New Exception Interfaces for Java-like Languages

Thiago B. L. Silva
Informatics Center, Federal University of Pernambuco
Recife-PE, Brazil
tbls@cin.ufpe.br

Fernando Castor
Informatics Center, Federal University of Pernambuco
Recife-PE, Brazil
castor@cin.ufpe.br

ABSTRACT
The Java programming language allows developers to specify the exception interface of a method using the throws clause. This mechanism supports static checking that is coherent with a statically-typed language. However, it is known to have limitations. In particular, throws clauses hinder understandability and maintainability due to their scattered nature and lack of information about the source of an exception, the place where it is handled, and everything in between. In this paper, we propose a simple form of exception interface specification, called Exception Propagation Channel (EPC), that provides a global view of exception flow and complements Java’s exception interfaces. Our approach stems from the observation that throws clauses scale well for methods that throw exceptions to large numbers of direct callers (horizontally), but not for long chains of method calls (vertically). We present an extension to the Java language, called Epic-Java that incorporates the proposed approach as well as a prototype compiler and a reengineering tool for that extension. Application of the proposed approach to three open-source systems resulted in a considerable reduction in the number of throws clauses and a comparatively smaller number of EPCs, thus suggesting that maintainability is improved.

Categories and Subject Descriptors
D.3 [Programming Languages]: Language Constructs and Features

1. INTRODUCTION
Exception handling [6] is a technique for improving software modularity and reuse in the presence of exceptional conditions. Many mainstream programming languages, such as Java, Ada, C++, Ruby, and C#, implement exception handling mechanisms. These languages provide constructs to indicate the occurrence of an error (throw an exception) and to associate a set of recovery actions with the error to remedy the problem (handle the exception). Error recovery measures are implemented within exception handlers.

The Java programming language requires developers to either handle the exceptions that a method encounters or indicate in the method’s throws clause (the exception interface of the method) the ones that it does not. This requirement applies to the so-called checked exceptions, subclasses of the Exception class that are not subclasses of RuntimeException. It is checked statically by the Java compiler and failure of a program to meet it results in a compilation error.

Problem. In the mid-2000’s, well-known developers took part in a debate [16, 15, 5] pertaining to the use of checked exceptions. This idea, albeit old [9], had not been incorporated in a mainstream language until the creation of Java, in the mid 90’s. Many developers involved in the debate argued that throws clauses hindered maintainability and even reliability, thus defeating the purpose of using checked exceptions. Moreover, due to the pervasiveness of checked exceptions in Java, developers tended to ignore them, using very general exception types in throws clauses, such as Exception, or wrapping them with unchecked exceptions [2]. However, the root for these problems does not stem from checked exceptions specifically. Instead, it is a consequence of the tendency of early exception handling research [10, 9] to present exceptions as the results that functions produce when they are unable to provide a normal response. Considering this local perspective, we say that throws clauses implement horizontal exception interfaces (Figure 1(a)). This pairing of exceptions with normal responses ignores the global nature of exceptions. The caller of a function often cannot handle an exceptional result in a useful way, differently from normal function results, so exceptions must often be handled higher up in the function call chain, and many exception handling mechanisms automatically propagate any exception that a method encounters but does not handle.

Even so, existing approaches to specify exception interfaces do not follow this trend. Instead, they mirror method return types, providing only local information about the exceptions that a method throws to its callers.

Proposed Approach. We believe that exception interfaces should support localized specification of global exception propagation. In other words, exception interfaces should be both horizontal and vertical (Figure 1(b)). In this paper, we propose an approach that complements throws clauses and localizes the specification of exception interfaces for chains of method calls. We call this approach to vertical exception interface Exception Propagation Channels (EPCs).
We emphasize, though, that EPCs and throws clauses complement each other, by addressing different issues. We present an extension to the Java language called EPiC-Java, that incorporates the proposed approach, as well as a prototype compiler and a reengineering tool for that extension.

