# Preventing Erosion in Exception Handling Design Using Static-Architecture Conformance Checking

Juarez L.M. Filho[1], Lincoln Rocha[1(✉)], Rossana Andrade[1], and Ricardo Britto[2]

[1] Group of Computer Networks, Software Engineering, and Systems, Federal University of Ceará, Fortaleza, CE, Brazil
{juarezmeneses,lincoln,rossana}@great.ufc.br
[2] Blekinge Institute of Technology, 37179 Karlskrona, Sweden
ricardo.britto@bth.se

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

**Keywords:** Exception handling design · Exception handling erosion · Architecture conformance checking

## 1 Introduction

Exception handling is a well-known error recovery approach to improve software robustness. An exception is an event or abnormal situation detected at runtime


A. Lopes and R. de Lemos (Eds.): ECSA 2017, LNCS 10475, pp. 67–83, 2017.
DOI: 10.1007/978-3-319-65831-5_5

that disrupts the normal control flow of a program [10]. When this happens, the exception-handling mechanism deviates the normal control flow to the abnormal (exceptional) control flow to handle the exceptional situation. The exception handling mechanism structures the exceptional control flow by using proper constructs to indicate in the source code where exceptions can be raised and handled. Most of mainstream programming languages (e.g., Java, C++, and C#) provide built-in facilities to implement exception handling features [4].

Architecture erosion is a phenomenon that occurs when the implemented (concrete) architecture of a software system diverges from its intended (planned) architecture [20]. In fact, it is a side effect of a non-controlled software evolution process in which changes made in the source code lead to architecture design rules violations [12]. To cope with this problem, architecture conformance checking provides means to control the architectural erosion by automatically monitoring the compliance between the implemented architecture and the intended one [17]. This systematic control aims at guaranteeing that the architect's design decisions - and the quality attributes derived from it - are properly reflected in the system implementation [5]. Additionally, once the architecture conformance checking requires a design specification as input (e.g., architectural elements declaration, mapping between architectural and implementation elements, and design constrains), the knowledge about the architectural design decisions becomes better documented and easier to share.

Despite its importance, studies have reported that exception handling is commonly neglected by developers and is the least understood, documented, and tested part of a software system [6,13,19]. Additionally, to promote software maintainability, modern programming languages (e.g., C#, Ruby, and Python) have incorporated new maintenance-driven flexibilities in its built-in exception handling mechanism [3]. This make changes in the source code more agile by not forcing developers to follow the exception handling constraints (e.g., declare in each method interface a list of exceptions that might be signaled and, therefore, should be handled by caller methods). Nevertheless, this flexibility allows developers to postpone the implementation of some parts of exception handling, taking the risk of forgetting to return and implement the remaining exception handling features. All these issues may lead developers to violate the software architect's intention concerning the exception handling design during the development, maintenance and evolution phases. Such kind of violations are dangerous because it can lead to: (i) the exception handling mechanism to behave erroneously or improperly at runtime; and (ii) exception handling software faults [7]. We call this problem *exception handling erosion* (EHE).

The state of the art conformance checking solutions [5,8,11,15,22] do not provide a proper support for architecture conformance checking of exception handling design. Even a most recent solution [1], devoted to conformance checking of exception handling design, do not provide a full-fledged support to deal with the EHE problem. Therefore, we address this gap in this paper through answering the following research questions: **RQ1** - *How can the EHE problem be addressed in a systematic way*? **RQ1.1** - *How effective is the proposed approach*

*in the identification of existing EHE problems*? and **RQ1.2** - *How useful is the proposed approach to identify EHE causes in the source code*?

To answer RQ1, we propose ArCatch, an architecture conformance checking solution that provides: (i) a declarative language (ArCatch.Rules) for expressing design constraints regarding exception handling; and (ii) a design rule checker (ArCatch.Checker) to automatically verify the exception handling conformance. The ArCatch is implemented as a Java internal DSL (Domain-Specific Language), easing its incorporation in a continuous integration environment by adopting the design test concept, a test-like program that automatically checks whether an implementation conforms to a specific design rule [2]. To answer RQ1.1 and RQ1.2, we conducted a case study [18].

The main contributions of this paper are: (i) a declarative DSL to specify and document design decisions about exception handling; (ii) an automatic verification tool to support the conformance check of exception handling design; and (iii) an automatic report generation to assist developers to find out which design rules are violated and locate in the source code the violation causes.

The remainder of this paper is organized as follows. Section 2 provides some background about exception handling design. The ArCatch solution is presented in Sect. 3 and the methodology, results and discussion of the case study are presented in Sect. 4. Finally, Sect. 5 discusses related work and Sect. 6 concludes the paper.

## 2 Exception Handling Design

In this section, we describe the exception handling concepts at the architecture level based on the IFTC (Idealized Fault-Tolerant Component) model (Sect. 2.1) and how design rules can be derived from it to express the exception handling design (Sect. 2.2).

### 2.1 Exceptions at the Architectural Level

At the software architecture level, exceptions and their control flow can be described using the IFTC model [14] (Fig. 1). It captures the essence behind the exception handling constructs of the mainstream object-oriented program languages [10], such as Java and C#. Each software component (callee) can receive service requests from other components (caller). The callee processes the request and sends back normal responses or exceptions.

Exceptions can be classified in three categories as depicted in Fig. 1: (i) interface exceptions - signaled when the request does not conform to the callee component service interface; (ii) failure exceptions - signaled to indicate that, for some reason, the callee component could not process the service request; and (iii) internal exceptions - raised and handled inside the callee component. The signaled exceptions are named external exceptions.

