Trading Robustness for Maintainability: An Empirical Study of Evolving C# Programs

Nélio Cacho,Thiago César, Thomas Filipe, Eliezio Soares, Arthur Cassio, Rafael Souza, Israel Garcia
Department of Informatics and Applied Mathematics, Federal University of Rio Grande do Norte, Natal, Brazil
neliocacho@dimap.ufrn.br, {lord.sena, thomasfdsdiniz, arthurecassio,rafael, israelbarbosa}@lcc.dimap.ufrn.com

Eiji Adachi Barbosa, Alessandro Garcia
Informatics Department, Pontifical Catholic University of Rio de Janeiro – PUC-Rio, Rio de Janeiro, Brazil
{ebarbosa,afgarcia}@inf.puc-rio.br

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 paper addressed 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 119 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.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Maintenance, and Enhancement–*corrections, restructuring, reverse engineering, reengineering.*

General Terms

Measurement, Reliability, Experimentation.

Keywords

Exception handling, robustness, maintainability

ICSE'14, May 31 – June 7, 2014, Hyderabad, India.

1. INTRODUCTION

Robust software systems [23] must provide their intended functions and operations in the presence of erroneous or unexpected conditions. Thus, a compulsory part of a robust software system is its exception handling [4, 37, 43]. Exceptions are abnormal events that occur at runtime to indicate that the system internal state is inconsistent and, therefore, recovery actions should be taken before the system continues its normal flow. An exception handling mechanism (EHM, for short) [16, 19, 32] is a model that allows software developers to detect and signal the occurrence of exceptions and also to structure recovery actions.

Experienced software developers explicitly recognize the importance of robust exception handling in evolvable software systems [37]. However, given the incremental nature of software development projects, exception handling is often explicitly treated and incorporated a posterior [26, 43]. There is a wide range of forces governing incremental treatment of exception handling in a software project [26]. In some cases, exceptions and appropriate recovery actions for existing functionalities can only be revealed after the core functionalities are fully known and implemented in a program [21]. In other cases, new requirements emerge afterwards; then, the related exceptions and handlers have to be identified, structured and realized on demand. Each software project may also impose particular constraints on when and how exception handling structure should evolve over time. As a consequence, the solution for exception handling implementation should enable modular and agile realization of software changes. Otherwise, software developers may be discouraged on maintaining exception handling in a program [37].

Nowadays, most mainstream programming languages provide built-in EHMs to support implementation of exception handling in software systems. Moreover, some programming languages provide built-in EHMs with extra facilities to avoid the introduction of faults in the exception handling code, which would decrease the program robustness. In particular, these mechanisms force developers to explicitly specify at methods signatures an exceptional interface, i.e., a list of exceptions that a method might raise during its execution. In this manner, these mechanisms can perform automatic reliability checks to verify if consumer methods implement appropriate handlers for the exceptions declared in the exceptional interface of provider methods. We call these mechanisms as *reliability-driven EHMs.* Typical examples of languages that provide reliability-driven EHMs are Java [20], Eiffel [27] and Guide[3]. Even though the facilities provided by reliability-driven EHMs are aimed at aiding developers to properly imple-

ment the exception handling of their programs, these facilities have well known adverse side-effects to software maintainability [6, 10, 25, 28, 35, 36]. Reliability-driven EHMs, therefore, favor software reliability in detriment to software maintainability.

In this manner, designers of programming languages, such as C# [44] and C++ [39], have increasingly argued for less rigid EHMs. We call these more flexible mechanisms as *maintenance-driven EHMs*. These mechanisms, in particular, do not enforce declarations of exceptional interfaces in methods signatures in order to make it easier to change exception handling structure according to project's evolving requirements. Despite the absence of exceptional interfaces, reliability checks are still supported at runtime. Therefore, their designers claim that these mechanisms allow developers to achieve a better trade-off between robustness and maintainability [37].

However, there is little empirical knowledge on how programs using such maintenance-driven facilities for exception handling evolve over time. Their impact and side effects on the reliability and maintainability in a software project is actually unknown. Fault-prone programming scenarios on changing both normal and exceptional codes need to be better understood when maintenance-driven exception handling is employed. Existing empirical studies focus on the investigation of patterns to handle exceptions in programming languages without built-in EHMs [1, 4] or reliability-driven facilities for exception handling supported, for instance, in Java and its dialects [10, 15, 25, 34, 35, 36].

This paper fills this gap by presenting an empirical study that assessed the relationship between program changes and C# programs robustness. We analyzed changes to both normal and exceptional code in 119versionsextracted from 16 software projects covering several domains. To conduct this study, change impact analysis and control flow analysis were performed on those versions of the analyzed projects. These data analyses allowed us to investigate how often certain recurring change scenarios decreased the software robustness of the evolving programs analyzed. The following particular findings were also derived:

- We confirmed that the use of maintenance-driven facilities in C# allows accommodating different evolution patterns for exception handling across the projects. We quantified which and to what extent specific change scenarios decrease software robustness.
- Indicators revealed that the effort – i.e. amount of code written – to include new exception handling code is consistently low across all C# projects. However, there were many types of change unexpectedly decreasing software robustness. This was the case, for instance, when: (i) adding handlers intended to remap and re-throw exceptions, and (ii) changing the handlers to catch more specific exception types. These changes focused on improving exception handling, but surprisingly and recurrently led to exception occurrences being uncaught. In particular, remapping of exceptions, for instance, is commonly referred as a good programming practice, but it was a common source of potential faults in C# programs.
- There was a high incidence of changes initially targeted at modifying normal code causing unexpected changes to exception handling. In other words, there is still a strong coupling between normal and exception code when using C# exception handling mechanism. The couplings are hard to detect by programmers as they cannot be observed syntactically (e.g. access to local variables). Thanks to this coupling and the lack of static reliability checks, programmers may often introduce faults in the exception handling code.

These findings seem to indicate that exception handling support still needs to be improved. Even though changes can be easily realized with C# programming mechanisms, the rate of possible exception handling faults was too high. And it was not just a matter of exception handling being ignored. The remainder of this paper is organized as follows. Section 2 describes basic concepts and the characteristics of exception handling mechanisms. Section 3 presents the experimental procedures adopted in this study. Sections 4 present results and their analyses. Section 5 presents the threats to validity. Section 6 discusses related work. Section 7 provides some concluding remarks.

2. EXCEPTION HANDLING

Exception handling mechanisms [16, 19, 32] are the most common models to cope with errors in the development of software systems. Their basic concepts and a classification are presented in the next subsections.

2.1 Basic Concepts

Although EHMs implementations vary from language to language, they are typically grounded on the *try/catch* constructs, as depicted by the following general structure:

```
try {S}
catch (E1 x) {H1}
catch (E2 x) {H2}
```

The *try* block delimits a set S, which is a set of statements that are protected from the occurrence of exceptions. The try block defines the *normal code*, and it is associated with a list of *catch* blocks. Each catch block defines a set H_n, which is a set of statements that implement the handling actions responsible for coping with an exception. The set H_n consists of the *exceptional code*, and it is commonly called the exception *handler*. Each catch block has an argument E_n, which is an exception type. These arguments are filters that define what types of exceptions each catch block can handle. When a catch block defines as argument an exception type E_1, it can handle exceptions of the E_1 type, and also exceptions that inherit from E_1. When a catch block handles an exception that is a subtype of its argument, it is said that this exceptions was handled by *subsumption*.

