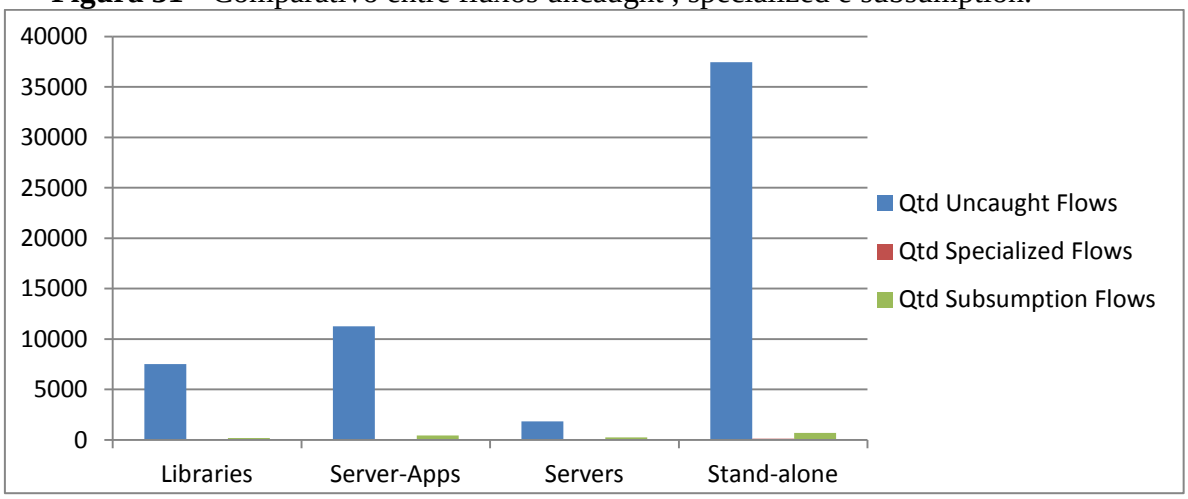
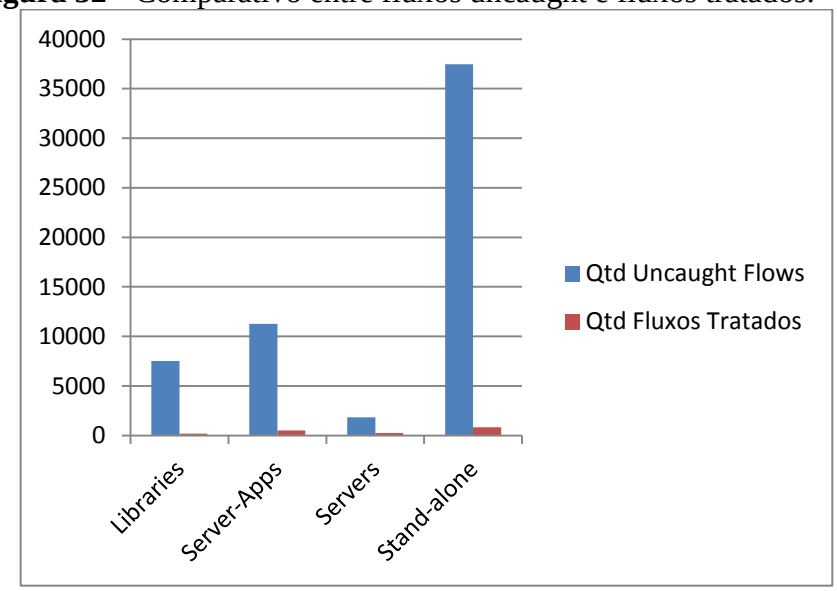



**Figura 32 -** Comparativo entre fluxos uncaught e fluxos tratados.




O baixo índice de fluxos especializados pode ser um indicativo ruim no que se trata da robustez desses softwares, pois uma vez que as exceções não são tratadas especificamente, elas podem não estar sendo tratadas de maneira adequada. No entanto, o gráfico da Figura 31

mostra que a proporção de fluxos genéricos (subsumption) também não é grande o que sinaliza um cenário ainda pior, pois como se pode perceber no gráfico apresentado na Figura 32, o índice de exceções não tratadas (uncaught) é muito elevado.

De todos os fluxos excepcionais das aplicações analisadas, apenas 2,92% deles possuem tratamento. A Figura 32 exibe a proporção de fluxos não tratados para tratados, na categoria Libraries o percentual de fluxos tratados, seja com tratador genérico ou especializado, é de 2,34%, na Server-Apps esse percentual é de 4,21%, na Servers é 12,16% e na Stand-Alone de 2,15%.

## 5.2 Análise das Hipóteses

Para a análise dos dados foram aplicados testes de correlação entre as variáveis, uma vez que os dados não são paramétricos utilizou-se o teste de correlação de Spearman. O teste de Spearman funciona classificando em ranking os dados e aplicando a equação de Pearson a esses rankings. O coeficiente de correlação, r, situa-se no intervalo $-1 \leq r \geq 1$ e indica a correlação entre as variáveis. Assume-se que um coeficiente de correlação r indica uma baixa correlação quando $0.1 < r < 0.3$, uma correlação média quando $0.3 < r < 0.5$, e uma correlação forte quando $0.5 < r < 1.0$. Não há correlação quando $0 < r < 0.1$.

Os resultados possuem também um valor de significância. A significância menor que 0.01 indica que a probabilidade da correlação ser um acaso é de 1%, convencionalmente são aceitas correlações com significância menores ou iguais a 0.05. Nas tabelas abaixo os valores com significancia menor que 0.05 são marcados com um asterisco (*) e os valores com significancia menor que 0.01 são marcados com um duplo asterisco (**).