We have reengineered three open-source applications to use exception propagation channels. We identified throws clauses referring to vertical exception interfaces and replaced these clauses by EPCs. For all of the target applications it was possible to eliminate more than 40% of the throws clauses by employing EPCs. Moreover, the resulting number of EPCs is considerably smaller, amounting to approximately 36% of the number of removed throws clauses in the worst case. As a consequence, the specification of exception interfaces became less scattered in these systems. This result suggests that EPCs can ameliorate some of the maintenance issues of using only throws clauses.

2. BACKGROUND

In the exception handling models of all the mainstream programming languages, developers specify the places that throw and handle exceptions. At runtime, when a method throws an exception, the underlying exception handling mechanism is responsible for changing the normal control flow of the program to its exceptional control flow [4]. However, the places through which exceptions pass along the way from throw site to handler are established implicitly, depending on the current configuration of the call stack. Exceptions may be implicitly propagated through many methods before reaching an appropriate handler [4], potentially affecting the program behavior along the way.

Currently, there is no way to specify the global effect of exception occurrences, and exception control flows are not explicitly declared. The Java programming language employs an approach to mitigate this problem. It requires developers to specify in the signature of each method the list of checked exceptions that it may throw, associated with a throws clause. Checked exceptions are subclasses of the Exception class that are not subclasses of RuntimeException. This list of exceptions, usually called the method’s exception interface [11], is employed as documentation of exceptional control flows. In addition, the Java compiler can statically check (hence the name “checked exception”) whether all the exceptions that a method throws appear in its exception interface. However, this solution is inherently local to each method and the problem of implicit control flows is intrinsically global [13]. Moreover, it raises maintenance issues during the evolution of the software.

Since checked exceptions require that every method whose instructions might throw a checked exception must explicitly handle the exception in its body or include in its exception interface the corresponding types of checked exceptions that it might throw, maintenance becomes problematic when changing the exception clause of a method because these changes often require manual changes in the interfaces of the callers, throughout the propagation call chain, as deep in the call chain as the handler for the exception type may be. For example, consider Figure 1(b), where m1–4 are methods and an arrow from a method a to a method b indicates that b invokes a. If, during software maintenance, a new exception F is added to the throws clause of m1(), the throws clauses of m2(), m3(), and m4() must be modified to include F as well if F does not concern them. Furthermore, changes in exception clause of overridden methods may require manual changes in the interface of all the overridden methods up in the hierarchy. Thus, not only methods which are only propagating exceptions need manual change, but removing a throwing instruction from a method can easily lead to spurious handling code and types in exception clauses in a call chain for a checked exception that doesn’t exist anymore.

These problems can be avoided by employing a different notion of exception interface. Java’s throws clause is useful when the exceptions that a method throws are handled by its direct callers. In this scenario, it does not matter how many callers a method has because its throws clause applies to all of them. Besides, changes to either callers or callee that do not affect the exception interface do not trigger non-local changes. Because of these features, we say that exception interfaces in Java are horizontal. However, there are scenarios where horizontal exception interfaces are insufficient. If a maintenance activity requires modifications to the exception interface of a method and its callers do not handle the checked exceptions that appear in its exception interface, their exception interfaces might have to be modified as well. This applies to the callers of the callers and so on. Since horizontal exception interfaces are local to each method, they fail to provide the global perspective that developers need in order to understand exception propagation in non-trivial systems. To address these problems, we claim that exception interfaces should also be vertical, in the sense that they should connect not only methods and their direct callers but also the throw sites and the handle sites of exceptions. Existing languages provide no support for specifying this kind of interface. In summary, to maximize maintainability, exception interfaces should be bi-dimensional.

The issues mentioned above have been addressed by previous work on the EJFlow language [3] and Anchored Exception Declarations [14]. EJFlow “extends the AspectJ [8] programming language with the aim of promoting enhanced robustness and program modularization”. Because it extends AspectJ, it relies heavily on aspect-oriented programming [7], which adds to the complexity of the solution. Furthermore, it relies on static analysis, which means that its accuracy is a consequence of the accuracy of the underlying analysis. Finally, EJFlow exception interfaces are not compatible with throws clauses. On the other hand, Anchored Exception Declarations define a construct to allow relative specification of exception interfaces. Since the exception interface of a method can be defined in terms of another’s the aforementioned maintainability problem is, to a large
extent, solved. However, it does not provide a global view of exceptions.