Handler Search. If an exception is thrown in the context of the set of protected statements S, the EHM performs at runtime the search for a proper handler. The search takes into account the list of catch blocks statically attached to the enclosed *try* block. The type of the exception thrown is compared with the exception types declared as arguments in each catch block. For the first matching pattern, the exception handler of that catch block is executed. If no matching pattern is found, the exception propagates up the call stack until a matching handler is found. If no handler is found in the whole call stack, the exception handler mechanism either propagates a general exception or the program is terminated. When no handler is found, it is said that the exception was *uncaught*.

Reliability-and Maintenance-Driven EHM. Amongst the different implementations of built-in EHM, we classify them in two main categories: *reliability-driven* and *maintenance-driven*. Reliability-driven EHMs promote software reliability by means of the explicit specification of exception handling constraints. These mechanisms enforce software developers to specify in each method interface a list of exceptions that might be raised during its execution and, therefore, should be handled by client methods. The list of exceptions that might be raised by a method is called its *exceptional interface*. In this manner, it is possible to perform

automatic reliability checks to verify if software modules adhere to the specified constraints. In Java, for instance, if a consumer method does not handle the exceptions specified in the exceptional interface of a provider method, nor specify them in its own exceptional interface, compilation errors occur. It is expected that such reliability-checks prevent software developers from introducing faults in the exception handling code.

Side Effects of Reliability-Driven EHMs. However, there are some adverse side effects of using exceptional interfaces. The main problem of supporting exceptional interfaces is related to its impact on software maintenance [28, 41]. If a new exception is added to the exceptional interface of a method at the bottom of the method call chain, the exceptional interfaces of all methods through which the new exception will be propagated also have to be updated. For programs with long method call chains, this is a time-consuming and error-prone task [28, 35,41], causing severe problems on the system architecture evolution. Furthermore, enforcing the declaration of exceptional interfaces in order to propagate exceptions leads to an artificial increase of the coupling between exception classes and intermediary methods that only propagate exceptions [6]. In some cases, this high coupling causes the inability to reuse normal behavior without changes to the normal and exceptional code [28].

The Omnipresence of Maintenance-driven EHMs. Maintenance-driven EHMs, on the other hand, promote software maintainability by not forcing developers to specify exception handling constraints. In this manner, developers are free of the burden of specifying exceptional interfaces. Most mainstream programming languages adopt maintenance-driven EHMs. In fact, amongst the top 10 most used programming languages[1], eight languages (C#, C++, Objective-C, PHP, (Visual) Basic, Python, JavaScript and Ruby) provide maintenance-driven EHMs, whereas one language (C) does not provide built-in EHM and one language (Java) provides reliability-driven EHM. Despite the large adoption of languages that provide built-in maintenance-driven EHMs, there is still a gap in the literature in terms of the impact that maintenance-driven EHMs have on software robustness. In this paper, we focus on the EHM provided by the C# programming language [44] to investigate such an impact of the maintenance-driven approach.

The built-in EHM provided by C# is a representative of the maintenance-driven category for two main reasons. First, C# supports automatic propagation of exceptions[16]. The automatic propagation of exceptions does not require inserting handlers to explicitly catch and rethrow exceptions along the call chain in order to ensure that these exceptions will make its way to the top of the call stack. Second, C# does not support the specification of exceptional interfaces in methods signatures, hence, methods can throw any exception without making this explicit in their signature. In this manner, developers can add new exceptions to existing methods without having to cope with the ripple effect of implementing the propagation of these exceptions along the whole call chain. This also avoids unnecessary tangling between normal and exceptional code [6] and facilitates the modification and reuse of the normal behavior of a system [28].

3. EXPERIMENTAL PROCEDURES

As described in the previous section, the maintenance-driven EHM focus on achieving superior maintainability of the normal and exceptional behavior of a system. However, there is limited empirical knowledge about the impact that the use of maintenance-driven EHM has on program robustness. Therefore, the goal of this empirical study is to investigate if and to what extent the benefits introduced by maintenance-driven EHM are counterbalanced by adverse side effects that decrease software robustness during software evolution.

The experimental procedures of this study comprised four stages: (i) in the first stage we defined the study hypotheses (Section 3.1); (ii) in the second stage, we selected the sample of subject programs (Section 3.2); (iii) then we defined the variables of the study and the suite of metrics to be computed from the source code and binaries of the subject programs (Section 3.3); and (iv) finally, we used the gathered metrics to perform statistical analysis using the statistical package SPSS (Section 4).

3.1 Hypotheses

This study relies on the analysis of the following hypotheses, which are set up as *null* hypotheses:

- **Hypothesis 1**: There is no significant relationship between the number of changes in the code responsible for implementing the normal behavior and the number of changes in the exceptional code (code within *catch* blocks).
- **Hypothesis *2***: There is no significant relationship between the number of changes in the code responsible for implementing the normal behavior and the level of robustness of a system.
- **Hypothesis 3:** There is no significant relationship between the number of changes in the exceptional code (code within *catch* blocks) and the level of robustness of a system.

The first hypothesis aims at investigating whether the number of changes in the exceptional code increases with the number of changes in the normal code. The second hypothesis investigates whether changes in the normal behavior of a program have side effects on the level of robustness of a system. The third hypothesis investigates whether changes in the exceptional behavior of a program have side effects on the level of robustness of a system.

3.2 Sample

We selected our sample of subject programs based on the sample of C# programs used on a previous study performed by Cabral and Marques [6]. We opted to select our subject programs from their study because their sample covers a wide spectrum of how maintenance-driven exception handling is typically used in real software development environments. We divided our subject programs in the same categories used by Cabral and Marques [6]: (i) **Libraries**: software libraries providing a specific application-domain API; (ii) **Applications running on server (Server-Apps)**: software applications running on server programs; (iii) **Servers**: server programs; and (iv) **Stand-alone applications**: desktop programs.

In our sample, we tried to include the maximum number of C# programs used in the study of Cabral and Marques. However, since in their study they only analyzed one version of each subject program, whereas in our study we were interested in analyzing programs during their evolution, we had to discard those programs that did not have more than one version available to download, nor had public version control systems available. For such cases, we replaced the discarded program with another program from the same category. Our final sample comprised a set of 16 open-source C# programs. Even though we had to perform some replacements in the original sample of Cabral and Marques study, our sample was also particularly diverse in the way exception handling is employed. For instance, we could find almost all the categories of exception handlers in terms of their structure [9], including nested exception handlers, masking handlers, context-

[1] http://www.tiobe.com/index.php/content/paperinfo/tpci/

dependent handlers and context-affecting handlers. We could also observe that the behavior of exception handlers significantly varied in terms of their purpose [6], ranging from error logging to application-specific recovery actions (e.g., rollback). Table 1 presents our final sample of subject programs.