### 5.2.1 A Relação Entre o Número de Mudanças no Código Normal e o Número de Mudanças no Código Excepcional

O primeiro grupo de hipóteses (Ver Tabela 6), definido na seção 4.2.2 deste trabalho, tem por objetivo elucidar a relação entre o número de mudanças no código que implementa o código normal e o número de mudanças no código excepcional. A hipótese nula **H0** diz que não há relação significante entre o número de mudanças no código que implementa o comportamento normal e o número de mudanças no código excepcional, enquanto a hipótese

alternativa **Ha** diz que, sim, há relação significante entre o número de mudanças no código que implementa o comportamento normal e o número de mudanças no código excepcional.

| Código Normal | Código Excepcional | Categorias de Aplicação | | | | Geral |
|---|---|---|---|---|---|---|
| | | Libraries | Server-Apps | Servers | Stand-Alone | |
| Normal Churned LOC | EH LOC Added | -0,140 | **0,569**** | 0,131 | **0,663**** | **0,350**** |
| | EH LOC Changed | -0,085 | 0,238 | 0,382 | **0,446**** | **0,251*** |
| | EH Loc Removed | -0,138 | 0,373 | 0,71 | **0,436**** | **0,203*** |

**Tabela 18 -** Ranking de correlação de Spearman entre a quantidade de mudanças no código normal e no código excepcional.

Aplicando o teste de correlação de Spearman com as variáveis NormalChurnedLOC, EHLOC Added, EHLOC Changed, EHLOC Removed, obtém-se os resultados apresentados na Tabela 18. Com base nesses resultados pode se concluir que de uma maneira geral (Considerando todas as categorias) a hipótese Ha é a verdadeira, uma vez que há correlações significantes entre a variável NormalChurnedLoc e todas as demais variáveis. Há uma correlação média entre o número de alterações no código normal e a adição de código excepcional, e correlação baixa com a modificação e remoção de código excepcional. A hipótese H0 pode ser rejeitada.

*5.2.1.1 Análise de Cenários*

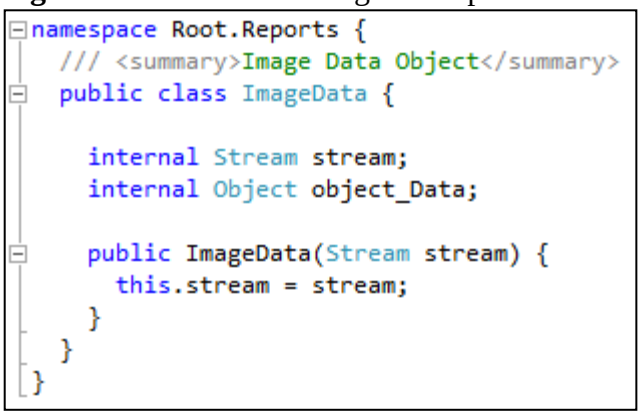
**Correlação nas aplicações por categorias**

As aplicações do grupo “Libraries”, predominantemente, fornecem interfaces, classes abstratas, enumerações, structs, com poucas ocorrências de métodos, ou trechos de código que gerem fluxos excepcionais. Por isso não há uma correlação entre alteração em código normal e o surgimento de novos fluxos excepcionais. A Figura 33 e a Figura 34 mostram exemplos de código extraídos das aplicações Direct Show Lib e Report Net, respectivamente, ambas pertencentes ao grupo das Libraries.

**Figura 33 -** Trecho de código da Direct ShowLib.

```
[Flags]
public enum AMGraphConfigReconnect
{
    DirectConnect = 0x00000001,
    CacheRemovedFilters = 0x00000002,
    UseOnlyCachedFilters = 0x00000004
}
```

**Figura 34 -** Trecho de código da ReportNet.



```
namespace Root.Reports {
  /// <summary>Image Data Object</summary>
  public class ImageData {

    internal Stream stream;
    internal Object object_Data;

    public ImageData(Stream stream) {
      this.stream = stream;
    }
  }
}
```

No caso do Server-Apps, todo o impacto da adição de código normal se refletiu na adição de blocos catch. A Figura 23 mostra que três das quatro aplicações desta categoria duplicaram o número de blocos catchs, ou seja, o EH LOC Added. Os resultados da Tabela 18 indicam que nas aplicações de servidor (server-apps) e nas desktop (stand-alone) há uma correlação forte entre alterações no código normal e a adição de código excepcional, o que indica que há uma maior atenção dos desenvolvedores quando se trata dessas duas categorias de aplicações. Os coeficientes de correlação das aplicações desktop (stand-alone) indicam uma correlação média entre alterações no código normal e alterações ou remoções no código excepcional dessas aplicações, o que reforça a atenção especial dos desenvolvedores no desenvolvimento dessa categoria de aplicações.

### Cenários de modificação

A Tabela 19 apresenta a relação entre os cenários de alterações de código normal e código excepcional exibindo a frequencia de ocorrências de alterações conjuntas de código normal e excepcional. As alterações no código normal (somente bloco try) que não tiveram alterações conjuntas de código excepcional estão dispostas na última linha, ou seja, a linha que cruza qualquer alteração no código normal com a linha "Nenhuma alteração no bloco catch". As alterações no bloco catch (código excepcional) sem alteração conjunta de código normal estão dispostas na última coluna da tabela - "Nenhuma alteração no bloco try".

Com base nesses dados pode se obter os cenários mais comuns analisados neste estudo. Pode se perceber, por exemplo que há tres cenários de modificação de código normal que não possuem alterações de código excepcional: (i) Modificação de variável; (ii) Modificação de fluxo de controle; (iii) Adição de invocação a um método. Uma vez que .NET não utiliza exceções checadas, a ausencia de alterações no código excepcional quando há novas

invocações a métodos pode ser um fator de risco, uma vez que adicionando novas invocações novos fluxos de execução surgem e dentre eles podem surgir fluxos excepcionais. Em 50,9% das novas invocações não houve alteração no código excepcional. A seção 5.2.2 apresenta a relação entre alterações no código normal e o nível de robustez das aplicações.