3. NEW EXCEPTION INTERFACES

In this section we present the proposed approach. We start by outlining the design goals for our extended notion of exception interface (Section 3.1). We then introduce the concept of EPC (Section 3.2) and present EPiC-Java, a Java extension that implements EPCs (Section 3.3). We also describe the tools that we have developed to support EPiC-Java: a prototype compiler (Section 3.4) and a reengineering tool (Section 3.5).

3.1 Design Goals

Our proposal to extend the current notion of exception interface is a consequence of a set of design goals. These goals aim to build on the strengths of throws clauses and related approaches (Section 2) while addressing their main weaknesses.

Ease of maintenance (DG1). If maintenance activities introduce new checked exceptions in a system, only a small part of the exception interfaces should have to be modified. In the same vein, the introduction of new method calls in methods that propagate exceptions should have minimal impact on the exception interfaces of these methods. In summary, our proposal should be both horizontal, in the sense that it should be easy to add new direct callers to a method that throws exceptions, and vertical, in the sense that the inclusion of new methods in a chain of method calls through which exceptions flow should have little or no impact in the specification of exception interfaces.

Backward compatibility with throws clauses (DG2). It should be possible to add EPCs to existing Java systems without the need to modify their implementation, incrementally. In fact, throws clauses should be just a special case of EPC. This greatly simplifies the semantics of the language.

Static checking (DG3). Checked exceptions, similarly to statically typed languages, can reduce the number of implementation errors in software development activities. We believe that the problems with Java’s checked exceptions stem from the exclusive use of the throws clause approach, not from static checking.

Global view (DG4). A throws clause provides a local view of the exceptions that clients of a method might encounter. For API code, this is useful documentation since library developers cannot foresee the ways in which library clients will use it. In this scenario, the throws clause forces the client to do something about potential exceptions. However, in application softwares, developers often know where an exception is expected to be handled. Oftentimes, it is not in the same method that triggered it. In fact, the handling method is frequently located much lower in the stack.

3.2 Exception Propagation Channels

Informally, an Exception Propagation Channel (EPC) can be mapped to a set of methods through which an exception flow, from the method that throws it, the throwing site of the EPC, to the method that is expected to handle it, the handling site of the channel. Moreover, along the way, it can pass through other methods, the intermediate sites of the channel. More specifically, an EPC is a 4-tuple \((EXCS, TS, HS, IS)\). \(EXCS\) is the set of types of the exceptions (subtypes of \texttt{Exception} class) that flow through the EPC. \(TS\) is the set of throwing sites of the EPC, methods that throw every \(e \in EXCS\). Both the \(EXCS\) and \(TS\) sets must have at least one element. For example, the exception flow in Figure 2 has \(EXCS = \{E\}\) and \(TS = \{C1.t1(int,String)\}\).

The \(HS\) set contains zero or more handling sites of the EPC. Handling sites, like throwing sites, correspond to methods in a program. A handling site is a destination for exceptions that flow through an EPC (instances of \texttt{EXCS} elements). For every method \(m \in HS\), there is a path in the call graph of the program from \(m\) to every other method \(t \in TS\). Method \(m\) is responsible for either handling each exception \(e \in EXCS\) or making it part of its exception interface. An alternative design would be to require that \(m\) throw \(e\) as well. However, that would mean that the channel would not include information about the method responsible for handling the exception. At the same time, requiring \(m\) to handle the exception by catching it would make no sense.

Exception interfaces in Java can be seen as syntactic sugar to ease the burden imposed by more conservative designs, such as CLU’s [9], which require exceptions to be handled at every level. If they did not exist, methods would have to explicitly catch every checked exception they encountered. An EPC where both the \(HS\) and \(IS\) sets are empty is equivalent to a set of throws clauses, one for each method in \(TS\). As a consequence, we can say that throws clauses are just a special case of EPCs. Considering the example of Figure 2, \(HS = \{C3.h\}\).

