

Visualization of Exception Handling Constructs to Support Program Understanding*

Hina Shah[†]
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.

Carsten Görg[‡]
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.

Mary Jean Harrold[§]
College of Computing
Georgia Institute of Technology
Atlanta, Georgia, U.S.A.

Abstract

This paper presents a new visualization technique for supporting the understanding of exception-handling constructs in Java programs. To understand the requirements for such a visualization, we surveyed a group of software developers, and used the results of that survey to guide the creation of the visualizations. The technique presents the exception-handling information using three views: the quantitative view, the flow view, and the contextual view. The quantitative view provides a high-level view that shows the throw-catch interactions in the program, along with relative numbers of these interactions, at the package level, the class level, and the method level. The flow view shows the type-throw-catch interactions, illustrating information such as which exception types reach particular throw statements, which catch statements handle particular throw statements, and which throw statements are not caught in the program. The contextual view shows, for particular type-throw-catch interactions, the packages, classes, and methods that contribute to that exception-handling construct. The paper also presents a case study in which we evaluated a prototype of the visualization system on a small set of developers.

Keywords: Exception handling, interactive visualization, multiple views, program understanding, Eclipse plugin.

1 Introduction

Object-oriented programming languages, such as Java or C#, provide native constructs for handling exceptions that occur during a program's execution. These constructs specify mechanisms to define exceptions, to raise exceptions, to address exceptions by executing designated code, and to return to the regular control flow of the program after an exception is raised. Studies on Java programs show that developers make frequent use of these exception-handling constructs [Sinha and Harrold 1998] and that the mechanisms to handle an exception are not applied locally (within a method) but they are scattered across different methods, classes, or even packages [Ryder et al. 2000].

Despite the native support of programming languages, exception-handling constructs and their behaviors at runtime are often the least understood parts of a program [Schaefer and Bundy 1993]. This problem exists for two main reasons: first, exception handling intro-

duces implicit control flow, and second, features of exception handling can be abused or misused by the developers. Implicit control flow introduces additional complexity to object-oriented programs, and thus, increases the probability that developers may overlook important interactions in the program [Sinha et al. 2004]. Abuses or misuses of exception-handling features can lead to code that is difficult to understand, verify, and maintain, or even to faulty code [Reimer and Srinivasan 2003].

Researchers have developed different approaches for alleviating the problems introduced by exception-handling constructs. Robillard and Murphy [2000] proposed a design approach for simplifying the exception structure in Java programs, Sinha and colleagues developed analysis and testing techniques [Sinha and Harrold 2000], and presented an approach that automates the support for development, maintenance, and testing of programs with exception-handling constructs [Sinha et al. 2004]. Fu and Ryder introduced exception-chain analysis to reveal the architecture of exception handling in Java server applications [Fu and Ryder 2007].

Researchers have also created visualization tools to help developers better understand analysis results related to exception-handling constructs. JEX [Robillard and Murphy 1999; Robillard and Murphy 2003] analyzes the flow of exceptions and JEXVIS¹ provides a visualization of the exception structure (the exception types that might arise and the handlers that are present) as a flow graph. EXTEST [Fu and Ryder 2005a; Fu and Ryder 2005b] shows the handlers for exceptions, their triggers, and their propagation paths as tree views, and supports navigation of exception-handling code. EXPO [Sinha et al. 2004] computes exception-handling statistics and visualizes the context of throw-catch pairs as a flow graph. Both, EXPO and EXTEST are plugins for the Eclipse IDE. Finally, EXCEPTIONBROWSER [Chang et al. 2002] visualizes the propagation of exceptions as a tree.

All these tools mainly focus on the analysis of exception-handling constructs and provide only basic visualizations in the form of lists, trees, or flow graphs. The visualizations are at a low-level of abstraction, do not present a system-wide overview, and do not provide sufficient context. Thus, using these tools, it is still difficult for developers to understand exception-handling constructs in their context within the program and not just locally.

Understanding the complex mechanisms of exception handling in a large software system is key for efficiently maintaining, testing, and debugging the system. Vessey [1985] conducted an observational study of programmers performing debugging tasks. She found that experts tend to use well-considered strategies to generate hypotheses and to validate them. These experts apply a breadth-first approach that assures that they first gain a high-level understanding, and then try to verify or refute the hypotheses in the context of the global situation. Novice programmers, in contrast, tend to apply a depth-first approach, verifying hypotheses one after another as they

*Patent pending, Georgia Institute of Technology

[†]e-mail: hinashah@cc.gatech.edu

[‡]e-mail: goerg@cc.gatech.edu

[§]e-mail: harrold@cc.gatech.edu

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

¹A. Sinha. JexVis: An interactive visualization tool for exception call graphs in Java. Unpublished report, <http://cs.ubc.ca/~tmm/courses/cpsc533c-05-fall/projects/anirbans/report.pdf>

are formed without developing a high-level understanding. These programmers failed more often in their debugging tasks.

Gaining a high-level understanding of a program, on the one hand, and validating hypotheses taking all low-level details into account, on the other hand, are important for successfully maintaining and developing programs—especially if the programs contain implicit control flow generated by exception-handling constructs. We believe, and our studies suggest, that interactive visualizations are well suited for bridging the different levels of program understanding and for supporting developers with both high-level and low-level tasks. Showing complex interactions among exception-handling constructs helps developers to gain and to maintain a high-level understanding, whereas communicating analysis results provides the low-level details necessary for writing and editing source code.