| **Mudanças no código excepcional** | **Mudanças no código normal (somente blocos try)** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Bloco try adicionado a um método existente | Novo método adicionado contendo um bloco try | Nova invocação a um método adicionada | Variável modificada | Fluxo de controle modificado | Bloco try removido | Método contendo bloco try removido | Nenhuma alteração no bloco try |
| Tratador genérico adicionado | 19.5% | 64.4% | 9.4% | - | 3.6% | - | - | - |
| Tratador vazio adicionado | 24.4% | 16.5% | 3.8% | - | - | - | - | - |
| Tratador genérico adicionado com rethrow | 34.1% | 3.1% | 3.8% | - | - | - | - | - |
| Tratador genérico removido | - | - | - | - | 3.6% | 87.0% | 96.8% | 7.7% |
| Tratador especializado adicionado | 12.2% | 10.2% | 3.8% | - | - | - | - | 3.8% |
| Tratador especializado vazio adicionado | 2.4% | 0.8% | - | - | - | - | - | 3.8% |
| Tratador especializado adicionado com rethrow | 7.3% | 4.7% | 1.9% | - | - | - | - | - |
| Tratador especializado removido | - | - | - | - | - | 13.0% | 3.2% | - |
| Mudança na política de tratamento de exceção | - | - | 11.3% | 8.3% | 7.1% | - | - | 53.8% |
| Mudança no bloco catch para usar código normal | - | - | 15.1% | 50.0% | 3.6% | - | - | 30.8% |
| Nenhuma alteração no bloco catch | - | - | 50.9% | 41.7% | 82.1% | - | - | - |
| Percentual | 11.0% | 34.0% | 14.5% | 3.2% | 7.5% | 6.1% | 16.8% | 7.0% |

**Tabela 19 -** Ocorrências de cenários de mudança de códigos normal e excepcional.

O cenário de adição de código normal mais frequente é a adição de bloco try. Em 11% das alterações, foram adicionados blocos try a métodos já existentes (coluna um), e em 34% novos métodos foram adicionados com blocos try (coluna dois). A maior parte das correspondencias no comportamento excepcional, no entanto, foram referentes a adições de blocos catch genéricos. Dentre as adições de bloco try em métodos existentes, 19,5% dos casos foram acompanhados da adição de um tratador genérico, 24,4% tiveram tratadores genéricos vazios adicionados e 34,1% dos casos foram acompanhados de tratadores genéricos que apenas repropagavam as exceções capturadas (rethrow). Para os casos de adição de métodos novos contendo blocos try os dados mostram que 64,6% deles possuem tratadores genéricos e 16,5% possuem tratadores genéricos vazios.

Em 7% dos cenários houve alteração no código excepcional sem alteração no código normal. Mais da metade desses casos correspondem a alterações na política de tratamento de exceções (53,8%) e em 30,8% alterações pontuais dentro do bloco catch. 7.7% dos casos tratam da remoção de tratadores genéricos enquanto 3,6% são de adições de tratadores específicos e mais 3,6% de tratadores específicos vazios. Ou seja 7,6% são adições de tratadores especializados indicando uma pequena preocupação tardia na melhoria do código de tratamento de exceções.

### 5.2.2 A Relação Entre o Número de Mudanças no Código Normal e o Nível de Robustez das Aplicações

O segundo grupo de hipóteses (Ver Tabela 7), definido na seção 4.2.2 deste trabalho, é direcionado à relação, ou ausência dela, entre o volume de mudanças no código normal de uma aplicação e o seu nível de robustez. A hipótese inicial, **H0**, diz que não existe relação significante entre o número de mudanças no código que implementa o comportamento normal e o número de fluxos excepcionais não tratados (uncaught flows). Por outro lado a hipótese alternativa, **Ha**, diz que há relação significante entre o número de mudanças no código normal e o número de fluxos excepcionais não tratados.

Conforme descrito na seção 4.2.3.1, neste estudo as informações dos fluxos excepcionais são consideradas para análise de robustez das aplicações. Foi aplicado o teste de correlação de Spearman com as variáveis NormalChurnedLOC e as tres variáveis de robustez: (i) Specialized Flow; (ii) Subsumption Flow; (iii) Uncaught Flow. Os coeficientes de correlação estão dispostos na Tabela 20. Com base nesses resultados pode se concluir que de uma

maneira geral (dentre as categorias analisadas) a hipótese Ha é a verdadeira, uma vez que há uma correlação média e significante entre a variável NormalChurnedLoc e a variável Uncaught Flow. A hipótese H0, portanto, pode ser rejeitada.

*5.2.2.1 Análise de Cenários*

A seção 5.2.1 conclui que há uma correlação entre alterações no código normal e alterações no código excepcional, no entanto, os dados da Tabela 20 não mostram uma correlação entre essas alterações no código normal com o aumento de fluxos especializados (specialized flows), por outro lado percebe-se uma correlação com o surgimento de fluxos subsumption e fluxos não tratados (uncaught flows), o que sugere que alterações no código normal diminuem a robustez.

| Código Normal | Fluxos Excepcionais | Categorias de Aplicação | | | | Total |
|---|---|---|---|---|---|---|
| | | Libraries | Server-Apps | Servers | Stand-Alone | |
| Normal Churned LOC | Specialized Flow | -0,344 | -0,221 | 0,480 | 0,176 | 0,041 |
| | Subsumption Flow | -0,063 | **0,621**** | 0,296 | **0,304*** | **0,262**** |
| | Uncaught Flow | -0,054 | 0,315 | **0,578*** | **0,656**** | **0,437**** |

**Tabela 20 -** Ranking de correlação de Spearman entre a quantidade de mudanças no código normal e o número de fluxos excepcionais em suas categorias.

As aplicações da categoria Server-Apps possuem uma alta correlação entre alterações no código normal e fluxos subsumption, as aplicações das categorias Servers e Stand-Alone possuem uma alta correlação com os fluxos não tratados. Além das categorias das aplicações, os tipos de modificações no código normal também influenciam no nível de correlação com a robustez das aplicações, nas subseções a seguir discutiremos tais pontos.

**Correlação entre alterações no código normal e exceções do tipo subsumption**

Segundo a Tabela 20, há uma correlação baixa e significante entre o número de alterações no código normal e fluxos excepcionais do tipo subsumption, o que pode ser considerado outro indicativo negativo do nível de robustez, uma vez que os fluxos estão sendo tratados de forma genérica, não observando-se as particularidades e demandas específicas de cada fluxo.

Analisando-se as amostras utilizadas no estudo constata-se que ao adicionar um bloco try, os programadores em muitos casos declaram um bloco catch com tipo genérico. A Figura 35 mostra o caso da aplicação ReportNet em sua versão 0.09.00. Dois blocos try foram adicionados nessa versão e ambos foram declarados com tipos genéricos em seus tratadores: “System.Exception” e “System.SystemException”. Dessa forma, qualquer exceção lançada dentro dos blocos try adicionados serão capturados por tipos genéricos, aumentando, portanto, o número de fluxos subsumption.