Finally, \(IS\) is a set of zero or more sequences of intermediate sites of the EPC. Considering the call graph of a program and the paths from handling sites to throwing sites, for every path \(P\) between a handling site and a throwing site, there is a \(s = (i_1, i_2, ..., i_n) \in IS\) sequence of distinct methods such that all methods in \(s\) are elements of \(P\) and appear in the same order as in \(P\). Also, every method in \(s\) throws every \(e \in EXCS\). For the example of Figure 2, \(IS = \{C2.h\}\).

3.3 The EPiC-Java Language

The EPiC-Java language is a simple extension to Java that materializes the concept of EPC. In EPiC-Java, EPCs are specified by propagate statements which can appear in any Java source file in the top level scope (outside class declarations). The following snippet presents a propagate statement in its simplest form:

```
propagate E: C1.t1();
```

where \(E\) is an exception type and \(C1.t1()\) is a method signature, both references to previously declared constructs. The example above materializes an EPC with a single throw-
Figure 3: An exception flow with multiple paths, where C1.t() is a throwing site of the checked exceptions E and F and an indirect caller, C5.h(), handles it.

Also, different EPCs may intersect, adding to the information of a method’s exception interface. The same holds for EPCs involving methods with explicit throws clauses. Since every method in the paths from throwing sites to handling sites (except for the latter) must throw every exception type informed in the propagate statement, there are no conflicts among EPCs and between EPCs and existing exception interfaces of methods.

Figure 2 shows an example where C1.t(int, String) may throw exceptions of type E, and C3.h() catches exceptions of that type. In this scenario, the following definition establishes that the exception type E is part of the exception interfaces for C1.t(int, String) and C2.i().

propagate E: C1.t(int, String) -> C3.h();

Notice that intermediate call sites are contemplated implicitly by the definition – in this case, C2.i(). Hence, during software evolution, changes in the exception types of the EPC do not require changes to those methods. EPCs avoid the cascading effects that occur when changing exception interfaces based only on throws clause (discussed in Section 1).

Exception flows do not always appear in such a simple form as in the last example, however. In situations where there are two or more paths from the throwing site to the handling site, a simple EPC with only throwing and handling sites, like the example above, affects all the paths, as described in the previous section. It is also possible to be more specific about the paths of interest by explicitly indicating intermediate sites. Considering the example of Figure 3, the three definitions below are equivalent and each one specifies that C1.t(), C2.i() and C3.i() throw both E and F, while C4.i() is not affected.

propagate E, F: C1.t() -> C2.i() -> C5.h();
propagate E, F: C1.t() -> C3.i() -> C5.h();
propagate E, F: C1.t() -> C2.i() -> C3.i() -> C5.h();

So far, all the examples above used the call operator ->, which specifies a direct or indirect caller/callee relationship. Further expressiveness is provided with a direct call operator => for scenarios where the latter is insufficient, as in the case of Figure 4. In this case, to specify a channel for the path from C1.t() to C3.h() not passing through C2.i(), the following definition suffices:

propagate E: C1.t() => C3.h();

Figure 4: An exception flow with multiple paths, where C1.t() is a throwing site of a checked exception E and a direct caller, C3.h(), handles it.

Additionally, overloaded methods may be uniquely identified using * instead of the formal parameter list. If a class C2 has many overloaded methods named i, the definition below would add E to the exception interfaces of every one of them.

propagate E: C1.t() -> C2.i(*) -> C3.h();

As EPC definitions grow in number, it is expected of many of them to share methods in common, possibly diverging only on a few methods in their paths. For such cases, a combination operator | allows one to specify branches. Thus, a set of EPCs which only diverge in a few call sites can be combined as a single EPC. For example, in the context of Figure 3, the definition below encompasses the entire exception flow:

propagate E:

\[
C1.t() \rightarrow (C2.i() \lor C4.i()) \rightarrow C3.h();
\]

The | operator is a shorthand to define two EPCs that are identical except for its operands. Hence, the above snippet is equivalent to

propagate E: C1.t() -> C2.i() -> C3.h();
propagate E: C1.t() -> C4.i() -> C3.h();