However, to better understand the actual needs related to exception-handling constructs of Java developers and to inform the design of our visualization system, we conducted a survey. The 34 participants consist of developers from industry and students pursuing a Ph.D. in the area of software engineering. The survey collected information about what the participants consider useful in understanding exception-handling constructs. We used the results to create our visualization system.

Our new visualization technique features three different views. A high-level view presents information about throw-catch pairs at the level of packages, classes, or methods. This view is useful for answering questions such as ‘Which packages are involved in exception handling?’ or ‘What is the flow of exceptions across classes?’. A second view provides information about the different mechanisms of selected exceptions flows. Type definitions, throw, catch, and finally clauses are visualized as abstract icons on separate layers, and the flow is represented as links between them. This view helps the developer to understand the structure of exception flows, and answers questions such as ‘Which different types does a catch clause handle?’ or ‘Which catch clauses can be reached from a particular throw?’. A third view provides low-level information by embedding exception-handling constructs and their flows in a type of SEESOFT view [Eick et al. 1992] of the system. This view lets developers examine questions such as ‘How does a program recover from an exception (i.e. how is the exception flow integrated in the control flow)?’ or ‘Which methods are on the propagation path of an exception?’.

The views are integrated into a plugin framework to Eclipse that we call ENHANCE (Exception HANDling CEntric visualization). ENHANCE provides multiple filter techniques and supports the search for patterns of exception-handling constructs, such as unhandled exceptions or type mismatches in throw-catch pairs. Interactive commands let the developer navigate among the different views.

The main benefit of our technique is that it presents a framework for visualizing flows at multiple levels of abstraction. Our technique is not restricted to exception flows but could, for example, also be used to visualize the flow of data. Another benefit of our technique is that it provides interactive techniques to navigate between different levels of abstraction without losing context.

The main contributions of this paper are:

- The results of a survey to understand the needs of developers related to exception-handling constructs in Java programs
- The description of a new interactive technique for visualizing exception flows in Java programs on three levels of abstraction
- The presentation of the results of a case study to obtain preliminary evaluation of our new visualization

2 Background

In this section, we provide a brief overview of exception-handling constructs in Java. We give only the detail that is needed to understand the examples and visualizations presented in this paper. The complete specification of the Java programming language is available in Reference [Gosling et al. 2005] and information about analysis techniques related to exception flow are presented in References [Fu and Ryder 2007; Robillard and Murphy 2003; Sinha et al. 2004].

There exist two different kinds of exceptions in Java: exceptions that are explicitly thrown in the code (checked exceptions) and exceptions generated by the Java virtual machine at runtime, such as out of memory exceptions (unchecked exceptions). In this paper, we consider only checked exceptions because we gather exception-related information from a static program-analysis tool. However, in the future, we plan to integrate dynamic-analysis results that will let us address unchecked exceptions at runtime.

The following program illustrates the way in which exceptions are used in a Java program that computes the factorial of an integer. The class definition of the exception and the method to read the input data are omitted because of space constraints. We use this program for our discussion of exception-handling constructs.

Program: Java program that computes the factorial and uses exception handling constructs.

```
public class Fac {
    private static int i, fac;
    public static void main(String args[]) {
1.     i = fac = 1;
2.     int n = readInt();
3.     try {
4.         while ( i <= n ) {
5.             mult();
6.             i++;
            }
        }
7.     catch ( ValueExceededException vee ) {
8.         System.out.println( "value exceeded" );
9.         return;
        }
10.    finally {
11.        System.out.println( "Program terminated." );
        }
    }
    private static void mult() throws
        ValueExceededException {
12.        if ( fac * i > MAXVAL )
13.            throw( new ValueExceededException() );
14.        fac = fac * i;
15.        System.out.println( "fac(" +i+ ")=" + fac );
    }
}
```

In Java, checked exceptions are modeled as regular objects and can be raised using the throw statement (e.g., line 13). To handle exceptions, Java provides try, catch, and finally statements. A try block (e.g., line 3-6) contains a sequence of statements and is executed until an exception is thrown or until the block is completed. A try block is followed by one or more catch blocks, by a finally block, or by both. A catch block (e.g., line 7-9) is associated with a try block, defines the type of the exception it handles, and contains a set of statements. A finally block (e.g., line 10-11) is also associated with a try block and contains a set of statements.

If an exception of type E occurs in a try block, the associated catch blocks are checked for a matching type (i.e., for type E or a superclass of E). If a matching catch block is found, its body is ex-

ecuted and the program continues its execution with the statement following the try block. Otherwise, the call stack is searched for a matching catch block. If a match is found, the program continues with the execution of that catch block's code; otherwise, the program terminates. If a finally block is present in a try-catch-finally sequence, its code is always executed: either after the try block (if no exception is raised or no matching catch block is found for a raised exception) or after the catch block (if a matching catch block is found for a raised exception).

Thrown exceptions can be deactivated by a matching catch handler or by a finally block containing a statement that transfers the control flow outside the finally block (e.g., a return or a continue statement). The flow of an exception consists of two parts: (1) the flow from the exception's type definition to reachable throw statements and (2) the flow from those throw statements to reachable catch statements. A throw statement is reachable from a type definition if an execution path exists from the type definition to the throw statement; a catch statement is reachable from a throw statement if an execution path exists from the throw statement to the catch statement and no statement along the path deactivates the raised exception.