In the IFTC model, the component activity can be divided into normal and abnormal (exceptional) activities (Fig. 1). In the normal activity, the component processes service requests according to its specification. In the abnormal

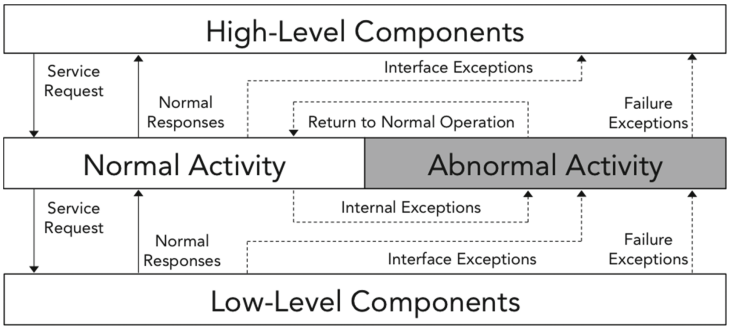


**Fig. 1.** Idealized Fault-Tolerant Component Model (adapted from [14]).

activity, the component performs contingency measures to deal with exceptions. Thus, a component can handle exceptions raised during its normal activity or exceptions signaled by low-level components (callees). However, exceptions that cannot be handled by a component are propagated to high-level components (callers) and so on. Moreover, before performing the exception propagation, a component can do either an exception re-raising or remapping. The exception re-raising occurs when the component captures the exception, performs some partial handling actions, and then re-raises it, forcing the exception propagation continuity. The remapping occurs when the component captures the exception, performs optionally some partial handling actions, and then raises another exception type, starting a new exception propagation.

At development time, the exception handling mechanism allows developers to define exceptions and structure the exception handling behavior by means of exception handlers. The exception handlers are component parts devoted to handle exceptions (gray parts in Fig. 1). At runtime, when an exception is raised, the exception handling mechanism deviates the normal control flow to the exceptional control flow, starting the search for an exception handler that can handle this exception. The search begins with the component in which the exception is raised and proceeds through all components in the service request chain in the reverse order in which they were called. When an appropriate handler is found, the exception handling mechanism passes the exception to the handler. After the exception is handled, the system may get back to its normal activity. Otherwise, if no handler is found, the system is forced to stop its execution.

### 2.2 Design Rules for Exception Handling

In the IFTC model, exceptions can be raised, signaled, handled, re-raised, and remapped by a system module and can flow through a list of several modules until be handled. These links can be expressed as different types of dependency relation between exceptions and modules at the architectural level. Based on such relations, dependency constraints can be derived to describe and to make

explicit how exceptions and modules can be combined towards expressing design rules governing the architectural exception handling design.

A set of design rules for exception handling can be expressed by applying semantic modifiers (e.g., "*must*", "*cannot*", "*only... can*", and "*can... only*") to constrain dependency relations types between modules and exception. Design rules can be used to document and make explicit the architect/designer intention/decision regarding the exception handling and its control flow. They can make explicit: (i) which modules can, cannot, or must raise, re-raise signal, or handle a specific exception type; (ii) which modules can, cannot, or must re-map a specific exception type to another; and (iii) which exception types can, cannot, or must flow through a specific list of modules. In this paper, we provide a way to express this kind of design rules and use it to check the exception handling design conformance to avoid erosion problems.

## 3 The Proposed Approach

To address RQ1, we developed ArCatch, which aims at providing a way to document architectural design decisions about exception handling and uses it to check the source code conformance. It is composed by a specification language (ArCatch.Rules) to express exception handling design rules, and a design rule checker (ArCatch.Checker) to automatically perform the conformance checking.

The overall flow of the ArCatch is depicted in Fig. 2. First, ArCatch.Checker receives as input the software source code under evaluation and the exception handling design rules written in ArCatch.Rules. Next, it performs the conformance checking and outputs a report describing which design rules are violated.

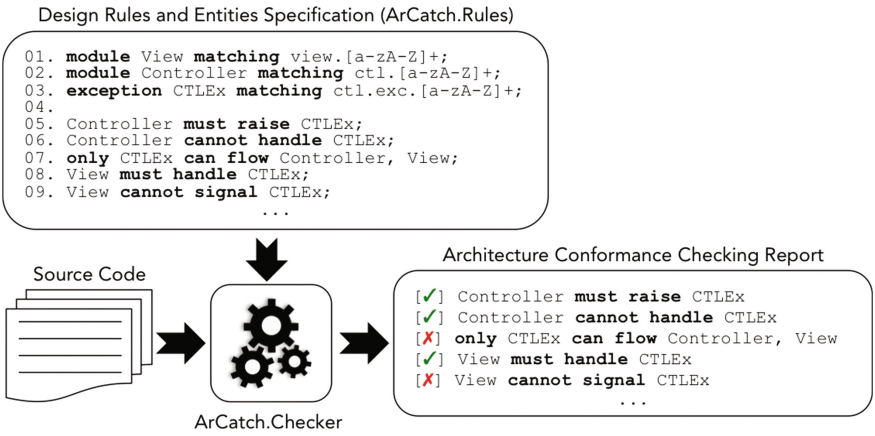


**Fig. 2.** The ArCatch Overview.

The architecture conformance checking report (at the bottom of Fig. 2) consists of a list of all specified design rules, which indicates the rules that passed

and the ones that did not. Such a report is useful for identifying which parts of the source code do not conform to the specification. For example, as shown in Fig. 2, the design rules specified at lines 5, 6, and 8 are valid, while the design rules specified at lines 7 and 9 are violated. Additionally, for all violated rules, ArCatch.Checker generates a counter example pointing out which parts of the source code are breaking the rules (see Sect. 4). Furthermore, both the exception handling design rules specification and the conformance checking report can help software architects and developers to better document, refine, implement and evolve architectural design decisions regarding exception handling.

ArCatch is implemented in Java and its current version provides support for exception handling conformance checking of Java programs. The ArCatch source code can be found at GitHub[1] and be freely downloaded. We detailed ArCatch.Rules and ArCatch.Checker next.

### 3.1 ArCatch.Rules: The Syntax

In our approach, the knowledge of software architects and developers about the exception handling design is documented as design rules using ArCatch.Rules (e.g., lines 05–09 in Fig. 2). Knowledge about the source code is very helpful when mapping the architectural elements (modules and exceptions) to its respective implementation elements (regular classes and exception classes). This knowledge is also documented in the module/exception declarations using ArCatch.Rules (e.g., lines 01–03 in Fig. 2). Hence, the exception handling design rules' specification is a knowledge-sharing artifact which both software architects and developers can use to fulfill their tasks.