Table 1: Subject Programs

	Name	Pairs	Average LOC
Libraries	Direct Show Lib	8	18,580.38
	Dot Net Zip (ZIP)	1	12,392.00
	Report Net	7	6,258.57
	SmartIRC4Net	3	6,117.67
	SubTotal	**19**	**223,298.00**
Server-Apps	Blog Engine (Admin)	4	4,656.75
	MVC Music Store	4	1,644.50
	PhotoRoom	4	1,048.50
	Sharp WebMail	11	2,597.55
	SubTotal	**23**	**57,972.00**
Servers	Neat Upload	4	8,635.00
	RnWood SMTP Server	3	2,638.00
	Super Socket Server (SocketEngine)	2	4,591.50
	Super Web Socket	5	1,048.80
	SubTotal	**14**	**45,608.00**
Stand-alone	Asc Generator	9	9,676.56
	CircuitDiagram (EComponents)	7	2,204.14
	NUnit (NUnitCore.Core)	11	6,862.78
	Sharp Developer (Main.Core)	21	5,289.35
	SubTotal	**48**	**270,070.00**
Total		**104**	**596,848.00**

Table 1 reports for each subject program its name, the number of consecutive version pairs analyzed, and the average number of lines of code. For the SmartIRC4Net library, for instance, we analyzed 3 pairs of versions: 0.2.0-0.3.0, 0.3.0-0.3.5 and 0.3.5-0.4.0. For each category, Table 1 also shows the total number of pairs and the total number of LOC analyzed. In the last row, it is shown the total number of pairs and the total number of LOC analyzed in this study.

3.3 Variables and Metrics

The selected variables are the metrics quantifying some characteristics of both normal and exceptional code. We chose different metrics to capture changes in the normal and the exceptional code during software evolution, and also metrics that quantify the robustness of a program. Therefore, the variables of interest of this study encompass a suite of metrics classified in three categories: size metrics, robustness metrics, and change metrics. The following subsections describe each suite of metrics and how they were computed.

3.3.1 Size Metrics

Size metrics capture the basic structure of the normal and the exception handling code of a software system. The size metrics used in this study are: (i) **NMod:** counts the number of modules (classes and interfaces) of each version of a system; (ii) **NMethod:** counts the number of methods of each version of a system; (iii) **NTry:** counts the number of *try* blocks of each version of a system; and (iv) **NCatch:** counts the number of *catch* blocks of each version of a system. We computed the size metrics using two different tools. NTry and NCatch were computed using the eFlowMining tool [17], whereas NMod and NMethod were computed using the SourceMonitor[38] tool.

3.3.2 Robustness Metrics

Robustness is the ability of a program to properly cope with errors during its execution [23]. If a handler is not defined or correctly bound to an exception, program robustness is decreased.In order to measure software robustness, we followed the typical approach adopted in the community in which exception flow information is used as an indicative of robustness [10, 15, 24, 35]. Exception flow is a path in a program call graph that links the method where the exception was raised to the method where the exception was handled. If there is no handler for a specific exception, the exception flow starts in the method where the exception was raised and finishes at the program entrance point. In the context of our analysis we used three metrics to support our analysis of software robustness, asthey are often used as an indicative of software robustness [10, 15, 24, 35]:

- **Specialized Flow**: counts the number of exception flows in which the raised exception is caught by a handler with the same exception type. Higher the number of specialized flows, higher the level of robustness is likely to be.
- **Subsumption Flow**: counts the number of exception flows in which the raised exception is caught by subsumption (Section 2.1). A high rate of handling by subsumption helps to determine the level of robustness as, for instance, exceptions are bound to general handlers and improper actions are being executed.
- **Uncaught Flow**: counts the number of exception flows that leave the bounds of the system without being handled. Uncaught exceptions terminate the execution of a program, hence, the higher the number of uncaught exception flows is, the lower the system's robustness is.

We employed the eFlowMining [17] tool to collect the robustness metrics. eFlowMining uses the Common Compiler Infrastructure framework [11] to analyze the binaries of a program and follows the approach proposed by [15] to perform an inter-procedural and intra-procedural dataflow analysis. The tool generates the exception flows for all exceptions, explicitly thrown by the application or implicitly thrown (e.g.,thrown by library methods). In this study we are assuming that only one exception is thrown at a time – the same assumption considered in [15]. As well as [6], we have ignored (i.e. not considered) exception flows created by exceptions that are not normally related to the program logic (Exe.:*System.MissingMethodException*). The full list of exceptions not considered in our analysis is detailed in [6].

3.3.3 *Change Metrics*

In order to quantify the changes in both normal and exceptional code, we collected a suite of typical change impact metrics [47]. Change impact metrics count at the program level the number of elements added, changed or removed between two versions of a given system. The elements considered in this study range from more coarse-grained elements, such as classes and methods, to more fine-grained elements, such as *try* and *catch* blocks. In our study, we considered exceptional code only the *catch* block, which handles an exception.

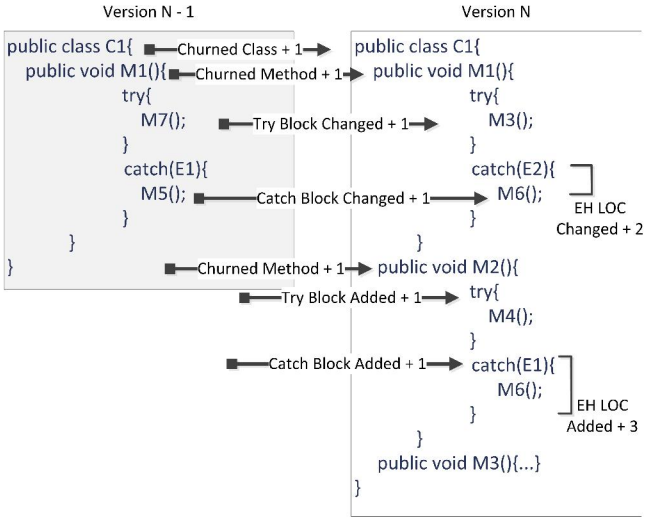

Figure 1: Collecting change impact measures: An example

The change metrics are computed based on the difference between the source code of a baseline version and the source code of a subsequent version. The definition of each change metric used in this study is presented next. Along with the definitions, there is also an example for each change metric. The examples are based on the code snippet presented in Figure 1. This figure shows two versions of a class, named *C1*, where the version N-1 (on the left) is the baseline version of *C1* and the version N (on the right) is the subsequent version of *C1*.