3 Visualization

In this section, first, we describe the way in which we gathered the requirements for our visualization (Section 3.1). Then, we discuss the design decisions we made for the visualizations and the way in which these decisions were guided by the gathered requirements and our past experiences (Section 3.2). Finally, we present the three views of our visualization in detail (Section 3.3).

The three views provide different perspectives on exception flows and the related exception-handling constructs. Often, it is useful to switch between different views to address various facets of a debugging or coding task related to exception flows. To ease navigation, the user interface provides a right-click-initiated pop-up menu with commands to switch between views. The currently selected elements are used as the focus for the context switch.

3.1 Requirements

To gather general insights that developers might need to better understand exception-handling constructs, we conducted a survey among software engineers from industry and academia. We created the initial survey questionnaire based on our past experiences, and we later revised it according to the feedback we received after conducting a pilot survey with some graduate students from our research lab. Of the 34 software engineers who participated in the survey, 44% of them had more than five years of professional industry experience as software developers. The main roles of the participants at the time of the survey were software developer, project manager, test engineer, graduate student in computer science, and researcher at an institute.

The survey consists of questions concerning what information related to exception-handling constructs would be beneficial to view, and whether exception-related quantitative information, detailed contextual-flow information of exceptions, and information about the change impact of exception-handling constructs on the rest of the program would be useful. We summarize the main results of the study.

One set of questions concerned the usefulness of exception-related *quantitative* information. 70% of the participants expressed the need for viewing exception-dependency information² because it

²Structural element *A* is *exception-dependent* on structural element *B*, if an exception thrown in *A* is caught in *B*; a structural element can be any

would help them to understand cyclic dependencies, tight coupling among structural elements, exception constructs' concentration in a particular element, and structural complexity of the program with respect to exceptions. 55% thought that it would be useful to see information about the number of exceptions of a particular type within a method, a class, or a package.

Another set of questions concerned the usefulness of exception-related *contextual* information. 75% of the participants thought that visualizations showing detailed contextual information about an exception's origin, its type, and its complete propagation path would be beneficial for better understanding of the exception flow. Additionally, they thought such views would also aid in quickly understanding change-impact details (e.g., how modifying a catch block's type may affect the set of exceptions it may handle). They thought that such tasks were tedious to perform with the current set of tools, and a visualization would be helpful.

3.2 Design Decisions

The survey results clearly indicated the need for representing exception-related information at two levels of detail: a high-level representation that provides quantitative information about exception constructs with respect to overall program structure, and a low-level representation that provides minute contextual details with respect to each exception-flow in the program. Based on our past experiences, however, we realized the need for an intermediate view that provides more specific details than the high-level quantitative view but abstracts the minute contextual details of the low-level view. This approach lets the user focus only on the flow details of the exception-handling constructs in the program (type, throw, catch, and finally). Such an intermediate-level view not only facilitates concentrating on the exception-handling constructs and their flow information, but also provides a smooth mental transition from the general high-level quantitative information to the specific low-level contextual information. This concept relates to the information visualization mantra "Overview first, zoom and filter, then details-on-demand", introduced by Shneiderman [1996].

Thus, based on our experiences and results from the survey, we created three views for our visualization. The high-level view represents the exception-related quantitative information, the intermediate-level view focuses on flow information of different exception-handling constructs, and the low-level view represents the minute contextual details of each exception's flow.

3.3 Details of the Views

This section discusses the three views of our visualization: the Quantitative View (Section 3.3.1), the Flow View (Section 3.3.2), and the Contextual View (Section 3.3.3). In the following we use a version of the Java program NANOXML³ to present examples of the different views. NANOXML has approximately 2700 LOC, three packages, five classes, and 85 methods.

3.3.1 Quantitative View

The *Quantitative View* provides information about throw-catch pairs at different structural levels of a program's hierarchy (i.e., package level, class level, and method level). This view also gives an overview, in the form of a matrix, of the exception dependencies between structural elements. The *columns* in the matrix represent structural elements containing throw statements and the *rows* represent structural elements containing catch statements. Thus, a cell,

program construct, such as a package, a class, or a method.

³<http://nanoxml.sourceforge.net/orig/>

[*column-name*, *row-name*] in the matrix, represents throw-catch pairs between the two intersecting structural elements in *column-name* and *row-name*.

A circle in a cell indicates that there exists at least one throw-catch pair between the two intersecting structural elements. The visualization uses five distinct shades of blue to provide relative information of the throw-catch pair density. Our technique allocates the color using a three step process: (1) calculate the range of the number of throw-catch pairs, (2) partition this range into five discrete sets of values, and (3) assign one shade of blue to each set such that the darkness of the shade increases with the set values. Thus, a circle with the darkest shade indicates that the intersecting structural elements are strongly exception-dependent on each other.

By default, the visualization uses a static color scheme: the number of throw-catch pairs in the entire program under consideration is used to calculate the range of number of throw-catch pairs. This choice of color scheme assures that the color assignment is consistent across the different structural levels. In some cases, however, using such a static scheme could result in most cells belonging to the same set of values and making them indistinguishable. To address this problem, a dynamic color scheme can be used on demand: the number of throw-catch pairs in the currently displayed set of packages, classes, or methods is used to calculate the range of number of throw-catch pairs.

We chose a plain design for the Quantitative View and do not display additional information in the circles so that the view scales to a reasonable size. Using cells of size 15x15 pixels, the Quantitative View can display up to 50 packages on a standard screen resolution when presenting the labels for the columns vertically.