Finally, EPCs can also be named to allow composition of channels. In the following example, three named channels are defined, where epc2 is defined in terms of epc1 and epc3 is defined in terms of epc1 and epc2.

epc1 = propagate E: C0.k() -> C1.h();
epc2 = propagate E: C2.t() -> epc1;
epc3 = propagate E: epc1 => epc2;

Notice that epc2 is defined for C2.t() called directly or indirectly by epc2’s throwing site. The definition of epc3 establishes that epc1’s handling site is directly called by epc2’s throwing site.

3.4 Implementation

Our current prototype compiler for EPiC-Java is implemented as an extension to OpenJDK’s javac. Compilation consists of a transformation of EPCs into regular throws clauses that the Java compiler can verify. This approach greatly simplifies the implementation and has good performance.

For each propagate definition, it’s call graph is searched on available AST information. If no matching call graph is found, a compilation error is issued. Otherwise, the exception types specified for the propagate definition are added to the AST nodes corresponding to the methods in the resulting call graph, minus the handling site. Consequently, code
is generated as if each method in the resulting call graph had these exception types in their own `throws` clause.

Because every instance method in Java is, by default, a virtual method, all method calls are potentially polymorphic calls. Furthermore, the compiler has to find the corresponding call graphs of the EPCs at compilation time with only static type information available. Consequently, the simplest EPC implementation can only find static call graphs. For example, Figure 5 shows an exception flow containing a method `i()` defined by interface `I`. Consider the following corresponding EPC:

```
propagate E: C1.t() => C3.h();
```

With only static types to reason about, a compiler is unable to derive the exception flow for this EPC in a practical way. An alternative is adopting static analysis frameworks [1, 12] which can be used to traverse the program call graph and estimates the paths that exceptions will travel at runtime. However, the main limitation of static analysis approaches is that they support discovery but not enforcement of exception paths. If developers employ such a tool to understand the exception structure of a system and later the system is modified, a large amount of effort might be spent discovering how these changes affected exception paths. At the same time, due to factors such as inheritance and polymorphism, these tools often report many false positives. Moreover, the effort of implementing an exception flow analysis that is precise and efficient is very work-intensive. The simpler approach we adopted is to extend the EPC syntax to specify polymorphic call sites, where explicit types of the hierarchy can be informed, thus helping the compiler to find the corresponding call graph. For example, an EPC for Figure 5 can be specified like the following, considering `Sub` to be a class that implements the interface `I`:

```
propagate E: C1.t() => {Sub <: I}.i() => C3.h();
```

This definition informs the compiler that every call site `i()` found whose receiver’s type is `I` is potentially a call to `Sub.i()`. Then, instead of searching the entire classpath for subtypes of `I` in order to find the associated call graph, the search space is restricted to `I` and `Sub`, enabling a practical and simple implementation of EPC specifications (for convenience, a list of subtypes for a given super type can be specified separated by commas). We emphasize that this is an implementation choice and not a limitation of the EPC approach. A limitation of this implementation solution is that EPCs involving anonymous classes are unsupported.

### 3.5 A Reengineering Tool

In the course of our research, we have developed a prototype tool to generate EPCs for a given Java application. It analyzes an existing program and replaces `throws` clauses of methods in propagation paths with EPC definitions.

Our tool works by removing every `throws` clause found in the source files. It uses the EPiC-Java compiler to gather checked exception errors, which contains call site information and exception types involved. As checked exception errors occur, EPCs are generated to satisfy them (using the `direct call operator: =>`), and these are supplied to the compiler. As further errors are encountered, EPCs grow in size and number, until all checked-exception errors are eliminated and the compiler is able to build the project successfully. A complete list of EPCs for the project is provided at the end, containing explicit exception flows of the software. Also, information about the number of checked-exceptions necessary to complete the compilation is provided. Generation of the EPCs requires manual intervention, however, when anonymous classes are present.