**Figura 35 -** Trecho de código da aplicação Report.NET, versão 0.09.00

```
226     private static void StartAcro(String sFileName, String sArguments, ProcessWindowStyle processWindowStyle) {
227       RegistryKey registryKey_Acro = null;
228       try {
229         registryKey_Acro = Registry.ClassesRoot.OpenSubKey(sRegKey_AcrobatReader, false);
230       }
231       catch (SystemException ex) {
232         throw new ReportException(rm.GetString("e_AcroRegistryInvalid"), ex);
233       }
234       if (registryKey_Acro == null) {
235         throw new ReportException(rm.GetString("e_AcroNotInstalled"));
236       }
237       String sAcroPath = (String)registryKey_Acro.GetValue("");
238       if (sAcroPath == null || sAcroPath == "") {
239         throw new ReportException(rm.GetString("e_AcroRegistryInvalid"));
240       }
241       sAcroPath = sAcroPath.Replace("\"", "");
242
243       try {
244         Process process = new Process();
245         process.StartInfo.FileName = sAcroPath;
246         if (sArguments == null) {
247           process.StartInfo.Arguments = sFileName;
248         }
249         else {
250           process.StartInfo.Arguments = sArguments + " " + sFileName;
251         }
252         process.StartInfo.WindowStyle = processWindowStyle;
253         process.Start();
254       }
255       catch (Exception ex) {
256         throw new ReportException(String.Format(rm.GetString("e_AcroStartProcess"), sFileName, sAcroPath), ex);
257       }
258     }
```

### Correlação entre alterações no código normal e exceções do tipo uncaught

Adicionalmente a Tabela 20 mostra uma correlação significante entre as mudanças nos blocos try e o surgimento de fluxos excepcionais do tipo uncaught. Isso ocorre quando em um método a aplicação não utiliza tratadores, ou utiliza tratadores especializados e com a modificação do bloco try não são acrescentados tratadores para atender aos novos potenciais fluxos.

A Figura 36 exibe um exemplo da aplicação AscGenerator. Em sua versão 0.7.1 o método FormConvertImage.ReadXML possuía um bloco try e dois tratadores para as exceções dos tipos “System.Xml.Exception” e “System.IO.FileNotFoundException”. Em sua versão 0.7.2 o mesmo bloco try teve apenas uma linha acrescentada que faz referência a um

método que faz requisições web e manipula arquivos. Logo novos fluxos passaram a existir a partir dessa alteração, mas nenhum tratador novo foi adicionado.

**Figura 36 -** Evolução do método FormConvertImage.ReadXML da aplicação AscGenerator (versões 0.7.1 e 0.7.2).

```
private bool ReadXML(string filename) {
    XmlDocument doc = new XmlDocument();

    try {
        doc.Load(filename);

        // check doc.DocumentElement.Attributes["version"]

        Variables.InitialInputDirectory =
        ReadNode(doc.SelectSingleNode("ascgen2/input/initialdirectory"),
            Variables.InitialInputDirectory, true);

        .
        .
        .
        .
    }
    catch(System.Xml.XmlException) {
        MessageBox.Show(Resource.GetString("Invalid settings file.
            Please delete it or save a new one"),
        Resource.GetString("Error"), MessageBoxButtons.OK, MessageBoxIcon.Error);
        return false;
    }
    catch (System.IO.FileNotFoundException) {
        return false;
    }
}
```

```
private bool ReadXML(string filename) {
    XmlDocument doc = new XmlDocument();

    try {
        doc.Load(filename);

        // check doc.DocumentElement.Attributes["version"]

        Variables.CheckForNewVersions = ReadNode(doc.SelectSingleNode(
            "ascgen2/checkfornewversions"), Variables.CheckForNewVersions);

        Variables.InitialInputDirectory =
            ReadNode(doc.SelectSingleNode("ascgen2/input/initialdirectory"),
            Variables.InitialInputDirectory, true);
        .
        .
        .
        .
    }
    catch(System.Xml.XmlException) {
        MessageBox.Show(Resource.GetString("Invalid settings file.
            Please delete it or save a new one"),
        Resource.GetString("Error"), MessageBoxButtons.OK, MessageBoxIcon.Error);
        return false;
    }
    catch (System.IO.FileNotFoundException) {
        return false;
    }
}
```

### Cenários frequentes de modificação

Como citado na seção 5.2.1, a Tabela 19 mostra que há casos frequentes de alteração de código normal sem uma alteração de código excepcional. Três tipos de modificação se destacam: (i) Modificação de variável; (ii) Modificação de fluxo de controle; (iii) Adição de invocação a um método. A análise detalhada dessas ocorrencias demonstra que adições de novas invocações a métodos e modificações de variáveis aumentam a ocorrência de exceções não tratadas em 100% dos casos, conforme detalha a Tabela 21.

| | Nova invocação a um método adicionada | Modificações de Variáveis | Fluxo de controle modificado | Total |
|---|---|---|---|---|
| Diminuição em Uncaught Flow | - | - | 50% | 25% |
| Aumento em Uncaught Flow | 100% | 100% | 50% | 75% |

**Tabela 21 -** Impacto das mudanças no código normal nos fluxos excepcionais não tratados.

O mecanismo de tratamento de exceção utilizado pela linguagem de programação C#, por ser dirigido a manutenibilidade, se mostra suscetível a falhas em casos semelhantes a esses. Com o objetivo de facilitar a manutenção dos softwares, o mecanismo implementado por C# não utiliza interfaces de exceção (ver seção 2.6.2), mecanismo que demanda que o desenvolvedor declare as exceções lançadas na interface do método. Através das interfaces de

exceção algumas linguagens validam e exigem que o desenvolvedor trate as exceções que os métodos invocados podem lançar. Sem essa checagem automática baseada nas interfaces de exceção os desenvolvedores não podem afirmar se uma nova exceção surgirá a partir de uma modificação e isso ocasiona muitos fluxos não tratados.