Figure 1 shows the Quantitative View for NANOXML at the package level: each of the three rows and columns in the matrix represents one package in the subject. The first package, labeled “(default)”, represents the default package in the program. The two circles at [nanoxml, nanoxml] and [nanoxml, nanoxml.sax] indicate that the package nanoxml may throw exceptions that may be caught by catch blocks either within the same package (nanoxml) or in another package (nanoxml.sax). The legend above the matrix shows the static color scheme for the program.

Moving the mouse over a cell displays a tooltip with the actual number of throw catch pairs. In the example in Figure 1, there are 11 throws in the nanoxml.XMLElement class that may be caught by catch blocks in the nanoxml.sax.SAXParser class.

The user interface provides simple operations that let the user interact with the view. A single click selects an element (multiple selections are possible using the CTRL key), and a double click switches to the next lower level while keeping the selected elements in focus (a SHIFT double click switches to the next higher level, using the up-arrow of the SHIFT key as metaphor). The user can make multiple selections by using the rubberband mechanism.

To help the user navigate between different levels, the visualization uses colors on the row and column headers of the matrix (i.e., the topmost row and the leftmost column in the matrix) according to their levels: dark orange for the package level, light orange for the class level, and cream for the method level. The selected colors belong to the same color group and take the level hierarchy into account: the higher the level is in the hierarchy, the darker is its color.

The Quantitative View helps users gain insights about the program’s implementation with respect to exceptions. For instance, with the exception-dependency information that the view provides, a user can get an overview of how well the program is implemented with

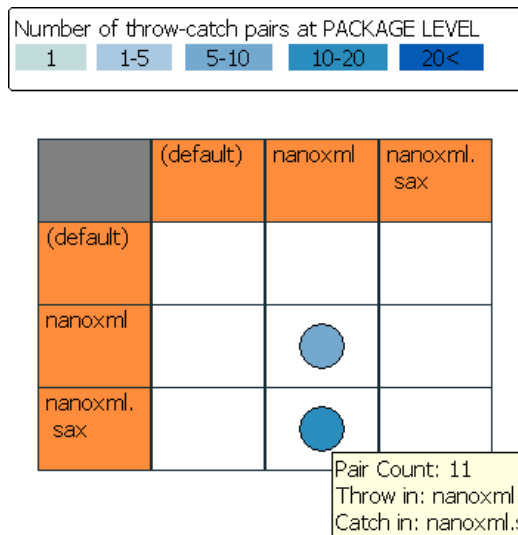


Figure 1: The Quantitative View showing exception dependencies in NANOXML at the package level.

respect to exception-handling constructs; if all circles on the package level are on the top-left to bottom-right diagonal in the matrix, the program has no cross-package dependencies in terms of exceptions.

3.3.2 Flow View

The Quantitative View displays information about the throw-catch pairs at different structural levels. However, it does not provide information about the types and flows of the exceptions. The Flow View provides further details about exception-handling constructs by showing a graph (the *exception-flow graph*) that consists of nodes representing three components of exception handling: exception types, throw statements, and catch statements.

The Flow View represents the components using different shapes: triangles for type nodes, squares for throw-statement nodes, and circles for catch-statement nodes. Circles with a white hole in the center represent empty catch handlers (i.e., catch blocks that do not contain any executable statements). An edge between a type node and a throw-statement node indicates that an exception of that type reaches that throw statement in the program. If an exception type is not explicitly defined but the throw statement throws the exception directly using its constructor, the edge between the throw node and the type node is colored gray to indicate that no explicit flow exists. An edge between a throw-statement node and a catch-statement node indicates that an exception thrown at that throw statement can reach that catch statement. All nodes are colored green except for one node that may be colored in red. The red node represents an exit node and is drawn as a red circle indicating that it is a special kind of catch-statement node; edges from throw nodes reaching this red node indicate that exception occurring at those throw statements may go uncaught and thus, reach the program’s exit.

Our technique uses a hierarchical graph layout algorithm to compute the layout for the flow graph. The graph consists of three layers and nodes are assigned to one of the three layers: all type nodes are assigned to the top layer, all throw-statement nodes to the middle layer, and all catch-statement nodes to the bottom layer. Within a layer, our technique sorts the nodes to minimize edge crossings using a heuristic [Tollis et al. 1998].