The Grammar 1.1 describes a simplified version of ArCatch.Rules EBNF (Extended Backus–Naur Form). The exception handling design specification ⟨*spec*⟩ is composed by entities ⟨*entity*⟩ and rules ⟨*rule*⟩ declaration.

⟨*spec*⟩ ::= (⟨*entity*⟩ | ⟨*rule*⟩)*
⟨*entity*⟩ ::= (‘`module`’ | ‘`exception`’) ⟨*id*⟩ ‘`matching`’ ⟨*regex*⟩ ‘`;`’
⟨*rule*⟩ ::= (⟨*only-can*⟩ | ⟨*can-only*⟩ | ⟨*cannot*⟩ | ⟨*must*⟩) ‘`;`’
⟨*relation*⟩ ::= ‘`raise`’ | ‘`reraise`’ | ‘`signal`’ | ‘`handle`’ | ‘`remap`’ | ‘`flow`’
⟨*only-can*⟩ ::= ‘`only`’ ⟨*id*⟩ ‘`can`’ ⟨*relation*⟩ ⟨*id*⟩ [‘`to`’ ⟨*id*⟩ | (‘`,`’ ⟨*id*⟩)+]
⟨*can-only*⟩ ::= ⟨*id*⟩ ‘`can`’ ⟨*relation*⟩ ‘`only`’ ⟨*id*⟩ [‘`to`’ ⟨*id*⟩ | (‘`,`’ ⟨*id*⟩)+]
⟨*cannot*⟩ ::= ⟨*id*⟩ ‘`cannot`’ ⟨*relation*⟩ ⟨*id*⟩ [‘`to`’ ⟨*id*⟩ | (‘`,`’⟨*id*⟩)+]
⟨*must*⟩ ::= ⟨*id*⟩ ‘`must`’ ⟨*relation*⟩ ⟨*id*⟩ [‘`to`’ ⟨*id*⟩ | (‘`,`’⟨*id*⟩)+]

**Grammar 1.1.** The ArCatch.Rules EBNF simplified.

The entity declaration supports two types of architectural elements: modules and exceptions. A module represents a set of implementation classes, which can be grouped or interact with each other to provide a well-defined system functionality. An exception represents a set of exception classes (types). The keywords ‘`module`’ and ‘`exception`’ are employed in the module and exception

[1] https://github.com/lincolnrocha/ArCatch.

entities declaration respectively. Both modules and exceptions have an identifier ⟨*id*⟩ (a string that must start with a letter) and a regular expression ⟨*regex*⟩ (a sequence of characters that define a search pattern) used to map it onto implementation elements (regular or exception classes).

The rule declaration ⟨*rule*⟩ describes how ArCatch.Rules expresses exception handling design rules as dependency constraints between exceptions and modules. Such dependencies can be expressed in terms of exception raising, re-raising, signaling, handling, remapping, and flow. The semantic modifiers *only*... *can*, *can*... *only*, *cannot*, and *must* are used to give a proper semantic to each rule.

All derived design rules follow the same syntactic structure, which includes: (i) a fixed part that comprises ...⟨*id*⟩...⟨*relation*⟩...⟨*id*⟩; and (ii) an optional part that can be '`to`'⟨*id*⟩ or (',' ⟨*id*⟩)$^+$. In both parts, fixed and optional, the identifier ⟨*id*⟩ can refer to an exception or a module identifier. The choice depends on the type of dependency relation ⟨*relation*⟩ being taken into account.

When the keyword '`raise`', '`reraise`', '`signal`', or '`handle`' is chosen, the first identifier in the fixed part refers to a module identifier and the second one refers to an exception identifier. In such cases, there is no optional part.

If the keyword '`remap`' is chosen, the fixed part derivation is similar to the one for the other keywords, but the second identifier (the exception identifier) represents the exception type to be remapped. In the optional part '`to`'⟨*id*⟩, the identifier refers to an exception identifier, which comprises the exception types that are targeted by the exception remapping process.

Finally, if the keyword '`flow`' is chosen, the first identifier in the fixed part must refer to an exception identifier and the second one refers to a module identifier; the module where the exception type may be raised and signaled. In the optional part (','⟨*id*⟩)$^+$, each derived identifier refers to a module identifier. These identifiers are the list of modules in which the exception may flow through.

The ArCatch.Rules are implemented as a Java internal DSL, making it easy to incorporate it in a continuous integration environment by adopting the design tests concept [2], a programmatic approach to check the software source code against design rules via automated testing tools, such as JUnit.

### 3.2 ArCatch.Checker: The Semantics

The ArCatch.Checker is responsible for establishing a link between declared modules and exceptions to its implementing classes respectively, and checking the specified design rules against the software source code. Each entity (module or exception) has a regular expression associated with it. Every class name that matches the defined regular expression is linked to the corresponding entity.

ArCatch.Checker uses the following conventions about the exception handling dependency relations at the source code level: (i) `raise`($m$,$e$) means method $m$ raises exception $e$; (ii) `reraise`($m$,$e$) means method $m$ re-raises exception $e$; (iii) `signal`($m$,$e$) means method $m$ signals exception $e$; (iv) `handle`($m$,$e$) means method $m$ handle exception $e$; (v) `remap`($m$,$e$,$f$) means method $m$ remaps exception $e$ to exception $f$; and (vi) `flow`($e$,$m_1$,...,$m_n$) means exception $e$ is signaled by method $m_1$ and flows through $m_2$,...,$m_{n-1}$ until be handled by method $m_n$.

In the following, we introduce some basic definitions and the exception handling design rules violation semantics.