### 5.2.3 A Relação Entre o Número de Mudanças no Código Excepcional e o Nível de Robustez das Aplicações

Esta seção analisa o terceiro grupo de hipóteses (Ver Tabela 8), definido na seção 4.2.2 deste trabalho, que trata do impacto das alterações em código excepcional no nível de robustez das aplicações. A hipótese inicial, **H0**, diz que não existe relação significante entre o número de mudanças no código que implementa o comportamento excepcional e o número de fluxos excepcionais não tratados (uncaught flows). Por outro lado a hipótese alternativa, **Ha**, diz que há relação significante entre o número de mudanças no código excepcional e o número de fluxos excepcionais não tratados.

| Código Excepcional | Fluxos Excepcionais | Categorias de Aplicação | | | | Total |
|---|---|---|---|---|---|---|
| | | Libraries | Server-Apps | Servers | Stand-Alone | |
| Catch Block Added | Specialized Flow | 0,225 | -0,247 | -0,134 | 0,030 | -0,048 |
| | Subsumption Flow | **0,575*** | **0,685**** | **0,621*** | **0,413**** | **0,548**** |
| | Uncaught Flow | **0,482*** | 0,292 | **0,641*** | **0,419**** | **0,405**** |

**Tabela 22 -** Ranking de correlação de Spearman entre a quantidade de mudanças no código excepcional e o número de fluxos excepcionais em suas categorias.

Aplicamos o teste de correlação de Spearman utilizando as variáveis "Catch Block Added" e as tres variáveis de robustez: (i) Specialized Flow; (ii) Subsumption Flow; (iii) Uncaught Flow. Os coeficientes de correlação são apresentados na Tabela 22, pelos quais pode-se concluir que há uma correlação alta entre a adição de código excepcional e a ocorrência de fluxos subsumption e uma correlação média com a ocorrência de fluxos uncaught. Dessa forma podemos rejeitar a hipótese H0, uma vez que a hipótese alternativa é verdadeira.

*5.2.3.1 Análise de Cenários*

**Relação entre mudanças no código excepcional e ocorrências de fluxos subsumption**

A correlação apresentada entre a adição de um bloco catch e o surgimento de fluxos sumbsumption pode ser percebida em diversos cenários. Um deles é quando um novo código é adicionado à aplicação, como uma classe, ou método, e nesse código há uma chamada que origina um fluxo excepcional e um bloco catch genérico que captura essa exceção. Outra situação comum é quando um código que lança, ou chama outros métodos que lançam exceções, mas não trata tais fluxos passa a ter um tratador genérico. Nesse último caso não é um novo fluxo excepcional que surge com a adição de um bloco catch, mas um antigo fluxo que não estava sendo tratado que passa a ser tratado genericamente.

**Figura 37 -** Método AjaxHelper.UpdateExtensionSourceCode na versão 2.5 da aplicação BlogEngine.

```
public static bool UpdateExtensionSourceCode(string sourceCode, string extensionName)
{
    if (!WebUtils.CheckRightsForAdminSettingsPage(false)) { return false; }
    if (!WebUtils.CheckIfPrimaryBlog(false)) { return false; }
    if (string.IsNullOrWhiteSpace(extensionName)) { return false; }
    var ext = ExtensionManager.GetExtension(extensionName);
    if (ext == null) { return false; }
    string extensionFilename = ext.GetPathAndFilename(true);
    if (string.IsNullOrWhiteSpace(extensionFilename)) { return false; }
    try{
        using (var f = File.CreateText(extensionFilename)) {
            f.Write(sourceCode);
            f.Close();
            return true;
        }
    }
    catch (Exception ex){
        Utils.Log("Error saving source code: " + ex.Message);
        return false;
    }
}
```

**Figura 38 -** Adição de try-catch no método CapacitorEditor.UpdateChanges da versão 0.1 para a 0.2 da CircuitDiagram.

```
public override void UpdateChanges( EComponent component)
{
  ((Capacitor)component).Capacitance =
double.Parse(tbxCapacitance.Text);
}
```

```
public override void UpdateChanges(EComponent component)
{
  Try
  {
    ((Capacitor)component).Capacitance =
double.Parse(tbxCapacitance.Text);
  }
  catch (Exception)
  {
    // incorrect input format
  }
}
```

A Figura 37 exemplifica o primeiro caso citado, o método "AjaxHelper.UpdateExtensionSourceCode" não existia na versão anterior, a 2.0, e com a nova versão foi implementado esse método que origina novos fluxos excepcionais e os trata com um handler genérico. A Figura 38, por sua vez, exemplifica o segundo caso, na qual é apresentado o método "CapacitorEditor.UpdateChanges" e sua evolução da versão 0.1 para a 0.2. Percebe-se na imagem que o código normal do método é exatamente o mesmo, mas foi posto dentro de um bloco try, cujo respectivo catch é genérico. Pode-se, então, concluir que na versão 0.1 haviam possíveis fluxos não tratados que passaram a ser tratados genericamente.

### Relação entre mudanças no código excepcional e ocorrências de fluxos não tratados (uncaught flows)

Cada cenário de modificação de código possui impactos nos fluxos do programa, especialmente alguns cenários de modificação possuem impactos positivos ou negativos na quantidade dos fluxos excepcionais não tratados. Até mesmo a adição de tratadores pode impactar negativamente a robustez das aplicações, conforme se observa na Tabela 23, existem casos de adições de tratadores genéricos que apenas relançam as exceções capturadas, esses casos são responsáveis por 28,9% dos surgimentos de exceções não tratadas. A adição de tratadores especializados acarreta em um vazamento de exceções de outros tipos, gerando 23,7% das exceções não tratadas pela aplicação. 13,2% desses fluxos não tratados são também gerados a partir da adição de tratadores especializados que apenas relançam exceções. Como esses, existem outros cenários que impactam nos fluxos excepcionais da aplicação, afetando sua robustez. Alguns desses casos são comentados a seguir.