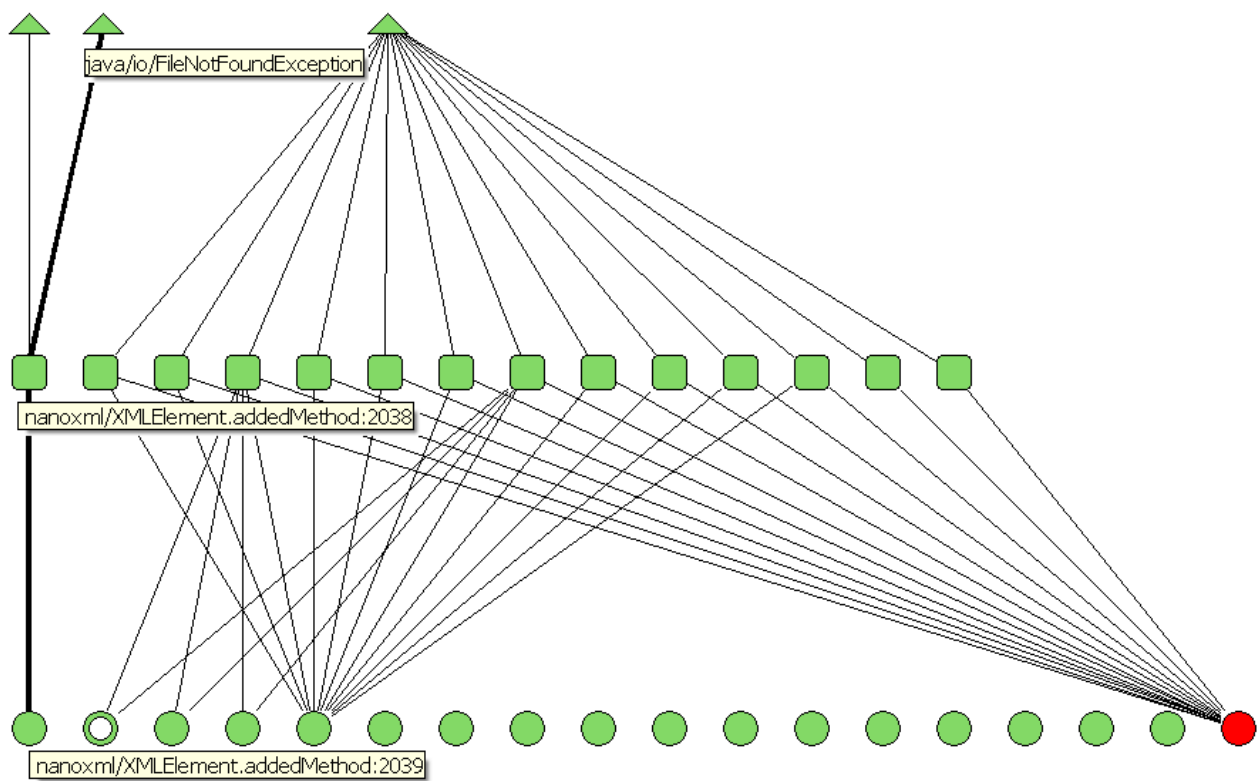


Figure 2: The Flow View showing exception flows in NANOXML.

Figure 2 shows the Flow View for NANOXML when we select both circles in the Quantitative View (Figure 1) and switch to the Flow View. The highlighted path, shown with thicker edges in the figure, along with the tooltips,⁴ show that an exception of type “FileNotFoundException” can be thrown from the throw statement at line 2038 in method nanoXML.XMLMethod.addedMethod and this exception can be caught at the catch statement at line 2039 in the same method. The figure also shows that exceptions of two types can reach the selected throw node.

Moving the mouse over a component displays further details of that component in a tooltip: the full name of the exception type for type nodes and the complete path (package name, class name, method name, and line number) for throw and catch nodes. For edges, a tooltip shows details about the two components it connects.

Nodes and edges can be selected using a single mouse click. Selecting a node highlights all exception-flow paths to which the node belongs, and selecting an edge highlights only the two adjacent nodes. Multiple selections are possible using the CTRL key.

The Flow View helps users infer information about the statements represented by the nodes. We illustrate this using two examples. First, a catch block with several incoming edges may indicate the impact of that catch block on the rest of the exception flow in the program. For example, many edges into a catch node indicate that the node represents a catch statement that is responsible for handling a number of exceptions and thus, changing such a catch block may impact different parts of the program. Second, tracing complete paths of a tuple [type,throw,catch] in the view may help to determine the type of a catch block. For example, a catch block handling different types of exceptions implies that catch block’s

type is a supertype of all the exception types it handles. In Figure 2, the subgraph headed by the two type nodes on the left shows two types of exceptions reaching the same throw statement, which then reaches a single catch block. This pattern indicates that the catch block handles two different types of exceptions. Furthermore, inspection reveals that the catch block is indeed of the type “Exception” which is handling exceptions of the type “FileNotFoundException” and “ClassNotFoundException”.

In addition, the Flow View helps in observing patterns of flow of exceptions in a program. For instance, edges from one throw-statement node to different catch-statement nodes indicate that there are different paths that an exception at that throw-statement may follow, depending on the program conditions. We define three perspectives on exception-flow graphs (each produces a subgraph of the exception-flow graph):

1. *type centric* with respect to type-definition statement s_D : the node set of this subgraph consists of s_D itself, all throw nodes that are reachable from s_D , and all catch nodes that are reachable from those throw nodes.
2. *throw centric* with respect to throw statement s_T : the node set of this subgraph consists of the set of type definition nodes that can reach s_T , s_T itself, and all catch nodes that are reachable from s_T .
3. *catch centric* with respect to catch statement s_C : the node set of this subgraph consists of the set of throw nodes that can reach s_C , all type definition nodes that can reach those throw nodes, and s_C itself.

The edge sets of these graphs are derived from the feasible control flow defined by the given node sets. The type centric perspective

⁴For better illustration, we modified the figure to show three tooltips.