**Definition 1 (Implementation Class).** *An implementation class is a 3-tuple* $\langle n, t, \Phi \rangle$*, where* $n$ *is the class name,* $t$ *is the class type, and* $\Phi$ *is the class methods.*

**Definition 2 (Access Functions).** *Let* $c = \langle n, t, \Phi \rangle$ *be an implementation class, (i)* $\texttt{getName}(c)$ *returns the class name* $n$*, (ii)* $\texttt{getType}(c)$ *returns the class type* $t$*, and (iii)* $\texttt{getMethods}(c)$ *returns the set* $\Phi$ *of all class methods.*

**Definition 3 (Architectural Element).** *An architectural element* $A = \langle n, t, \phi \rangle$ *is a 3-tuple where* $n$ *is the element name,* $t \in \{\texttt{M}, \texttt{E}\}$ *is the element type, which can be a module type* (M) *or an exception type* (E)*, and* $\phi$ *is the regular expression used to map the implementation classes from source code.*

**Definition 4 (The `match` Function).** *Let* $\phi$ *be a regular expression and* $C$ *be a set of implementation classes, the function* $\texttt{match}(\phi, C) = \{c \mid c \in C \wedge \texttt{getName}(c) \in \omega(\phi)\}$ *returns all classes whose names matches* $\phi$*.* $\omega(\phi)$ *is all words described/matched by* $\phi$*.*

**Definition 5 (The `map` Function).** *Let* $A = \langle n, t, \phi \rangle$ *be an architectural element,* $C$ *be a set of implementation classes,* $\xi$ *be the root type of the exception types hierarchy, and* $<:$ *be a subtype relation where* $C$*,* $\xi$*, and* $<:$ *are defined in compliance to the rules of the underlying programming language used to build the system. The function* $\texttt{map}(A, C)$ *performs the mapping between an architectural element and its classes is defined as:*

$$\texttt{map}(A, C) = \begin{cases} t = \texttt{M},\ \{c \mid c \in \texttt{match}(\phi, C) \wedge \texttt{getType}(c) \not<: \xi\} \\ t = \texttt{E},\ \{c \mid c \in \texttt{match}(\phi, C) \wedge \texttt{getType}(c) <: \xi\} \end{cases}$$

**Definition 6 (The `methods` Function).** *Let* $M = \langle n, \texttt{M}, \phi \rangle$ *be a module and* $C$ *be a set of implementation classes, the function* $\texttt{methods}(M, C) = \{\ m \mid \forall c \in \texttt{map}(M, C),\ m \in \texttt{getMethods}(c)\}$ *returns all methods defined in each class of mapping* $\texttt{map}(M, C)$*.*

**Definition 7 (The `call` Function).** *Let* $m$ *and* $n$ *be methods and* $C$ *be a set of implementation classes, the function* $\texttt{call}(C, m, n)$ *returns* `true` *if* $\exists\ (c, d \in C \wedge m \in \texttt{getMethods}(c) \wedge n \in \texttt{getMethods}(d))$ *s.t. "*$n$ *calls* $m$*" and* `false` *otherwise.*

**Definition 8 (The `chains` Function).** *Let* $M_1, \ldots, M_n$ *be modules and* $C$ *be a set of implementation classes, the function* $\texttt{chains}(C, M_1, \ldots, M_n) = \{(m_1, \ldots, m_n) \mid \forall i \in [1, n),\ m_i \in \texttt{methods}(M_i, C) \wedge m_{i+1} \in \texttt{methods}(M_{i+1}, C) \wedge \texttt{call}(C, m_i, m_{i+1})\}$ *returns all method call chains of size* $n$ *starting in* $M_1$ *and ending in* $M_n$*.*

**Cannot Semantics**: (**Case 1**) Let $E = \langle eid, \texttt{E}, \phi_E \rangle$ be an exception, $M = \langle mid, \texttt{M}, \phi_M \rangle$ be a module, $\oplus$ be a relation in $\{\texttt{raise}, \texttt{reraise}, \texttt{signal}, \texttt{handle}\}$, and $S$ be a set of implementation classes. Rules of type "*mid* `cannot` $\oplus$ *eid*" are

violated if $\exists\ (m \in \texttt{methods}(M,S) \wedge e \in \texttt{map}(E,S))$, such that $\oplus(m,e)$. (**Case 2**) Let $E = \langle eid, \texttt{E}, \phi_E \rangle$ and $F = \langle fid, \texttt{E}, \phi_F \rangle$ be exceptions, $M = \langle mid, \texttt{M}, \phi_M \rangle$ be a module, and $S$ be a set of implementation classes. Rules of type "*mid* `cannot remap` *eid* `to` *fid*" are violated if $\exists\ (m \in \texttt{methods}(M,S) \wedge e \in \texttt{map}(E,S) \wedge f \in \texttt{map}(F,S))$, so that $\texttt{remap}(m,e,f)$. (**Case 3**) Let $E = \langle eid, \texttt{E}, \phi_E \rangle$ be an exception, $M_1 = \langle mid_1, \texttt{M}, \phi_{M_1} \rangle, \ldots,\ M_n = \langle mid_n, \texttt{M}, \phi_{M_n} \rangle$ be a list of $n$ modules, and $S$ be a set of implementation classes. Rules of type "*eid* `cannot flow` $mid_1, \ldots,$ $mid_n$" are violated if $\exists\ (e \in \texttt{map}(E,S) \wedge (m_1, \ldots, m_n) \in \texttt{chains}(S, M_1, \ldots, M_n))$, so that $\texttt{flow}(e, m_1, \ldots, m_n)$.

**Must Semantics**: (**Case 1**) Let $E = \langle eid, \texttt{E}, \phi \rangle$ be an exception, $M = \langle mid, \texttt{M}, \phi \rangle$ be a module, $\oplus$ be a relation in {`raise`, `reraise`, `signal`, `handle`}, and $S$ be a set of implementation classes. Rules of type "*mid* `must` $\oplus$ *eid*" are violated if $\nexists\ (m \in \texttt{methods}(M,S) \wedge e \in \texttt{map}(E,S))$, such that $\oplus(m,e)$. (**Case 2**) Let $E = \langle eid, \texttt{E}, \phi \rangle$ and $F = \langle fid, \texttt{E}, \phi \rangle$ be exceptions, $M = \langle mid, \texttt{M}, \phi \rangle$ be a module, and $S$ be a set of implementation classes. Rules of type "*mid* `must remap` *eid* `to` *fid*" are violated if $\nexists\ (m \in \texttt{methods}(M,S) \wedge e \in \texttt{map}(E,S) \wedge f \in \texttt{map}(F,S))$, so that $\texttt{remap}(m,e,f)$. (**Case 3**) Let $E = \langle eid, \texttt{E}, \phi \rangle$ be an exception, $M_1 = \langle mid_1, \texttt{M}, \phi \rangle, \ldots, M_n = \langle mid_n, \texttt{M}, \phi \rangle$ be a list of $n$ modules, and $S$ be a set of implementation classes. Rules of type "*eid* `must flow` $mid_1, \ldots,$ $mid_n$" are violated if $\nexists\ (e \in \texttt{map}(E,S) \wedge (m_1, \ldots, m_n) \in \texttt{chains}(S, M_1, \ldots, M_n))$, so that $\texttt{flows}(e, m_1, \ldots, m_n)$.