Analisando a Tabela 24, pode-se perceber que a adição de tratadores genéricos com relançamento de exceções fez crescer o número de fluxos excepcionais não tratados em 100% dos casos observados. Em 90% dos casos de adição de tratadores especializados, houve também um aumento no número de fluxos não tratados. Alterações no código normal que não foram acompanhadas de alterações no bloco catch geraram novos fluxos não tratados em 75% dos casos (última linha da Tabela 24).

| **Mudanças no código excepcional** | Libraries | Server-Apps | Servers | Stand-Alone | Percentual de Exceções não tratadas |
|---|---|---|---|---|---|
| Tratador genérico adicionado com rethrow | 2.6% | 2.6% | 23.7% | - | 28.9% |
| Tratador especializado adicionado | 2.6% | - | - | 21.1% | 23.7% |
| Nenhuma alteração no bloco catch | 5.3% | 5.3% | - | 5.3% | 15.8% |
| Tratador especializado adicionado com rethrow | - | 2.6% | 7.9% | 2.6% | 13.2% |
| Mudança na política de tratamento de exceção | - | - | 5.3% | 2.6% | 7.9% |
| Mudança no bloco catch para usar código normal | 7.9% | - | - | - | 7.9% |
| Tratador genérico removido | - | - | - | 2.6% | 2.6% |

**Tabela 23 -** Percentual de cenários de mudança afetando a ocorrência de fluxos não tratados.

| **Mudanças no código excepcional** | Crescimento no número de fluxos excepcionais | Decrescimento no número de fluxos excepcionais |
|---|---|---|
| Tratador genérico adicionado | - | 100% |
| Tratador genérico vazio adicionado | - | 100% |
| Tratador genérico adicionado com rethrow | 100% | - |
| Tratador genérico removido | 7.1% | 92.9% |
| Tratador especializado adicionado | 90% | 10% |
| Mudança na política de tratamento de exceção | 75% | 25% |
| Mudança no bloco catch para usar código normal | 100% | - |
| Nenhuma alteração no bloco catch | 75% | 25% |

**Tabela 24 -** Percentual de cenários de mudança aumentando ou diminuindo a ocorrencia de fluxos não tratados.

**Figura 39** - Método Initialize na versão 2 da Super Web Socket.

```
39  public bool Initialize(IServerConfig config)
40  {
41      if (m_CommandAssembly != null)
42          return true;
43      var commandAssembly = config.Parameters.GetValue("commandAssembly");
44
45      if (string.IsNullOrEmpty(commandAssembly))
46          return true;
47      try
48      {
49          m_CommandAssembly = Assembly.Load(commandAssembly);
50          return true;
51      }
52      catch (Exception e)
53      {
54          LogUtil.LogError(e);
55          return false;
56      }
57  }
```

**Figura 40** - Método StartAcro na versão 0.09.00 da ReportNet.

```
226   private static void StartAcro(String sFileName, String sArguments, ProcessWindowStyle processWindowStyle) {
227     RegistryKey registryKey_Acro = null;
228     try {
229       registryKey_Acro = Registry.ClassesRoot.OpenSubKey(sRegKey_AcrobatReader, false);
230     }
231     catch (SystemException ex) {
232       throw new ReportException(rm.GetString("e_AcroRegistryInvalid"), ex);
233     }
234     if (registryKey_Acro == null) {
235       throw new ReportException(rm.GetString("e_AcroNotInstalled"));
236     }
237     String sAcroPath = (String)registryKey_Acro.GetValue("");
238     if (sAcroPath == null || sAcroPath == "") {
239       throw new ReportException(rm.GetString("e_AcroRegistryInvalid"));
240     }
241     sAcroPath = sAcroPath.Replace("\"", "");
242
243     try {
244       Process process = new Process();
245       process.StartInfo.FileName = sAcroPath;
246       if (sArguments == null) {
247         process.StartInfo.Arguments = sFileName;
248       }
249       else {
250         process.StartInfo.Arguments = sArguments + " " + sFileName;
251       }
252       process.StartInfo.WindowStyle = processWindowStyle;
253       process.Start();
254     }
255     catch (Exception ex) {
256       throw new ReportException(String.Format(rm.GetString("e_AcroStartProcess"), sFileName, sAcroPath), ex);
257     }
258   }
```

Conforme apresenta a Tabela 22, há uma correlação média entre a adição de blocos catch e a ocorrencia de fluxos excepcionais não tratados. Ao investigar esse dado, pode-se perceber alguns fatos interessantes. Focando nas aplicações da categoria “Servers”, por exemplo, vê-se que a Super web Socket e a RnWood SMTP Server, possuem uma correlação elevada. Entre as versões 1 e 2 da Super Web Socket foram adicionados blocos catch que registram a ocorrência de uma exceção em log. Para isso, é executado o método LogUtil.LogError (ver Figura 39), porém esse método potencialmente lança dois tipos de exceção e não foi inserido

em um bloco seguro. O método LogUtil.LogError lança as seguintes exceções: “System.ArgumentNullException” e “System.OutOfMemoryException”.

A aplicação ReportNet, pertencente ao grupo “Libraries” pode também servir de exemplo para a correlação entre a adição de blocos catch e o surgimento de novos fluxos excepcionais não tratados. Há vários blocos catch que apenas lançam uma nova exceção do tipo “Root.Reports.ReportException” sem que haja tratador algum no local, ou nos demais níveis da pilha de chamadas. Um exemplo disso pode ser visto na Figura 40.

# 6. Conclusão

Mecanismos de tratamento de exceção dirigidos a manutenção tem se tornado cada vez mais populares em linguagens de programação, tais como C#. Melhorar a robustez dos programas é a motivação principal para utilizar esses mecanismos. Entretanto, existe pouco conhecimento empírico acerca do impacto de mecanismos de tratamento de exceção dirigidos a manutenção no processo de evolução dos sistemas de software. Este estudo sugere que programadores C# frequentemente trocam robustez por manutenibilidade em muitas categorias de aplicação. Os programadores desenvolveram seus programas seguindo diferentes padrões de mudança para o código de tratamento de exceção, uma vez que o mecanismo de tratamento de exceção de C# facilita a realização de mudanças.

O uso de mecanismo de tratamento de exceção em C#, no entanto, mostrou-se frágil e frequentemente levou a um decréscimo na robustez dos programas. Programadores frequentemente introduziram bugs quando foram realizadas alterações sutis nos blocos try. Um grande número de fluxos excepcionais não tratados foram introduzidos quando blocos catch foram modificados. Esses achados sugerem que ainda existe muitas lacunas a serem preenchidas para o melhoramento dos mecanismos de tratamento de exceção nativos das liguagens de programação.

Este estudo investigou o impacto de atividades de manutenção na robustez de softwares investigados. Para tal definimos e testamos três grupos de hipóteses, o primeiro grupo avalia o efeito do número de mudanças no comportamento normal das aplicações no número de mudanças no código excepcional. O segundo grupo testa o efeito do número de mudanças no código responsável pelo comportamento normal no nível de robustez das aplicações, enquanto o terceiro grupo foca no efeito do número de mudanças no código excepcional (blocos catch) no nível de robustez das aplicações.