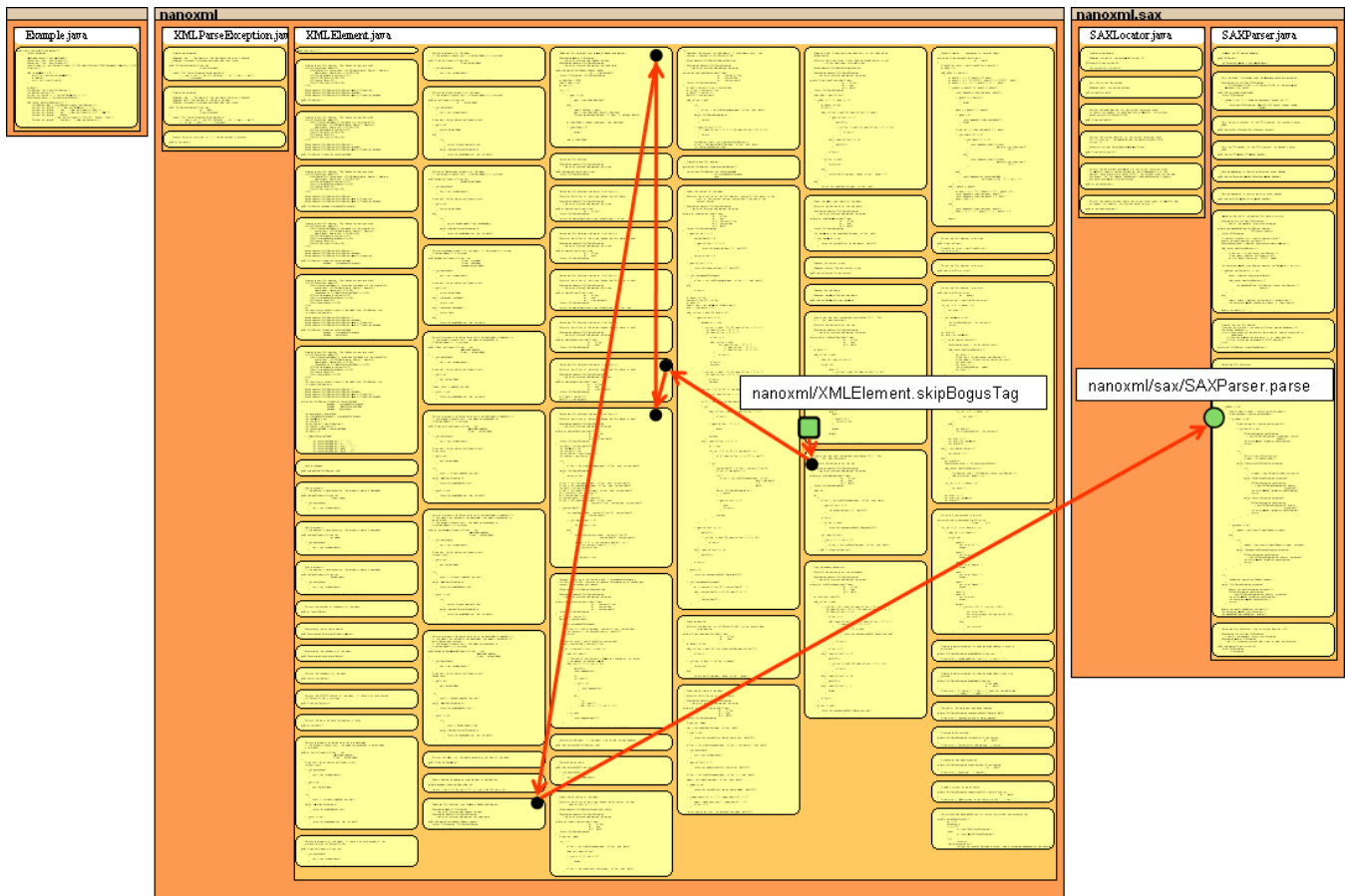


Figure 3: The Contextual View showing the exception flow of an exception embedded in the source code. The green square shows the location of the throw statement, the green circle shows the location of the catch statement, and the smaller black circles show the locations of methods along the propagation path.

leads to two patterns: *single type to single throw* and *single type to multiple throws*. The throw centric perspective leads to four patterns: *single type to single throw*, *multiple types to single throw*, *single throw to single catch*, and *single throw to multiple catch*. The catch centric perspective leads again to two patterns: *single throw to single catch* and *multiple throws to single catch*.

3.3.3 Contextual View

The Flow View displays flow information about the exceptions at the statement level with respect to throw and catch statements. However, it does not show this flow information in the presence of the statements' context with respect to the programs hierarchical structure (e.g., to which class and method a statement belongs). Also, the Flow View does not show information about the complete propagation path of an exception including the methods through which the exception may propagate before reaching the catch (i.e., methods that use the *throws* construct). The *Contextual View* provides this information by extending the exception-flow graph to show exception-propagation information (we call this the *exception-propagation graph*), and then embedding this graph into a hierarchical representation of the source code that uses the SEESOFT [Eick et al. 1992] metaphor.

The hierarchy, representing the package, class, and method levels, is composed of three rectangles: outermost dark orange rectangles represent packages, intermediate light orange rectangles represent

the classes within these packages, and innermost cream rectangles represent the methods within these classes. (This color scheme is the same as the color scheme we use to color the header row and column of the Quantitative View.) Within the method rectangle, the visualization displays the method's code in a small font using the SEESOFT metaphor. Although the code is not readable, the preserved line structures and indentations of the code help to quickly identify locations in the source code. Our technique ignores the code outside of method blocks, such as the variable declaration and import statements, because they do not directly relate to exception-handling constructs.

Our visualization uses a simple heuristic to recursively compute the layout of the hierarchy. The maximal height of a package is defined using the available screen real estate. Based on this maximal height, the technique computes each class's height. Methods are arranged in columns within classes and they are wrapped accordingly to the maximal height, and classes are arranged the same way in packages.