- **EHLocAdded, EHLocChanged** and **EHLocRemoved:** they count, respectively, the number of lines of exceptional code added, changed and removed between a pair of versions. In the example presented in Figure 1, three lines of exceptional code are added, two lines are changed and no line is removed. Hence, the value of the EHLocAdded metric is 3, the value of EHLocChanged metric is 2 and the value of EHLocRemoved is 0.
- **TryBlockAdded** and **TryBlockChanged:** they count, respectively, the number of try blocks added and changed between a pair of versions. In the example presented in Figure 1, one try block is added and another is changed. Notice that when lines of code are added to or removed from a try block, we considered as it has changed. Hence, the value of TryBlockAdded is 1 and the value of TryBlockChange is 1.
- **CatchBlockAdded** and **CatchBlockChanged:** they count respectively the number of catch blocks added and changed between a pair of versions. In the example presented in Figure 1, one catch block is added and another is changed. Hence, the value of CatchBlockAdded is 1 and the value of CatchBlockChange is 1.
- **ClassChurned:** It counts the number of classes that have a catch block added or changed between a pair of versions. In the example presented in Figure 1, catch blocks were added and changed in the same class, so the value of ClassChurned is 1.
- **MethodChurned:** It counts the number of methods that have a catch block added or changed between a pair of versions. In the example presented in Figure 1, catch blocks were added and changed in two different methods, so the value of MethodChurned is 2.
- **NormalChurnedLOC:** It counts the sum of the added and changed lines of normal code for a pair of versions. In the example presented in Figure 1, one line of normal code is changed (within the first try block) and six lines are added. Hence, the value of the NormalChurnedLOC is 7. We employed the Microsoft Line of Code Counter tool (LOCCounter)[29] to count NomalChurnedLOC. LOCCounter provides the ChurnedLOC metric, which counts the sum of the added and changed lines without distinguishing whether the code is normal or exceptional. For that reason, we subtract the absolute value of the ChurnedLOC metric provided by the tool from the sum of EHLocAdded and EHLocChanged, i.e., NormalChurnedLOC = ChurnedLOC – (EHLocAdded + EHLocChanged).

Our main goal during the change impact analysis was to gather deeper knowledge about the recurrent change scenarios observed and their impact on software robustness. Therefore, we had to not only compute the change metrics, but also describe each change scenario observed and assess its impact on software robustness. Since some of these information are inherently qualitative, we could not rely on any existing static analysis tool to automatically extract them from the source code, nor implement a tool of our own. In this manner, we had to perform a manual inspection in the source code in order to: textually describe the change scenarios and assess the impact of the observed changes in the software robustness. These tasks were performed by a group of six master students, a PhD student and two senior researchers. Each master student performed a manual inspection on the source code of the versions of a given subject program and simultaneously: (i) computed the change metrics, (ii) textually described the observed change scenarios and (iii) assessed the impact of the changes on software robustness by means of an exception flow analysis. The data produced by the master students were reviewed by the PhD student and the main researchers. Further clarifications, modification or improvements were performed by the master students when divergences were identified by the reviewers. Whenever necessary, the students performed an open discussion with the reviewers to resolve any conflict and reach a consensus on the data produced. If on one hand this manual inspection process did not allow us to scale our study to a larger sample, on the other hand it provided us with more accurate data and a better understanding of the change scenarios observed. Moreover, it also allowed us to identify and categorize change scenarios that improved or deteriorated the robustness of the subject programs. Next, each task is described in more details.

Change metrics computation and textual description. The change impact metrics were manually computed with the aid of the DiffMerge[14] tool. The DiffMerge tool performs a visual side-by-side comparison of two folders, showing which files are present in only one folder, as well as file pairs (two files with the same name but in different folders) that are identical or different. For file pairs with differences, the tool also graphically shows the

changes between the two files. The DiffMerge tool was used to compare the project folder of two subsequent versions of a given subject program. For each file pair with differences, the master students manually computed the change impact metrics by inspecting the differences pinpointed by the tool. For each change scenario analyzed, the students also textually described the observed changes occurred within try blocks and catch blocks. For instance, in one of the change scenarios analyzed, one of the students described it as follows: "It was observed a new IF statement added to the try block and a new method invocation added to the catch block". On a further step, the main researcher applied a coding technique to the textual descriptions of each change scenario in order to extract categories of change scenarios, divided in categories of changes in the normal code and categories of changes in the exceptional code. The result of this categorization is presented on Table 2.

Table 2: **Change scenarios and corresponding descriptions.**

	Scenarios	Description
Changes in exceptional code	Generic Handler added	A catch block which catches a generic exception is added
	Empty Generic Handler added	An empty catch blocks is added to handle a generic exception
	Generic Handler added to rethrow exception	A catch block which catches a generic exception is added to construct a new exception which is thrown
	Generic Handler removed	A catch block which catches a generic exception is removed
	Specialized Handler added	A catch block which catches a specific exception is added
	Empty Specialized Handler added	An empty catch blocks is added to handle a specific exception
	Specialized Handler added to rethrow exception	A specific handler is added to construct a new exception which is thrown
	Specialized Handler removed	A specialized handler is removed
	Changing the Exception Handling Policy	Changes how an specific exception type is handled
	Changing the catch block to use normal code	The catch block needs to change its code in order to use some references of the normal code.
	No changes to catch blocks	The try blocks changes but nothing changes to the catch block
Changes in the normal code	Try block added to existing method	A try block is added to an existing method declaration
	New method with try block added	A new method declaration with a try block is added
	New method invocation added	A new method invocation is added within a try bock
	Variables modified	A variable within a try block is modified
	Control flow modified	A control flow statement is modified within a try block
	Try block removed	A try block is removed from a method declaration
	Method with try block removed	A method declaration with a try block is removed from the project
	No changes to try block	The catch block changes but nothing changes in the try block

Exception flow analysis. Simultaneously to the computation of the change metrics, the students also performed an exception flow analysis for each change scenario analyzed. The exception flow analysis was performed with the aid of the eFlowMining tool. For each pair of subsequent versions analyzed, the eFlowMining computed the difference between the number of uncaught exception flows observed for each method. The difference was computed as:

$$\Delta Uncaught = Uncaught_{Subsequent} - Uncaught_{Baseline}$$

If the value of the difference was higher than zero, then it meant that the number of uncaught exception flows increased during the evolution; therefore, the software robustness decreased in the given change scenario. On the other hand, if the value of the difference was lower than zero, then it meant that the number of uncaught exception flows decreased during the evolution; therefore, the software robustness increased in the given change scenario. Finally, if the value of the difference was equal to zero, then it meant that no changes were observed in the number of uncaught exception flows; therefore, the change scenarios observed had no impact on software robustness.

In this manner, by combining the change metrics values, the categories of change scenarios extracted from the textual descriptions and the exception flow analysis, we were able to better understand which categories of change scenarios actually improved or deteriorated the robustness of the subject programs. In particular, the manual inspection process allowed us to systematically discover the scenarios that were more prone to generate uncaught exception flows.

Table 3: **Descriptive statistics for analyzed categories.**

		Application Categories				
		Libraries	Server-Apps	Servers	Stand-alone	Total
Size Metrics	**NMod**	7,959	702	929	4,446	14,036
	NMethod	14,048	4,838	5,369	33,602	57,855
	NTry	203	372	230	1,638	2,443
	NCatch	214	400	254	1,750	2,618
Change Metrics	**Class Churned**	22	70	33	122	247
	Method Churned	32	99	41	196	368
	Try Block Changed	20	17	5	43	85
	EHLocAdded + EHLocChanged	203	338	89	648	1,238
	Normal-Churned LOC	38,711	13,697	11,764	25,789	89,961

4. RESULTS AND ANALYSIS

This section reports our empirical findings and statistical tests of the hypotheses presented in Section 3.1. As far as statistical analyses are concerned, Spearman rank correlation analysis was applied to identify highly-correlated metrics. A Spearman value *rs* of +1 and -1 indicates high positive or high negative correlations. It is assumed there is no correlation when $0 < rs < 0.1$, low correlation when $0.1 \leq rs < 0.3$, median correlation when $0.3 \leq rs < 0.5$, and strong correlation when $0.5 \leq rs < 1$. Similar intervals also apply for negative correlations. The results of the correlation tests are presented in the next sections.