**Only-Can Semantics**: (**Case 1**) Let $E = \langle eid, \texttt{E}, \phi \rangle$ be an exception, $M = \langle mid, \texttt{M}, \phi \rangle$ be a module, $\oplus$ be a relation in {`raise`, `reraise`, `signal`, `handle`}, and $S$ be a set of implementation classes. Rules of type "`only` *mid* `can` $\oplus$ *eid*" are violated if $\exists\ (c \in S \setminus \texttt{map}(M,S) \wedge m \in \texttt{getMethods}(c) \wedge e \in \texttt{map}(E,S))$, such that $\oplus(m,e)$. (**Case 2**) Let $E = \langle eid, \texttt{E}, \phi \rangle$ and $F = \langle fid, \texttt{E}, \phi \rangle$ be exceptions, $M = \langle mid, \texttt{M}, \phi \rangle$ be a module, and $S$ be a set of implementation classes. Rules of type "`only` *mid* `can remap` *eid* `to` *fid*" are violated if $\exists\ (c \in S \setminus \texttt{map}(M,S) \wedge m \in \texttt{getMethods}(c) \wedge e \in \texttt{map}(E,S) \wedge f \in \texttt{map}(F,S))$, so that $\texttt{remap}(m,e,f)$. (**Case 3**) Let $E = \langle eid, \texttt{E}, \phi \rangle$ be an exception, $M_1 = \langle mid_1, \texttt{M}, \phi \rangle, \ldots, M_n = \langle mid_n, \texttt{M}, \phi \rangle$ be a list of $n$ modules, and $S$ be a set of implementation classes. Rules of type "`only` *eid* `can flow` $mid_1, \ldots,$ $mid_n$" are violated if $\exists\ (e \in S \setminus \texttt{map}(E,S) \wedge (m_1, \ldots, m_n) \in \texttt{chains}(S, M_1, \ldots, M_n))$, so that $\texttt{flows}(e, m_1, \ldots, m_n)$.

**Can-Only Semantics**: (**Case 1**) Let $E = \langle eid, \texttt{E}, \phi \rangle$ be an exception, $M = \langle mid, \texttt{M}, \phi \rangle$ be a module, $\oplus$ be a relation in {`raise`, `reraise`, `signal`, `handle`}, and $S$ be a set of implementation classes. Rules of type "*mid* `can` $\oplus$ `only` *eid*" are violated if $\exists\ (m \in \texttt{methods}(M,S) \wedge e \in S \setminus \texttt{map}(E,S))$, such that $\oplus(m,e)$. (**Case 2**) Let $E = \langle eid, \texttt{E}, \phi \rangle$ and $F = \langle fid, \texttt{E}, \phi \rangle$ be exceptions, $M = \langle mid, \texttt{M}, \phi \rangle$ be a module, and $S$ be a set of implementation classes. Rules of type "*mid* `can remap only` *eid* `to` *fid*" are violated if $\exists\ (m \in \texttt{methods}(M,S) \wedge ((e \in \texttt{map}(E,S) \wedge f \in S \setminus \texttt{map}(F,S)) \vee (e \in S \setminus \texttt{map}(E,S) \wedge f \in \texttt{map}(F,S)) \vee (e \in S \setminus \texttt{map}(E,S) \wedge f \in S \setminus \texttt{map}(F,S)))$, so that $\texttt{remap}(m,e,f)$. (**Case 3**) Let $E = \langle eid, \texttt{E}, \phi \rangle$ be an exception, $M_1 = \langle mid_1, \texttt{M}, \phi \rangle, \ldots, M_n = \langle mid_n, \texttt{M}, \phi \rangle$ be a list of $n$ modules, and $S$ be a set of implementation classes. The rules of type "*eid* `can flow oly` $mid_1, \ldots,$

$mid_n$" are violated if $\exists\ (e \in \texttt{map}(E,S) \wedge (m_1,\ldots,m_k) \notin \texttt{chains}(S,M_1,\ldots,M_n))$, so that $\texttt{flows}(e,m_1,\ldots,m_k)$ with $k > 1$.

In ArCatch.Checker, all source code information relevant for the checking process is extracted using the Design Wizard[2] tool and the Java Compiler Tree API[3]. The Design Wizard provides means to extract the program class dependencies, such as class inheritance trees and method call-graphs to feed our design rules checking algorithm. The Compiler Tree API provides support to inspect the AST (Abstract Syntax Tree) of Java programs, helping in the identification whether raising, re-raising, and remapping cases occurs in the source code.

## 4 Case Study

In this section, we describe the design employed to conduct the case study (Sect. 4.1), the associated results (Sect. 4.2) and threats to validity (Sect. 4.3).

### 4.1 Case Study Design

**The Case and Unit of Analysis.** The case and unit of analysis is an open source system called Health Watcher (HW), which was developed to improve the quality of the services provided by health care institutions in Brazil. HW is a Java web-based system that allows citizens to register complaints regarding health issues, so that associated health care institutions can promptly investigate the complaints and take the required actions [21]. The HW system was chosen because it has a clear exception handling design and has been used in several empirical studies regarding software modularity and exception handling [9,11, 16] (conveninece sampling). Furthermore, considering the importance of data triangulation in case studies, it was important to select a system whose software architect would be available for a follow-up interview, which was the case of HW.

HW follows a multilayered architectural style composed by 4 layers: view layer (`ViL`), "the highest layer", distribution layer (`DiL`), business layer (`BuL`), and data layer (`DaL`), "the lowest layer". We analyzed 10 versions of HW. All evaluated 10 versions are available on the Web[4] and it varies from 7070 up to 100054 lines of code, 80 up to 136 classes, and 19 up to 25 packages.