The exception-propagation graph consists of nodes and edges. Nodes are exception-related or non-exception-related. Exception-related nodes use the same color and shape representation as the Flow View: green squares represent throw statements and green circles represent catch statements. Non-exception-related nodes, represented as smaller black circles, denote the methods within the propagation path of the exception flow. Edges show the flow of the exception along its propagation path.

Figure 3 displays the Contextual View that shows the propagation path of an exception across two packages. The view shows two packages, `nanoxml` and `nanoxml.sax`, of NANOXML and their contained classes and methods. The embedded exception flow shows that a throw in the method `XMLElement.skipBogusTag` in package `nanoxml` is caught by the catch block in method `SAXParser.parse` in package `nanoxml.sax` after it is propagated through five other methods.

Moving the mouse over an element in the Contextual View displays further details of that element in a tooltip: the name and line number for nodes representing throw- and catch-statements, and the method name for nodes representing intermediate points in the propagation path.

The Contextual View can help the user understand how different parts of the program are involved in exception flows. For example, the view shows how any changes made, with respect to exceptions, to the intermediate methods involved in the exception-propagation path (e.g., removing *throws* construct and introducing a catch block), may affect the flow of the exception. The view can also help to understand the inappropriate coding pattern—“large distance between throw and catch”—discussed by Sinha and colleagues [2004]. The exception-propagation path provides the context of this large-distance pattern by showing the methods through which the exception is propagated and helps the developer to decide whether refactoring is necessary.

4 Implementation and Evaluation

To evaluate our visualization technique, we developed the ENHANCE (Exception HANDling CEntric visualization) prototype and conducted a case study. This section first presents the overview of our prototype implementation and then discusses a preliminary evaluation.

4.1 Prototype

ENHANCE implements the three views we presented in Section 3 as a plugin framework to Eclipse.⁵ ENHANCE is implemented in Java, and has more than 3000 LOC, eight packages, 51 classes, and 381 methods. We used the Eclipse plugin environment⁶ to implement the framework of our plugin and the Draw2D and Graphical Editing Framework (GEF)⁷ to implement the three views. We chose to develop a plugin instead of a stand-alone application because of the scalability and reusability benefits that plugin frameworks provide [Lintern et al. 2003]. In the current plugin design, the three views are integrated as three separate tabs in a single Eclipse view. This design does not let the user arrange the views freely on the screen and work with them in parallel. In the future, we plan to implement each view of our visualization as a separate Eclipse view to avoid this restriction.

Figure 4 shows a screenshot of Eclipse with the ENHANCE visualization. ENHANCE’s main view lets the user select one of the three visualizations (i.e., the Quantitative View, the Flow View, or the Contextual View) using tabs. The left column provides five filters for the views:

- An *Exception Type* filter that lets the user select the exception type(s) for which details will be provided in the three views.
- Three location filters—*Throw Statements*, *Catch Statements*, and *Finally Statements*—that let the user select the structural

elements to which the throw, catch, and finally statements belong. The views then show filtered information about the exception-handling constructs of the selected structural elements.

- A *Patterns* filter that lets the user select a pattern and view exception flows that form the selected pattern. The patterns represent the six flow patterns introduced in Section 3.3.2. The top row of the Patterns filter represents (from left to right) patterns *single type to single throw*, *multiple types to single throw*, and *single type to multiple throws*. The bottom row of the Patterns filter represents (from left to right) patterns *single throw to single catch*, *single throw to multiple catch*, and *multiple throw to single catch*. This filter is specific to the Flow View and is disabled when one of the other views is used.

The location filter for the finally statements does not yet provide any functionality. However, we have included the filter in the visualization framework because we plan to extend our system to show finally-related information as well. ENHANCE provides two kinds of filtering mechanisms: filtering by selecting and interacting directly with the entities in one of the three views (as we described in Section 3) or filtering by using any combination of the five provided filters. Because the three views are organized as tabs, it is possible to switch between the views while maintaining the same context defined by the filters.

Figure 4 shows the Flow View for NANOXML. The view shows that all the six patterns discussed in Section 3 exist in the program. The visualization also shows that there exists only one empty catch handler; its code is shown in the editor view located at the top of the eclipse window in the figure.

The information that our visualization presents is obtained from EPTOOLS, a program-analysis tool that gathers exception-related information for Java programs [Sinha and Harrold 2000]. However, the visualization is not restricted to any particular analysis tool; it can be extended easily to use other analysis tools that provide the required information.

4.2 Case Study

To evaluate our visualization, we conducted interviews with three graduate students from the software-engineering group at the Georgia Institute of Technology. Each participant has between one and four years of software industry experience in Java development. Each interview lasted for approximately one hour. The semi-structured interview guide was driven by the results obtained from our initial survey.

Whereas the initial survey (described in Section 3.1) focused on *what* information would support developers’ understanding of exception-handling constructs, this interview focused on *how* developers understand exception-handling constructs and whether ENHANCE can help them to better understand exception-handling constructs. Throughout the interviewing process we applied the think-aloud approach [van Someren et al. 1994]. The interviews consisted of three main parts:

1. In the first part of the interview, we asked general questions about approaches that participants currently follow in performing tasks related to understanding programs that use exception-handling constructs. The participants mentioned the two following common approaches: manually inspecting the exception flow (which they classified to be very tedious) and ignoring the exception-related code if it does not form a part of the core functionality.