4.1 Co-changes to Normal and Exceptional Code

By analyzing the data presented in Table 3, we observed that changes of try (i.e. normal code) and catch (i.e. exceptional code) blocks were performed on a considerable number of methods and

classes. This observation is confirmed by the measures ClassChurned and MethodChurned. Library programs suffered fewer changes than the others. Moreover, changes to both normal and exceptional code were consistently spread throughout many modules in all the application categories. Results and analysis about the structure and evolution of exception handling for each category is presented in Section 4.3.

Table 4: **Classification of normal and exceptional change scenarios.**

Changes in the exceptional code	Changes in the normal code (try blocks only)							
	Try block added to existing method	New method added with try block	New method invocation added	Variables modified	Control flow modified	Try block removed	Method with try block removed	No changes to Try blocks
Generic Handler added	19.5%	64.6%	9.4%	-	3.6%	-	-	-
Empty Generic Handler added	24.4%	16.5%	3.8%	-	-	-	-	-
Generic Handler added to rethrow	34.1%	3.1%	3.8%	-	-	-	-	-
Generic Handler removed	-	-	-	-	3.6%	87.0%	96.8%	7.7%
Specialized Handler added	12.2%	10.2%	3.8%	-	-	-	-	3.8%
Empty Specialized Handler added	2.4%	0.8%	-	-	-	-	-	3.8%
Specialized Handler added to rethrow	7.3%	4.7%	1.9%	-	-	-	-	-
Specialized Handler removed	-	-	-	-	-	13.0%	3.2%	-
Changing the Exception Handling Policy	-	-	11.3%	8.3%	7.1%	-	-	53.8%
Changing the catch block to use normal code	-	-	15.1%	50.0%	3.6%	-	-	30.8%
No changes to handlers	-	-	50.9%	41.7%	82.1%	-	-	-
Percentage	11.0%	34.0%	14.5%	3.2%	7.5%	6.1%	16.8%	7.0%

We investigated whether the number of changes in the exceptional code increased with the number of changes in the normal code. To do so, we manually inspected all the 402 change scenarios encompassing addition, modification and deletion of try-catch blocks. We observed and distinguished: (i) recurring co-changes between the normal code blocks and exceptional blocks, and (ii) independent changes made to either of them. Each change scenario was classified according to the action taken on its normal code (try block only) and exceptional code (catch block), following the classification presented in Section 3.3.3. Table 4 summarizes the frequency of change scenarios involving co-changes between the normal (columns) and exceptional (rows) code. The frequency of independent changes made to the normal code is captured in the row "No changes to handlers"; whereas independent changes to the exceptional code is captured in the last column "No changes to try blocks". The last row labeled "Percentage" shows the distribution of the frequency for normal change scenarios.

Poor Handlers for New Try Blocks. An analysis of the last row in Table 4 reveals that additions of try blocks represent the most common changes to normal behavior. They are either added to existing methods (first column) in 11% of the cases, or together with the addition of new methods (second column) in 34% of the cases. However, for those scenarios involving new try blocks, developers usually attached poor exception handlers. For instance, generic handlers were introduced in: (i) 64.6% of the cases of new methods, and (ii) 19.5% of the cases of new try blocks for existing methods. The addition of try blocks to existing methods is also performed together with other forms of handlers: (i) addition of generic handlers to rethrow exceptions (34.1% of the total), and (ii) addition of empty generic handlers (24.4% of the total). This result is opposite to the claim that the utilization of poor handlers is a consequence of using features of reliability-driven EHM, such as checked exceptions[34, 41].

Table 5: **Spearman rank correlation between changes in the normal code and exceptional code.**

		Application Categories				
		Libraries	Server-Apps	Servers	Stand-alone	Total
NormalChurned 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	0.071	**0.436****	**0.203***
	Spec. Flow	-0.344	-0.221	0.480	0.176	0.041
	Sub Sumpt. Flow	-0.063	**0.621****	0.296	**0.304***	**0.262****
	Uncaught Flow	-0.054	0.315	**0.578***	**0.656****	**0.437****
Catch Block Added	Spec. Flow	0.225	-0.247	-0.134	0.030	-0.048
	Sub Sumpt. Flow	**0.575***	**0.685****	**0.621***	**0.413****	**0.548****
	Uncaught Flow	**0.482***	0.292	**0.641***	**0.419****	**0.405****

** and **: Correlations at the 0.05 level are marked with *, while those at the 0.01 level are marked with **.*

Later Changes to Catch Blocks. Another common change scenario in the normal behavior occurs when a new method invocation is added within a try bock (14.5% of the total). For that scenario, we found that 50.9% of the cases were due to independent changes to try blocks (*No changes to handlers* row). Independent changes to try blocks also occurred often when variables were changed (41.7%) and control flow was modified (82.1%). Next subsection identifies how often robustness was decreased in these circumstances involving normal code changes. When changes were also made to exception handlers, they were of diverse nature, varying from changes to the exception handling policy (11.3%), use of variables in the normal code (15.1%) to different types of handlers added. We observed that the majority of these changes to handlers occurred in late program versions when the normal behavior was already stable.

Independent Changes to Handlers. Finally, developers independently changed only the handler – i.e. without changing the try block – in 7% of the scenarios. For this scenario, changes to the exception handling policy make up 53.8% of exceptional changes;

changes the catch block to use normal code represents 30.8%; removal of generic handlers makes up 7.7%; and additions of specialized handlers represent 7.6%.We also observed that independent changes to handlers were always made in late versions. Developers clearly tend to improve exception handling over time. The small proportion (7%) of independent changes may also indicate developers still neglect to fully maintain the exceptional code.

Test of Hypothesis 1. The Table 5 presents the values of the Spearman´s rank correlation coefficient. The last column shows the overall result for the correlation coefficients considering all categories of the subject programs. The result shows that NormalChurnedLOC is correlated to all change impact metrics (EHLocAdded, EHLocChanged and EHLocRemoved). In other words, result shows that when the normal code is changed, the exceptional code is also changed. In this manner, we can reject Hypothesis 1.

Table 6: **Normal code changes increasing uncaught exceptions.**

	New method invocation added	Variables modified	Control flow modified	Total
Reduced Uncaught	-	-	50%	25%
Increased Uncaught	100%	100%	50%	75%

4.2 Normal Code Changes and Robustness

Previous section reported that there is a correlation between normal code changes and exceptional code changes. This section discusses the relation between changes to the normal code and the program robustness. We assessed if normal code changes lead to an increase (or decrease) in the number of: uncaught exceptions, specialized exception handling or exception handling by subsumption. Table 5 presents the correlations between the metrics quantifying changes in the normal code (NormalChurnedLOC) and the metrics computing system robustness (Spec. Flow, Subsumpt. Flow and Uncaught Flow).