**Data Collection.** The data used in our analysis was collected through two methods: repository mining and unstructured interview. We employed repository mining to extract the code of HW. We conducted two unstructured interviews with the software architect of HW. The goal of the first interview was to confirm the exception handling strategy employed in HW. In the second interview, we discussed the results of our analysis with the software architect, to collect additional insights about the results. The interviews were conducted via Skype in October and December 2016 and took about 20 and 50 min respectively.

[2] https://github.com/joaoarthurbm/designwizard.
[3] https://docs.oracle.com/javase/7/docs/jdk/api/javac/tree/.
[4] http://ptolemy.cs.iastate.edu/design-study/#healthwatcher.

We made notes during the interviews and discussed the notes with the interviewee, to ensure that the notes reflected the content of the interview.

**Data Preparation.** To be able to evaluate our approach, we have to conduct some prepation of the mined source code. Each layer of HW architecture was represented as a module (Listing 1.1, line 1) and mapped to the corresponding implementation classes at the source code level. Listing 1.1 shows this mapping[5] performed to the v1 version of HW. The `viL` module was mapped (Listing 1.1, line 2) to all classes of package "`healthwatcher.view.servlets`". The `diL` module was mapped to all classes of package "`lib.distribution.rmi`" and the `IFacade`, `HealthWatcherFacade`, and `HealthWatcherFacadeInit` classes (Listing 1.1, line 3). The `buL` module was mapped to all classes of subpackages of "`healthwatcher.business`" (Listing 1.1, line 4). Finally, the `daL` module was mapped to all classes of packages and subpackages of "`healthwatcher.data`" and "`lib.persistence`" (Listing 1.1, line 5).

**Listing 1.1.** Health Watcher Layers Mapping.

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

All exceptions defined in version v1 are in the package `lib.exceptions`. Based on the HW documentation and the source code analysis, six groups of exceptions were defined and mapped, as shown in Listing 1.2. The `diLEx` exception represents the user-defined exceptions (i.e., defined by the programmer) related to the `DiL` layer, `buLEx` exceptions are related to the `BuL` layer, and `daLEx` exceptions are related to the `DaL` layer. The `svtEx` and `sqlEx` are platform-defined exceptions and `allEx` represents all user-defined exceptions.

**Listing 1.2.** Exceptions Mapping.

```
1 ExceptionElement diLEx, buLEx, daLEx, sqlEx, svtEx, allEx;
2 diLEx = exception("DiLE").matching("(java.rmi.RemoteException|lib.
      exceptions.CommunicationException)*").build();
3 buLEx = exception("BuLE").matching("lib.exceptions.(ObjectAlready)\\w*").
      build();
4 daLEx = exception("DaLE").matching("lib.exceptions.(Persistence|ObjectNot|
      Repository|Transaction)\\w*").build();
5 sqlEx = exception("SQLE").matching("java.sql.SQLException").build();
6 svtEx = exception("SVTE").matching("javax.servlet.ServletException").build
      ();
7 allEx = exception("AllE").matching("lib.exceptions.(\\w*.)*\\w*").build();
```

Note that once the system evolves, the mappings of classes into layers also changes. Thus, for each HW version, it was necessary to perform some fine-tunes in the mapping to capture changes occurred from one version to other.

[5] The symbol "\w" represents a word character: `[a-zA-Z_0-9]`.

Another step in the data preparation was to define an exception handling policy to evaluate the HW exception handling design based on the intention of HW's software architect (collected via an unstructured interview) and good practices recommended by the Oracle's BluePrints Design Patterns[6] for multi-layered architectures of Java systems. This policy states that an exception can be raised in or signaled by an arbitrary layer. When a specific layer (callee) signals an exception, such exception can only propagate to the immediately upper layer (caller), which is responsible for catching the exception and performing handling actions (*catch-and-handle* strategy). This puts the system back in its normal control flow. If this exception cannot be handled in this scope, the caller layer must perform an exception type remapping and signal the new exception type to the next upper layer (*catch-and-remap* strategy). This process repeats until the exceptional situation is finally handled at an upper layer. Exceptions signaled by third-party components to a specific layer must be handled in this layer or be remapped and propagated to the next upper layer.

Table 1 shows all design rules defined to enforce the established policy. Each rule enforces a specific aspect of the exception handling policy. For instance, the rule R01 enforces that exceptions signaled by the under layer `diL` must be handled by the upper layer `viL` (*catch-and-handle*). The rules R06 and R10 have a similar purpose. The rules R07 and R11 ensure that the *catch-and-remap* strategy is used. The rules R14 and R15 enforce that `sqlEx` exceptions signaled by third-party components must be handled by `daL` module. The rule R03 enforces that no user-defined exception can be signaled by `viL`. Finally, the R16 enforces that `daLEx` exceptions cannot flow through modules `daL`, `buL`, and `diL`.

### 4.2 Results and Discussion

Table 1 summarizes the evaluation results. Each HW version is checked against the same set of 16 design rules. All versions fully comply with six design rules (R02, R04, R07, R11, R12 e R14) and do not conform to 7 design rules (R01, R05, R06, R08, R09, R10, and R16). On one hand, R03 and R13 start to be violated in versions 10 and 4, respectively. On the other hand, R15 is violated in version 9 and starts be satisfied in the last version. In short, versions v1–v3 and v4–v10 has a 50% and 44% of conformance degree respectively.

Looking at the ArCatch.Checker conformance report (Listing 1.3), the R03 is violated in version 10 because the method `initFacade()` of class `HWServlet` starts to signal the exception `CommunicationException` after the modularization of exception handling code. The R013 starts to violate in version 4 because the implementation of Observer pattern, after that the method `notify()` of class `Subject` starts signaling the exceptions (Listing 1.4) `ObjectNotFoundException`, `RepositoryException`, `ObjectNotValidException` e `TransactionException`.