Finally, because the tool removes every `throws` clause found in the source code and uses the EPiC-Java compiler to re-generate exception interface information so as to satisfy compilation, the resulting set of exception interfaces may be smaller than the original set due to: (i) `throws` clause containing only runtime exception types; (ii) abstract methods with `throws` clause but no respective caller; and (iii) spurious `throws` declarations in methods for which no throw site is detected by the compiler for the respective checked exception types.

### 4. EVALUATION

In this section, we present a preliminary analysis of three open-source projects – namely, FreeMind 0.9.0, jEdit 4.2 and Mozilla Rhino 6R6 – whose source code were subject to reengineering to use EPC-style exception interfaces. Our goal with this study is twofold: (i) to assess whether EPCs can reduce the scattering of exception interface information; (ii) to assess if EPCs can save work for developers responsible for modifying the exception interface of a method that is called by other methods.

The reengineering of the projects involved stripping all `throws` clause existing in source files and running our generator. With the resulting EPC list, we identify channels defined solely by a throwing site as candidates for using the traditional `throws` clause. The remaining channels are manually refactored to use the `call operator` eliminating the explicit mentions of intermediate sites. We then attempt to combine similar EPCs by employing the `I` operator. Furthermore, channels with the same path but distinct exception types become a single channel with the corresponding list of exception types. We emphasize the refactoring of EPCs where manually performed, so the results reported here are conservative estimates.

Since the reengineering tool does not generate EPCs for the methods that originally have a `throws` clause that is not required for successful compilation (as noted in Section 4), we have also counted the number of `required` checked exception methods: the minimum set of methods that may throw checked exceptions.

Table 1 summarizes the collected data. For each project, we counted the total number of original `throws` clause found in the source code, the number of `required` checked exception

---

1 We used source code from the /freemind, /org/jgit and /src/org subdirectories of each project, respectively.
methods for successful compilation, the number of EPCs we created (vertical interfaces) and how many methods should still use throws clause in their signature (horizontal interfaces).

From the data collected, the resulting number of EPCs is considerably smaller, amounting to 36% of the number of removed throws clauses in the worst case. As a consequence, the specification of exception interfaces became less scattered in these systems. On the one hand, this stems from the localization of information about exception propagation that EPCs promote. On the other hand, it is a consequence of not being necessary to indicate every method that throws exceptions in the definitions of EPCs. This result suggests that the proposed approach can alleviate some of the maintenance issues of using only throws clauses while keeping the static checking that the Java compiler performs.

Finally, we observed FreeMind and JEdit build time increasing up to 10%. Rhino’s build, on the other hand, was slower by a factor of 8. We believe this is related to the use of indirect call operator \( \rightarrow \) associated with certain call graph patterns.

5. CONCLUDING REMARKS

To the best of our knowledge, this is the first work to explicitly acknowledge the need for a bi-dimensional approach to exception interface specification. We consider this to be a contribution of our work. In addition, we propose a model for the definition of vertical exception interfaces and an extension of the Java programming language named EPiC-Java that implements this model. Such model and language extensions can be implemented in Java-like languages with similar exception handling mechanisms. We have also developed a prototype compiler and reengineering tool for EPiC-Java. Finally, we report on our results in reengineering three applications to use the proposed approach.

So far, our current definition of EPC does not accommodate parametric types. Given that Java’s support for generic programming enables parametric types in the exception interface of methods, exception flows may contain many concrete exception types associated with a single parametric exception type, offering challenges to define exception channels in a simple form. Moreover, work remains to formally define our language extension. Furthermore, given that it is not always clear what methods are being affected by a propagate declaration, tooling support to aid the visualization of exception flows defined by EPCs becomes important. Finally, we intend to evaluate the proposed approach from a usability perspective.

Acknowledgements. We would like to thank the anonymous referees, who helped to improve this paper. Thiago is supported by CAPES/Brazil. Fernando is supported by CNPq (308383/2008-7 and 475157/2010-9), FACEPE (APQ-0395-1.03/10), and by INES (CNPq 573964/2008-4 and FACEPE APQ-1037-1.03/08).

6. REFERENCES