Concluímos com base nos dados coletados que há relação entre o número de mudanças no código que implementa o comportamento normal e o número de mudanças no código excepcional, quanto mais alterações se faz no código normal, o número de alterações no código excepcional aumenta. Concluímos também que há relação entre o número de mudanças no código normal e o número de fluxos excepcionais não tratados, fazendo cair a robustez dos softwares. Por fim, concluímos que há relação significante entre o número de mudanças no código excepcional e o número de fluxos excepcionais não tratados. Ou seja, com as características providas pelo mecanismo de tratamento de exceção da linguagem C#,

percebeu-se que as atividades de manutenção realizadas nas aplicações analisadas possuiram efeitos negativos na robustez das mesmas.

## 6.1 Publicações Relacionadas

Os resultados desta pesquisa nos permitiram obter uma publicação na trigésima sexta edição do ICSE (Proceedings of the 36th International Conference on Software Engineering (ICSE'14), Hyderabad, India, June 2014) com o artigo de título "***Trading Robustness for Maintainability: An Empirical Study of Evolving C# Programs***". Esse artigo apresenta um resumo desta pesquisa e receberá o prêmio "ACM Distinguished Paper award" no mesmo evento.

# REFERÊNCIAS


ALFREDO, J., MAGALHÃES, P. De, STAA, A. VON, DE, C. J. P. (2007). **Evaluating the Recovery Oriented Approach through the Systematic Development of Real Complex Applications.** Software Pract. Exper. 39, 3, 315 - 330, 2009.

ARAUJO, J. et al. **Handling Contract violations in Java Card using explicit exception channels.** 5th International Workshop on Exception Handling (WEH), pp. 34, 40, 9-9. 2012.

ARNOLD L. ROSENBERG. 2009. ***The Pillars of Computation Theory: State, Encoding, Nondeterminism***(1st ed.). Springer Publishing Company, Incorporated.

BALTER, R., LACOURTE, S., RIVEILL, M. **The Guide Language.** Comput. J., 37(6): pp. 519-530. 1994.

BARNETT, M., FAHNDRICH, M., VENTER, H. **Common Compiler Infrastructure**. 2010. Disponível em: <http://research.microsoft.com/en-us/projects/cci/>.

BERTRAND MEYER. **Object-Oriented Software Construction (2nd Ed.).** Prentice-Hall, Inc., Upper Saddle River, NJ, USA. 1997

BESSEY, A. *et al*. **A Few Billion Line of Code Later: Using Static Analysis to Find Bugs in the Real World**. Communications of the ACM, pages 66-75. 2010.

BOEHM, B. W., BROWN, J. R., LIPOW, M. **Quantitative evaluation of software quality.** In Proceedings of the 2nd international conference on Software engineering, IEEE Computer Society, Los Alamitos (CA), USA, 1976; 592-605.

CABRAL, B., MARQUES, P., SILVA, L. **RAIL: Code Instrumentation for .NET**. In Proc. of the 2005 ACM Symposium On Applied Computing (SAC'05), ACM Press, Santa Fé, New Mexico, USA. 2005.

CABRAL, B., MARQUES, P. **Exception Handling : A Field Study in Java and . NET.** *European Conference on Object-Oriented Programming.* Pages. 151–175. 2007.

CABRAL, B., SACRAMENTO, P., MARQUES, P. **Hidden truth behind .NETs exception handling today**. IET Software 1(6): 233-250 (2007)

CACHO, N. *et al*. **EJFlow: Taming Exceptional Control Flows in Aspect-Oriented Programming**. In Proceedings of the 7th international conference on Aspect-Oriented Software Development - AOSD., Pages 72-83. 2008.

CACHO, N. *et al*. **Exception Flows made Explicit: An Exploratory Study**. XXIII Simpósio Brasileiro de Engenharia de Software - SBES'09.

CANFORA, G., CIMITILE, A. **Software Maintenance.** 2000.

CASTOR FILHO, F., GARCIA, A., RUBIRA, C.: **Extracting Error Handling to Aspects: A Cookbook. ICMS'07 (2007)**

CHANG, B.-M. *et al*. **Interprocedural exception analysis for Java**. In SAC '01: Proceedings of the 2001 ACM symposium on Applied computing, pp. 620–625. ACM, New York, NY, USA.

COELHO, R., RASHID, A., GARCIA, A., FERRARI, F. and CACHO, N. (2008). **Assessing the Impact of Aspects on Exception Flows: An Exploratory Study**. European Conference on Object Oriented Programming (ECOOP 2008). 2008.

CRISTIAN, F. **A recovery mechanism for modular software.** In ICSE '79:Proceedings of the 4th international conference on Software engineering, pp. 42–50.A. IEEE Press. 1979.

CRISTIAN, F. **Exception Handling. Dependability of Resilient Computers**, pp. 68–97. 1989.

EVAIN, J. B. **Mono Cecil**. 2010. Disponível em: <http://www.mono-project.com/Cecil>.

F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira. **Exceptions and aspects: the devil is in the Details.** In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of Software engineering*, pages 152–162, New York, NY, USA, 2006. ACM.

FU, C., RYDER, B. G. **Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications**. In Proceedings of ICSE'07, pages 230-239, Minneapolis, USA, 2007.

GARCIA, A. *et al*. **A comparative study of exception handling mechanisms for building dependable object-oriented software.** The Journal of Systems and Software, 59(2):pp. 197–222. 2001.

GARCIA, I. and CACHO, N. **eFlowMining: An Exception-Flow Analysis Tool for .NET Applications.** *2011 Fifth Latin-American Symposium on Dependable Computing Workshops*, n. i, p. 1–8. 2011

GOODENOUGH, J. B. **Exception handling: Issues and a proposed notation**. Comm. of the ACM, 18(12):683–696. (1975).

GORBENKO, A., ROMANOVSKY, A., KHARCHENKO, V., MIKHAYLICHENKO, A. **Experimenting with exception propagation mechanisms in service-oriented architecture.** *Proceedings of the 4th international workshop on Exception handling - WEH '08*, p. 1–7. 2008.

GOSLING, J., JOY,B., AND STEELE, G. **The Java Language Specification.** Addison Wesley. Longman, Inc., Reading, MA. 1996.