Normal Code Changes often Decrease Robustness. The analysis of Table 5 reveals that there is often strong or medium correlation between normal code changes and most robustness measures. The level of correlation varies according to: (i) the application category, and (ii) the type of change to normal behavior. Strongest correlations were most often observed in C# programs falling in the Stand-Alone category. In this category, it was also observed strong correlation between changes to normal code (NormalChurnedLOC) and uncaught exceptions. This type of correlation was also strong for the Servers category. As far as the Server-Apps category is concerned, there is no correlation between additions or changes of normal code with the number of unhandled exceptions or specialized exception handlers. On the other hand, there is a strong correlation between the former with the number of exceptions handled by subsumption. This result suggests that features added to the programs of the Server-app category were protected by generic try/catch blocks.

Types of Changes Leading to Uncaught Exceptions. Uncaught exceptions represent the most effective indicator of robustness decrease. Independent changes to normal code often occurred when variables were changed and the control flow was modified. We observed that they often lead to uncaught exceptions. Table 6 lists the normal change scenarios that most often generated uncaught exceptions: (i) 100% of the additions of method invocations, and (ii) 100% of the modifications to variables. This result was consistent through all the program categories. This finding also reveals that the use of maintenance-driven EHM of C# is error prone in these circumstances. Many dependencies between normal and exceptional blocks are hard to identify even by experienced programmers. Without the automatic reliability check performed based on exceptional interfaces,developers could not determine if a new exception would be raised in the changed try block and, hence, it often became uncaught. There were cases that the addition of a single line in the normal code increased the number of uncaught flows, i.e., decreased the robustness of the program. This scenario was responsible for generating 15.8% (Table 7) of the uncaught exceptions, the third most common.

Table 7: **The percentage of change scenarios leading to uncaught exceptions.**

	Application Categories				
	Libraries	Server-Apps	Servers	Stand-alone	Percentage of Uncaught Exceptions
Generic Handler added to rethrow exception	2.6%	2.6%	23.7%	-	28.9%
Specialized Handler added	2.6%	-	-	21.1%	23.7%
No changes to catch blocks	5.3%	5.3%	-	5.3%	15.8%
Specialized Handler added to rethrow	-	2.6%	7.9%	2.6%	13.2%
Changing the Exception Handling Policy.	-	-	5.3%	2.6%	7.9%
Changing the Catch Block code to use normal code	7.9%	-	-	-	7.9%
Generic Handler removed	-	-	-	2.6%	2.6%

Test of Hypothesis 2. The last column of Table 5 shows the overall result of the correlations when considering all applications, regardless of program category. The result shows that 2 of the 3 correlations are significant. This means that the data pinpoints to a trend: when the normal code changes, this change is likely to adversely impact the robustness of C# programs. In particular, when additions and modifications in the normal code are not made together with proper exception handler, the number of uncaught flows often increases. Thus, we reject Hypothesis 2.

4.3 Exceptional Code Changes and Robustness

This section analyzes if changes to the exceptional code have any positive or negative relation to the program robustness. As those changes are focused on enhancing error handling, they should significantly improve program robustness. For instance, with the addition of new handlers it is expected to be observed an increase in the number of handled exceptions. In particular, given the maintainability-driven nature of the EHM in C#, improvements in the exception handling are expected to occur often as the program evolves. Furthermore, they should, at least, reduce the number of uncaught exceptions.

Diverse Patterns of Exception Handling Evolution. In order to test the third hypothesis, we analyzed first how exception handlers evolved over time in the subject programs. Figure 2presents the results by showing the NCatch measures (Section 3.3.1), i.e. the number of catch blocks (Y-axis) for the versions (X-axis) of the C# projects. Each graph presents the data for the programs in each application category. There is a significant NCatch variation in all the software projects. This result reinforces that C# programmers exploit the maintenance flexibility yielded by the EHM in different ways.

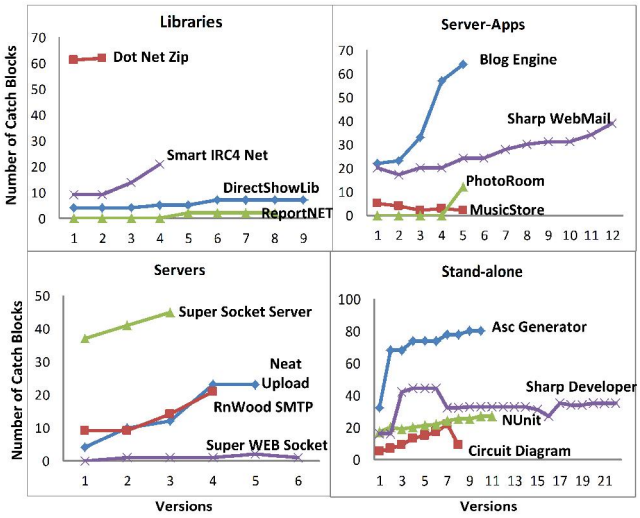

Figure 2: Number of catch blocks throughout the versions.

The curves in Figure 2 also reveal that the patterns of handler changes are different in each program. First, NCatch measures slightly increased in the Libraries category, except in one case (SmartIRC4Net), where the number of handlers doubled over time. The handlers in Libraries mostly follow a uniform and stable pattern since the first version of a program. They catch exceptions, partially treat them (e.g. performing clean-up actions), and remap them to the client applications. Second, there were moderate to significant increases in the number of handlers in most of the Server and Server-app programs. There was a gradual decrease in the MVC Music Store project and a punctual drop in the last version of Super Web Socket. Third, the widest variation was observed in the group of Stand-alone applications, including projects where there were significant drops in the number of handlers.

Leveraging Maintenance-Driven EHM. In addition to the quantitative analysis of catch blocks, we also analyzed the nature of the changes in the exceptional behavior. We observed that the specialized exception flows tend to increase through the program versions (Table 4). Previous studies [28, 34,37] of reliability-driven EHM, supported by Java and its dialects, have shown that programmers follow an opposite behavior. Programmers often introduce generic exception handlers in the initial versions of their programs thanks to their time constraints. They also face significant difficulties later to create new exception types and introduce the corresponding specialized blocks in later versions [28, 34,37]. As a result, the number of application-specific exception types and handlers tends to remain stable in evolving programs. The reason is the reliability checks based on interfaces of checked exceptions.

Certain Exceptional Code Changes Decrease Robustness. Even when the number of specialized handlers increases over time, the program robustness would be decreased if exceptions were not properly bound to handlers. Then, we analyzed the correlation between changes to exceptional behavior and the number of: (i) exceptions caught by subsumption and, more importantly, (ii) uncaught exception flows. As far as the first case is concerned, Table 5 shows that there is significant correlation between the addition of catch blocks and the number of flows handled by subsumption. There are two harmful change scenarios that exemplify this phenomenon. The first is when new code is added to a program and this code originates an exceptional flow inadvertently captured by a generic catch block. The second scenario occurs when a generic handler is added to a given method that threw exceptions and did not originally handle these exceptions. In this case, it is not a new exceptional flow that emerges in the code, but an old flow that starts to be handled by a new generic handler.