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

**Table 1.** Exception handling design rules and checking results.

| ID | Exception handling design rule | Health watcher versions | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 |
| R01 | module(viL).mustHandle(diLEx).build() | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| R02 | only(viL).canSignal(svtEx).build() | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| R03 | module(viL).cannotSignal(allEx).build() | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ |
| R04 | only(diL).canRaise(diLEx).build() | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| R05 | only(diL).canSignal(diLEx).build() | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| R06 | module(diL).mustHandle(buLEx).build() | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| R07 | only(diL).canRemap(buLEx).to(diLEx).build() | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| R08 | only(buL).canRaise(buLEx).build() | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| R09 | only(buL).canSignal(buLEx).build() | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| R10 | module(buL).mustHandle(daLEx).build() | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| R11 | only(buL).canRemap(daLEx).to(buLEx).build() | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| R12 | only(daL).canRaise(daLEx).build() | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| R13 | only(daL).canSignal(daLEx).build() | ✓ | ✓ | ✓ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| R14 | only(daL).canHandle(sqlEx).build() | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| R15 | module(daL).cannotSignal(sqlEx).build() | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✓ |
| R16 | exception(daLEx).cannotFlow(daL, buL, diL).build() | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |

**Listing 1.3.** Rule R03 Example Report (HW v10).

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

**Listing 1.4.** Rule R13 Example Report (HW v4).

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

The R15 is violated from versions 1 until version 9, starting to be satisfied in version 10. These violations occurred because some classes of `daL` signal the exception `SQLException`. However, after the modularization of exception handling code in version 10, such violations no long occurred. The full set of conformance checking reports for all 10 versions of HW can be found on the paper's website[7].

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

**Listing 1.5.** Rule R15 Example Report (HW v1–v9).

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

To further validate the results, we interviewed the HW's software architect. First, we discussed the seven design rules that are violated in all versions. Thus, after looking at the violation report, he recognized that all violations represented clear deviations from his intention as software architect, confirming the existence of exception handling erosion problems in HW.

Second, we discussed the violation of R03 and R13. Regarding R03, the architect recognized that such violation introduced in version 10 is a mistake made by a developer and a possible solution could be create a try-catch bock on method `initFacade()` to catch the exception `CommunicationException` and perform an log operation or a page redirection to proper present the error. Regarding R13, the architect argued that such violation is not a proper violation itself, but a side effect caused by implementation of Observer design pattern. However, he decided that it must be fixed in a future version. Finally, looking at the violation report of R15, the architect had no doubt that such violation represents a deviation from his intention, which was fixed in version 10.

No evaluation regarding performance and usability of ArCatch was conducted in this paper. However, in the evaluation scenario, ArCatch takes about 50 s (average) to perform the source code analysis and check the conformance of each design rule in each HW version. After define the exception handling policy, the specification of all design rules using ArCatch.Rules takes less then 1 h. The mapping process was the most time consuming part, once we were not familiar with its source code; we needed to analyze manually the source code of each version. The first version analysis took more than 5 h, while the sum of all other versions analysis took about of 5 h, i.e. the entire mapping process took 10 h.

### 4.3 Threats to Validity

The threats to validity associated with our investigation are discussed using the classification by Runeson and Höst [18]. Since no causal relationship was investigated in the case study, we do not discuss internal validity threats.

**Reliability** validity threats are related to the repeatability of a study, i.e. how dependent are the research results on the researchers who conducted it [18]. We minimized this threat by involving several researchers in the design and execution of our investigation. Furthermore, our observations and findings were verified by HW's software architect to avoid false interpretations.

**Construct** validity threats reflect whether the measures used really represent the intended purpose of the investigation [18]. To mitigate this threat, we collected data using multiple methods (data triangulation). Moreover, some information about source code is extracted and represented as a static call-graph. However, some relations represented in the static call-graph can never occur in actual program runs. In fact, it is an undecidable problem. The static call-graph

provides over-approximative information, which can lead ArCatch to find rule violations that may never happen at runtime (i.e., false alarms).

**External** validity threats limit the generalization of the findings of the investigation [18]. Since we employed the case study method, our findings are strongly bounded by the context of our study. In addition, the investigated case involved only one product, which is not used intensively by different users. To mitigate this threat, we made an attempt to detail the context of our study as much as possible. However, this is a strong limitation of our study, which we intend to address by evaluating our approach by conducting other case studies.

## 5 Related Work

We have reviewed the state of the art on architecture conformance checking solutions focusing on their support to the exception handling conformance checking.

The Semmle .QL [15] is a conformance checking solution where design constraints are specified as queries performed in the software source code. The .QL syntax is inspired in the SQL language. The LogEn [8] solution is based on dependency relations between implementation elements of different levels of granularity. LogEn provides a visual DSL as an Eclipse IDE plug-in to specify the mapping between architectural and implementation elements and express dependency constraints. Both solutions adopt Datalog, a logical query language, to perform the conformance checking. Regarding the exception handling conformance checking, LogEn only provides support to deal with raising and handling relations, while .QL only provides support to handling relation.

The DCL Suite [22], TamDera [11], and Dictō [5] provide a textual external DSL to describe dependence constraints between system modules and a checker to verify the compliance between the implemented and intended architectures. TamDera provides means to deal with architectural degradation in terms of erosion and drift problems, while DCL Suite and Dictō only provide support to deal with architectural erosion problems. In contrast to DCL Suite and TamDera, which provides their own conformance checker implementation, Dictō performs the conformance checking using existent conformance tools (e.g., JPF and PMD). Both DCL Suite and TamDera have been developed as a plug-in for Eclipse IDE. Therefore, programmers can carry out the conformance checking process as the source code is being written. The Dictō has been developed as an IDE agnostic solution that can be easily integrated in a static analysis tools such as SonarQube. Regarding the exception handling conformance checking, TamDera and Dictō only provide support to signaling and handling relations, while DCL Suite only provide support for signaling relations.

The EPL [1] is a conformance checking solution devoted to check the conformance of exception handling policies in Java programs. In EPL, an exception handling policy is a set of design decisions governing the exceptions usage in a software project. EPL provides an external DSL to describe exception handling policies involving exceptions and compartments, which is a language constructor used to express which classes and methods are taken into account in the

conformance checking process. EPL has been developed as a plug-in for Eclipse IDE and the conformance checking can be performed as the source code is being written. EPL provides its own conformance checker based on the Eclipse JDT. Regarding the exception handling conformance checking, EPL is the most complete of the solutions we analyzed, only without support to express and check dependency constraints related to the exceptional control flow.