ICAZA, M. **Mono Project**. 2001. Disponível em: < http:// www.mono-project.com/Start>.

IEEE (1990). IEEE Standard Glossary of Software Engineering Terminology. v. 121990.

JOHNSON, B., SONG, Y., MURPHY-HILL, E., BOWDIDGE, R. **Why Don ' t Software Developers Use Static Analysis Tools to Find Bugs ?** *ICSE*, 2013.

LAPRIE, J.-C., RANDELL, B. **Basic Concepts and Taxonomy of Dependable and Secure Computing.** IEEE Trans. Dependable Secur. Comput., 1(1):pp. 11–33. 2004. Fellow-Algirdas Avizienis and Senior Member-Carl Landwehr.

LEE, P. A., ANDERSON, T. **Fault Tolerance: Principles and Practice. Dependable computing and fault-tolerant systems**. Berlin; New York, 2nd edn.1990

MALAYERI, D., ALDRICH, J. **Practical Exception Specifications**. In Dony, C., Knudsen, J. L., Romanovsky, A. B. and Tripathi, A. (eds.), Advanced Topics in Exception Handling Techniques, vol. 4119 of Lecture Notes in Computer Science, pp. 200–220. Springer. 2006.

MARINESCU, C. **Are the classes that use exceptions defect prone?** *Proceedings of the 12th international workshop and the 7th annual ERCIM workshop on Principles on software evolution and software evolution - IWPSE-EVOL '11*, p. 56. 2011.

MATSUMOTO, YUKIO, AND K. ISHITUKA. **RUBY PROGRAMMING LANGUAGE**. 2002.

MILLER, R., TRIPATHI, A. **Issues with Exception Handling in Object-Oriented Systems.** Lecture Notes in Computer Science, 1241. 1997.

M. LIPPERT and C. V. LOPES. **A study on exception detection and handling using aspect-oriented programming.** In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 418–427, New York, NY, USA, 2000. ACM.

M. WILLIAMS. **Microsoft Visual C# .NET.** Microsoft Press, 2002.

PARNAS, D. L., WURGES, H. **Response to undesired events in software systems**. In ICSE '76: Proceedings of the 2nd international conference on Software engineering, pp. 437–446. IEEE Computer Society Press, Los Alamitos, CA, USA. 1976.

PFLEEGER, S.L., ATLEE, J.M. **Software Engineering: Theory and Practice (4th Edition)**. Prentice Hall. 2009.

R. DE LEMOS and A. B. ROMANOVSKY. **Exception Handling in the Software Lifecycle.** *International Journal of Computer Systems Science and Engineering*, 16(2):119–133, 2001.

R. MILLER, A. TRIPATHI. **Issues with exception handling in object-oriented systems.** In: ECOOP'97 proceedings, Lecture Notes in Computer Science, Vol. 1241, Mehmet Aksit and Satoshi Matsuoka editors, Springer-Verlag, 85-103, 1997.

RANDELL B., LEE, P., TRELEAVEN, P. C. **Reliability Issues in Cmputing System Design.** ACM Comput. Surv., 10(2):pp. 123-165. 1978

REIMER, D., SRINIVASAN, H. **Analyzing exception usage in large Java applications.** In Proceedings of ECOOP'2003 Workshop on Exception Handling in Object-Oriented Systems, pp. 10–18. 2003.

ROBILLARD, M. P., MURPHY, G. C. **Designing robust Java programs with exceptions**. In Proceedings of the 8th ACM SIGSOFT international Symposium on Foundations of Software Engineering: Twenty-First Century Applications. ACM Press, New York, 2000, 2-10.

ROBILLARD, M. P., MURPHY, G. C. **Static analysis to support the evolution of exception structure in object-oriented systems**. ACM Trans. Softw. Eng. Methodol., 12(2):pp. 191–221. 2003.

ROBINSON, S. *et al*. **Professional C# Third Edition.** Wiley Publishing, Inc., Indianalopis, Indiana, USA. 2004.

RUSTAN, K. SCHULTE, W. Leino, M. **Exception safety for C#.** In *SEFM 2004*—Second International Conference on Software Engineering and Formal Methods, pages 218–227. IEEE, September 2004.

K.O. DULAIGH, J.F. POWER, and P.J. CLARKE. **Measurement of exception handling code: An exploratory study.** In Exception Handling (WEH), 2012 5th International Workshop on, pages 55–61, 2012.

SACRAMENTO, P., CABRAL, B., MARQUES, P. **Unchecked Exceptions: Can the Programmer be Trusted to Document Exceptions?** In Second Edition of the International Conference on Innovative Views of .NET Technologies (IVNET'06).

SCHAEFER, C. F., BUNDY, G. N. **Static analysis of exception handling in Ada.** Softw. Pract. Exper., 23(10):pp. 1157–1174. 1993.

SHAH, H. CARSTEN, G., HARROLD, M. **Visualization of exception handling constructs to support program understanding**. In Proceedings of the 4th ACM symposium on Software visualization (SoftVis '08). ACM, New York, NY, USA, 19-28.

SHAH, H. CARSTEN, G., HARROLD, M. **Understanding Exception Handling : Viewpoints of Novices and Experts.** v. 36, n. 10, p. 1–12. 2010.

SINHA, S., ORSO, A., HARROLD, M. J. **Automated Support for Development, Maintenance, and Testing in the Presence of Implicit Control Flow**. In ICSE '04: Proceedings of the 26th International Conference on Software Engineering, pp. 336-345. IEEE Computer Society, Washington, DC, USA. 2004.

VAN ROSSUM, GUIDO. **Python language website**. *World Wide Web: http://www.python.org*. 2007.

WOHLIN, C., et al. **Experimentation in Software Engineering - An Introduction,** Kluwer Academic Publishers, Boston, MA. 2012.

YANG, M. X. **Discovering Faults in Idiom-Based Exception Handling**. ELEC876 : Paper Presentation. 2008.

YAU, S. AND COLLOFELLO, S. **Design Stability Measures for Software Maintenance.** Trans. on Softw. Engineering, 11(9), p. 849-856, 1985.

YEMINI, S., BERRY, D. **A Modular Verifiable Exception Handling Mechanism.** ACM Transactions on Programming Languages and Systems,7(2):pp. 214–243. 1985