Table 8: **The percentage of change scenarios increasing or decreasing uncaught flows.**

	Increase Uncaught	Decrease Uncaught
Generic Handler added	-	100%
Empty Generic Handler added	-	100%
Generic Handler added to rethrow Exception	100%	-
Generic Handler removed	7.1%	92.9%
Specialized Handler added	90%	10%
Changing the Exception Handling Policy	75%	25%
Changing the catch block to use normal code	100%	-
No changes to catch blocks	75%	25%

Uncaught Exceptions. We observed that certain types of change unexpectedly led to new uncaught exception flows. Table 5 shows that in three of the four subject programs categories, there is a significant correlation between the number of added catch blocks and the number of uncaught exceptions. That is, the addition of catch blocks seemed to increase the number of uncaught flows. In particular, this was the case when adding generic or specific handlers intended to remap and re-throw exceptions. These cases represented 28.9% and 13.2%, respectively, of the uncaught exceptions (last column of Table 7). When only the tally of change scenarios falling in each of these types are considered, 100% and 92.9% of them increased the number of uncaught exceptions (2nd column of Table 8). Uncaught exceptions were also often caused by changes to the handlers to catch more specific exception types. They represented 23.7% of uncaught flows (last column of Table 7). Surprisingly, all these types of changes were (apparently) focused on improving exception handling. However, they recurrently led to exception occurrences being uncaught. In particular, remapping of exceptions, for instance, is commonly referred as a good programming practice, but a common source of bugs in C# programs.

Test of Hypothesis 3. Finally, the last line of Table 5 shows the overall result of the correlations when considering all applications, regardless of category. The results show that 2 of the 3 correlations are significant. In other words, when changing the exceptional code, such changes may be reflected in the robustness of the system. Specifically, when addition and modification in the exceptional code are not properly performed, there is an increase in the number of uncaught flows. Therefore, we reject Hypothesis 3. This finding contradicts our initial expectation. It also indicates that even experienced programmers might end up unconsciously trading robustness (the main purpose for using an EHM) for main-

tainability. As represented by Table 7, the change scenarios that added exception handlers were amongst the main causes of the observed increase in the number of uncaught exception flows. In fact, the addition of handlers was related with more than 74% of the observed uncaught exception flows. There were some cases that an exception was raised from within the handler due to a method invocation and was not further handled. Initially, this was an unexpected scenario, but it actually occurred very often: 23.7% of the uncaught exception flows were related with this sort of change scenario.

5. THREATS TO VALIDITY

This section discusses threats to validity that can affect the results reported in this paper.

Internal Validity. Threats to internal validity are mainly concerned with unknown factors that may have had an influence on the experimental results [45]. To reduce this threat, we have selected a set of subject programs whose developers had no knowledge that this study was being performed for them to artificially modify their coding practices. Additionally, our results can be influenced by the performance, in terms of precisions and recall, of the used tools. We tried to limit the number of false positive through a manual validation. The use of this validation mitigates threats to internal validity, but does not completely remove them as a certain amount of imprecision cannot be avoided by the tools that generally deal with undecidable problems. Likewise, the change metrics collected manually were validated by one Ph.D. student who was not aware of the experimental goal.

Construct Validity. Threats to construct validity concern the relationship between the concepts and theories behind the experiment and what is measured and affected [45]. To reduce these threats, we use metrics that are widely employed to quantify the extent of changes over the software evolution [18, 30] and to measure the software robustness [10, 15, 35].

External Validity. External validity issues may arise from the fact that all the data is collected from 16 software systems that might not be representative of the industrial practice. However, the heterogeneity of these systems helps to reduce this risk. They are implemented in C#, which is one of the representative languages in the state of object-oriented practice. Most of the applications are widely-used and extensively evaluated in previous research [6]. To conclude, the characteristics of the selected systems, when contrasted with the state of practice, represent a first step towards the generalization of the achieved results.

Reliability Validity. This threat concerns the possibility of replicating this study. The source code of the subject programs is publicly available at [40]. The way our data was collected is described in detail in Section 3.3 and at [40]. Moreover, both the eFlowMining tool and SourceMonitor tool are available [40] to obtain the same data. Hence, all the details about this study are available elsewhere [40] to enable other researchers to control it.

6. RELATED WORK

Cabral and Marques [6] analyzed the exception handling code of 32 different open-source software systems from different domains and for both Java and .NET platforms. They performed their analyses in only one version of their subject programs, whereas we analyzed our subject programs during their evolution. Coelho *et al.* [10] assessed the interplay of exception handling and software robustness in the context of aspect-oriented programs. The authors have assessed the robustness of three medium sized programs implemented in AspectJ. Similarly to our approach, the authors assessed the subject programs during their evolution and used the metric *Uncaught Exception Flows*. Their study showed that the number of uncaught exception flows increased during programs' evolution. Our studies have some major differences, since Coelho *et al.* solely investigated robustness during software evolution and did not explore the relationship between maintenance tasks and software robustness. In this sense, our work complements theirs.

Marinescu [25] analyzed the defect proneness of classes that either catch or throw exceptions in the context of three releases of Eclipse. Similarly, Sawadpong*et al.* [36] studied six major releases of Eclipse and compared the defect densities of exception handling code and normal code. Their findings show that the defect density of exceptional code for the subject programs is considerably higher than the overall defect density. Our studies are similar in the sense that we both investigate the impact of exception handling on software robustness. However, our study is more precise in the sense that it describes the recurring change scenarios where possible exception handling faults were introduced, whereas their study only shows that exception handling code is more fault-proneness than normal code. Moreover, our studies also differ in: (i) the metric used to measure robustness: we used the metric of *Uncaught Exception Flows*, while they used the metric *Defect Density*; and (ii) the exception handling mechanisms provided by the programming languages in which the subject programs were implemented: their subjects were implemented in a reliability-driven exception handling mechanism, provided by Java, whereas ours were implemented in a maintenance-driven exception handling mechanism, provided by C#.

7. CONCLUSION

Maintenance-driven EHM are becoming increasingly popular in programming languages, such as C# (Section 2.2). Improving program robustness is the key motivation for using an EHM. However, there is little empirical knowledge about the impact of maintenance-driven EHM on the evolution of software systems. Our study suggests that C# programmers often unconsciously traded robustness for maintainability in a wide range of program categories. The programmers evolved their programs following different change patterns for the exception handling code. Moreover, the EHM of C# allowed flexible realization of changes.

However, the use of EHM in C# was very fragile and often led to robustness decrease. Programmers often introduced bugs when performing subtle changes in try blocks. A high number of uncaught exception flows were also introduced when the catch blocks were changed. These findings seem to indicate that there is still much room for improving built-in EHM in programming languages. Our ongoing work encompasses the implementation of the EFlow model [2, 7,8] for C# programming language. EFlow aims to improve the simultaneous satisfaction of software maintainability and reliability by using the notion of explicit exception channels, which support modular representation of global exceptional-behaviour properties.

8. ACKNOWLEDGMENTS

This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES), grants 573964/2008-4(CNPq) and APQ-1037-1.03/08 (FACEPE).