## 6 Conclusion and Future Work

In this paper, we have presented ArCatch, a conformance checking solution that tackles the exception handling erosion problem (RQ1). ArCatch aims at enforcing exception handling design decisions in Java projects, by providing: (i) a declarative language (ArCatch.Rules) for expressing design constraints regarding exception handling; and (ii) a rule checker (ArCatch.Checker) to automatically verify the exception handling conformance. Furthermore, ArCatch provides support for several kinds of dependence relation concerning the exception handling design, such as raising, re-raising, remapping, signaling, handling, and flow.

To evaluate our approach (RQ1.1 and RQ1.2), we conducted a case study and identified that: (i) at least 7 design rule violations in each version were detected; (ii) all versions conform to 6 design rules; and (iii) three violations appear in three different versions. ArCatch proved useful in the identification of existing exception handling erosion problems and its causes. This erosion can be avoided if adopting our approach in the system project since the beginning.

As future work, we plan to perform a user-centric evaluation to analyze ArCatch in terms of performance, scalability, usability, and learning curve. We also want to analyze if it is possible to use the ArCatch.Rules specifications to derive software tests for the exception handling code. Finally, we intend to conduct other case studies involving companies from different domains.

## References

1. Barbosa, E.A., Garcia, A., Robillard, M.P., Jakobus, B.: Enforcing exception handling policies with a domain-specific language. IEEE Trans. Softw. Eng. **42**(6), 559–584 (2016)
2. Brunet, J., Guerrero, D., Figueiredo, J.: Design tests: an approach to programmatically check your code against design rules. In: 31st International Conference on Software Engineering, pp. 255–258, May 2009
3. Cacho, N., Barbosa, E.A., Araujo, J., Pranto, F., Garcia, A., Cesar, T., Soares, E., Cassio, A., Filipe, T., Garcia, I.: How does exception handling behavior evolve? an exploratory study in Java and C# applications. In: ICSME 2014, pp. 31–40. IEEE (2014)
4. Cacho, N., César, T., Filipe, T., Soares, E., Cassio, A., Souza, R., Garcia, I., Barbosa, E.A., Garcia, A.: Trading robustness for maintainability: an empirical study of evolving c# programs. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 584–595 (2014)

5. Caracciolo, A., Lungu, M., Nierstrasz, O.: A unified approach to architecture conformance checking. In: Proceedings of the 12th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 41–50. ACM Press, May 2015
6. Chang, B.M., Choi, K.: A review on exception analysis. Inf. Softw. Technol. **77**(C), 1–16 (2016)
7. Ebert, F., Castor, F., Serebrenik, A.: An exploratory study on exception handling bugs in java programs. J. Syst. Softw. **106**(C), 82–101 (2015)
8. Eichberg, M., Kloppenburg, S., Klose, K., Mezini, M.: Defining and continuous checking of structural program dependencies. In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 391–400. ACM (2008)
9. Ferrari, F., Burrows, R., Lemos, O., Garcia, A., Figueiredo, E., Cacho, N., Lopes, F., Temudo, N., Silva, L., Soares, S., Rashid, A., Masiero, P., Batista, T., Maldonado, J.: An exploratory study of fault-proneness in evolving aspect-oriented programs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, ICSE 2010, pp. 65–74. ACM, New York (2010)
10. Garcia, A.F., Rubira, C.M., Romanovsky, A., Xu, J.: A comparative study of exception handling mechanisms for building dependable object-oriented software. J. Syst. Softw. **59**(2), 197–222 (2001)
11. Gurgel, A., Macia, I., Garcia, A., Staa, A., Mezini, M., Eichberg, M., Mitschke, R.: Blending and reusing rules for architectural degradation prevention. In: Proceedings of the 13th International Conference on Modularity, pp. 61–72. ACM (2014)
12. van Gurp, J., Bosch, J.: Design erosion: problems and causes. J. Syst. Softw. **61**(2), 105–119 (2002)
13. Kechagia, M., Spinellis, D.: Undocumented and unchecked: exceptions that spell trouble. In: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, pp. 312–315. ACM, New York (2014)
14. Lee, P.A., Anderson, T.: Fault Tolerance: Principles and Practice. Dependable Computing and Fault-Tolerant Systems, 2 edn., vol. 3. Springer, Wien (1990)
15. Moor, O.d., Verbaere, M., Hajiyev, E., Avgustinov, P., Ekman, T., Ongkingco, N., Sereni, D., Tibble, J.: Keynote address: .ql for source code analysis. In: SCAM 2007, pp. 3–16. IEEE Computer Society, Washington, DC (2007)
16. Oizumi, W.N., Garcia, A.F., Colanzi, T.E., Ferreira, M., Staa, A.V.: On the relationship of code-anomaly agglomerations and architectural problems. J. Softw. Eng. Res. Dev. **3**(1), 1–22 (2015)
17. Passos, L., Terra, R., Valente, M., Diniz, R., Mendonça, N.C.: Static architecture-conformance checking: an illustrative overview. IEEE Softw. **27**(5), 82–89 (2010)
18. Runeson, P., Höst, M., Rainer, A., Regnell, B.: Case Study Research in Software Engineering: Guidelines and Examples. Wiley, Hoboken (2012)
19. Shah, H., Gorg, C., Harrold, M.J.: Understanding exception handling: viewpoints of novices and experts. IEEE Trans. Softw. Eng. **36**(2), 150–161 (2010)
20. de Silva, L., Balasubramaniam, D.: Controlling software architecture erosion: a survey. J. Syst. Softw. **85**(1), 132–151 (2012)
21. Soares, S., Laureano, E., Borba, P.: Implementing distribution and persistence aspects with aspectj. In: OOPSLA 2002, pp. 174–190. ACM, New York (2002)
22. Terra, R., Valente, M.T.: A dependency constraint language to manage object-oriented software architectures. Softw. Pract. Exper. **39**(12), 1073–1094 (2009)