9. REFERENCES

[1] Alfredo, J., et al.Evaluating the recovery-oriented approach through the systematic development of real complex applications. Softw.Pract.Exper. 39, 3 (March 2009), 315-330.

[2] 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 June 2012.

[3] Balter, R., Lacourte, S. and Riveill, M. The Guide Language.Comput. J.,37(6):pp. 519–530. 1994.

[4] Bruntink, M., van Deursen, A., and Tourwé, T. Discovering faults in idiom-based exception handling. In Proceedings of the 28th international conference on Software engineering (ICSE '06). ACM, New York, NY, USA, 242-251.

[5] Bundy, G. N. and Mularz, D. E. Error-Prone Exception Handling in Large Ada Systems. In Ada-Europe '93: Proceedings of the 12th Ada-Europe International Conference, pp. 153–170. Springer-Verlag, London, UK. 1993.

[6] Cabral, B. and Marques, P. Exception handling: a field study in Java and .NET. In Proceedings of the 21st European conference on Object-Oriented Programming (ECOOP'07), Erik Ernst (Ed.). Springer-Verlag, Berlin, Heidelberg, 151-175.

[7] 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 '08).ACM, New York, NY, USA, 72-83.

[8] Cacho, N., Cottenier, T. and Garcia, A. Improving robustness of evolving exceptional behaviour in executable models. In Proceedings of the 4th international workshop on Exception handling (WEH '08). ACM, New York, NY, USA, 39-46.

[9]Castor Filho, F., Garcia, A., Rubira, C.: Extracting Error Handling to Aspects: A Cookbook. In: ICSM'07 (2007)

[10] Coelho, R., et al.Assessing the Impact of Aspects on Exception Flows: An Exploratory Study. In *Proceedings of the 22nd European conference on Object-Oriented Programming* (ECOOP '08), Jan Vitek (Ed.). Springer-Verlag, Berlin, Heidelberg, 207-234.

[11] Common Compiler Infrastructure. http://research.microsoft.com/en-us/projects/cci/

[12] Cristian, F. Exception Handling. Dependability of Resilient Computers, pp. 68–97. 1989.

[13] 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.

[14] DiffMerge -http://www.sourcegear.com/diffmerge/ (01/09/2013)

[15] Fu, C. and Ryder, B. Exception-Chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. In Proceedings of the 29th international conference on Software Engineering (ICSE '07).IEEE Computer Society, Washington, DC, USA, 230-239.

[16] Garcia, A. F., Rubira, C. M. F., Romanovsky, A. and Xu, J**.** *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.

[17] Garcia, I. andCacho, N., "eFlowMining: An Exception-Flow Analysis Tool for .NET Applications," *Fifth Latin-American Symposium onDependable Computing*, vol., no., pp.1,8, 25-29 April 2011.

[18] Giger, E.,Pinzger, M. and Gall, H. Comparing fine-grained source code changes and code churn for bug prediction. In Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11).ACM, New York, NY, USA, 83-92.

[19] Goodenough, J. B. Exception handling: issues and a proposed notation. Commun. ACM, 18(12):pp. 683–696. 1975.

[20] Gosling, J., Joy, B., and Steele, G. 1996. The Java Language Specification. Addison Wesley.Longman, Inc., Reading, MA.

[21] Kienzle, J. On exceptions and the software development life cycle.In Proceedings of the 4th international workshop on Exception handling (WEH '08).ACM, New York, NY, USA, 32-38.

[22] Laprie, J.-C. andRandell, B. Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Trans. Dependable Secur.Comput., 1(1):pp. 11–33. 2004.

[23] Lee, P. A. and Anderson, T. Fault Tolerance: Principles and Practice.Dependable computing and fault-tolerant systems. Berlin ; New York, 2nd edn.1990.

[24] Maji, A., et al.An empirical study of the robustness of Inter-component Communication in Android. In Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) (DSN '12). IEEE Computer Society, Washington, DC, USA, 1-12.

[25] Marinescu, C. Are the classes that use exceptions defect prone?.12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution (IWPSE-EVOL '11), In Proceedings of the. 2011.

[26] Maxion, R.A.; Olszewski, R.T., "Eliminating exception handling errors with dependability cases: a comparative, empirical study," Software Engineering, IEEE Transactions on , vol.26, no.9, pp.888,906, Sep 2000.

[27] Meyer, B. 1997. Object-Oriented Software Construction (2nd Ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[28] Miller, R. andTripathi, A. 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.

[29] Microsoft Line of Code (LOC) Counterhttp://archive.msdn.microsoft.com/LOCCounter(01/09/2013)

[30] Nagappan, N. and Ball, T. Use of relative code churn measures to predict system defect density. In Proceedings of the 27th international conference on Software engineering (ICSE '05). ACM, New York, NY, USA, 284-292.

[31] Parnas, D. L. The influence of software structure on reliability.In Proceedings of the international conference on Reliable software, pp. 358–362.ACM Press, New York, NY, USA. 1975.

[32]Parnas, D. L. and 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.

[33] Randell, B., Lee, P. and Treleaven, P. C. Reliability Issues in Computing System Design. ACM Comput.Surv., 10(2):pp. 123–165. 1978.

[34] Reimer, D. and 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.

[35] Robillard, M. and Murphy, G. Static analysis to support the evolution of exception structure in object-oriented systems. ACM Trans. Softw.Eng. Methodol. 12, 2 .

[36] Sawadpong, P., Allen, E.B. and Williams, B.J., Exception Handling Defects: An Empirical Study, High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on, 2012

[37] Shah, H.B.; Gorg, C.; Harrold, M.J., Understanding Exception Handling: Viewpoints of Novices and Experts, Software Engineering, IEEE Transactions on , vol.36, no.2, pp.150,161, March-April 2010

[38]SourceMonitor-http://www.campwoodsw.com/sourcemonitor.html(01/09/2013)

[39] Stroustrup, B. The design and evolution of C++.ACM Press/Addison-Wesley Publishing Co., New York, NY, USA. 1994.

[40] Trading Robustness for Maintainability - Web site: http://www.dimap.ufrn.br/~neliocacho/csharpstudy.html

[41] van Dooren, M. and Steegmans, E. Combining the robustness of checked exceptions with the flexibility of unchecked exceptions using anchored exception declarations. In OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, pp. 455–471. ACM Press, New York, NY, USA. 2005.

[42] Weimer, W. and Necula, G. C. "Finding and preventing run-time error handling mistakes." Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04), 19th annual ACM SIGPLAN Conference on, 2004.

[43] Weimer, W. and Necula, G. 2008. Exceptional situations and program reliability.ACM Trans. Program. Lang. Syst. 30, 2, Article 8 (March 2008)

[44] Williams, M. Microsoft Visual C# .NET. Microsoft Press, 2002.

[45] Wohlin, C., et al. Experimentation in Software Engineering – An Introduction, Kluwer Academic Publishers, Boston, MA.

[46] Yang, H. and Ward, M. Successful Evolution of Software Systems. Artech House, Inc., Norwood, MA, USA. 2003.

[47] Yau, S. and Collofello, S. Design Stability Measures for Software Maintenance. Trans. on Softw.Engineering, 11(9), p. 849-856, 1985.