⁵<http://www.eclipse.org/>

⁶<http://www.eclipse.org/pde/>

⁷<http://www.eclipse.org/gef/>

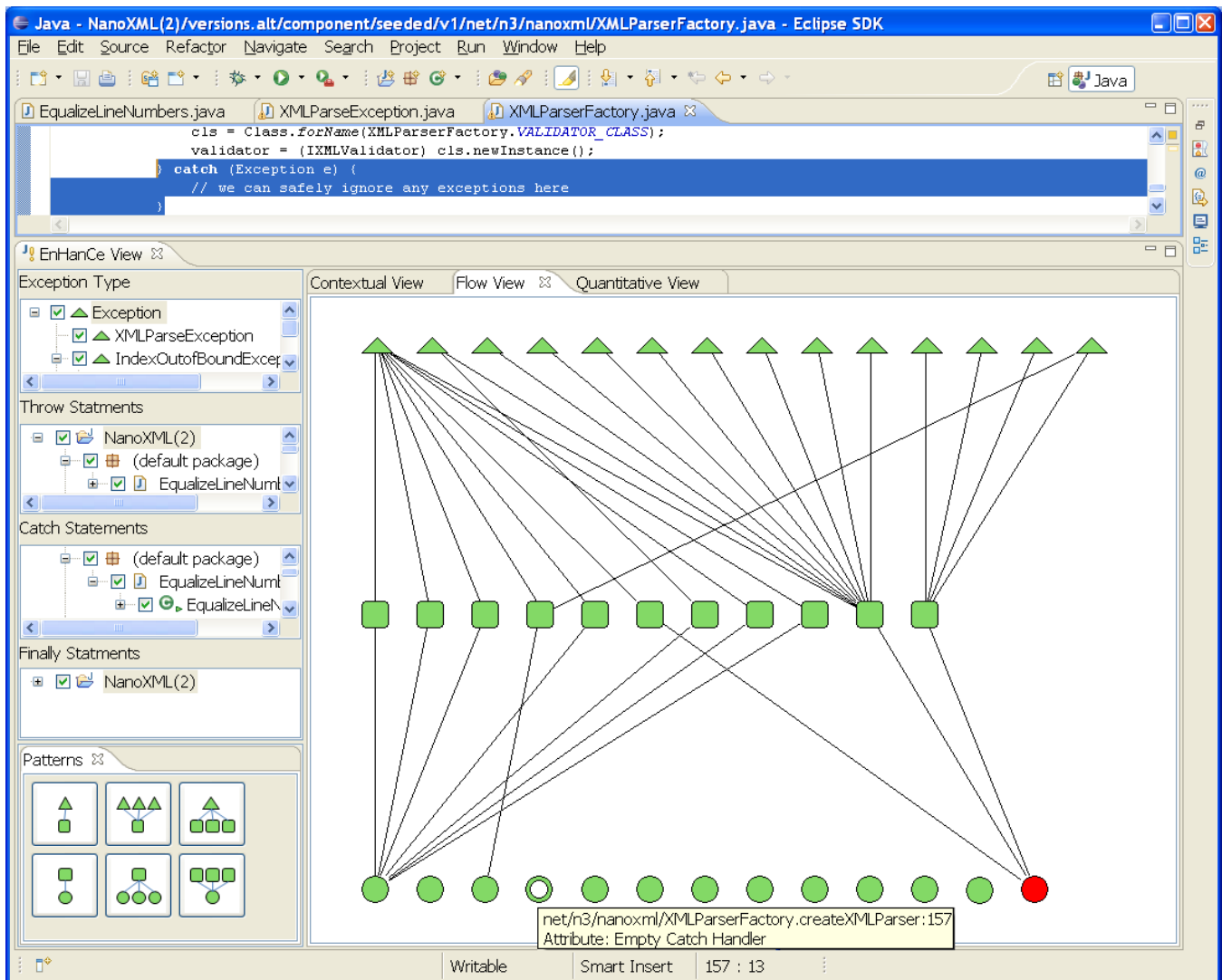


Figure 4: A screenshot of ENHANCE in the Eclipse environment shows the Flow View for NANOXML.

Then, we asked the participants to explain how they would solve three tasks: finding dependencies between structural elements, changing exception-related code and determining the change impact on the rest of the program, and understanding the entire propagation path of an exception (from its throw to its catch). The participants mentioned interesting search strategies and ignore-for-now approaches where they would change the code in the desired manner and leave the consequences for later.

2. In the second part of the interview, we first demonstrated how to use the ENHANCE plugin within the Eclipse environment. We introduced each view and then let the participants explore and play around with the views. Then, for each view, we asked how it could be used to support the comprehension of exception flow related problems.

For the Quantitative View, participants responded that they would use such a view to inspect dependencies for the purpose of testing, debugging, and identifying bad coding patterns. For example, if all circles are in one row, it shows that one module catches all exceptions, indicating a bad coding

pattern that requires refactoring. If the circles are mostly on the diagonal, it shows that the system is well designed because the exception flow is not spread throughout the program but is contained locally.

For the Flow View, participants stated they would use the view to find thrown exceptions that are not caught but that exit the program, to understand impacts related to change of the exception flow, and to find unreachable catch blocks. In addition, they thought this view could help in debugging and refactoring tasks, such as removing unreachable catch blocks.

For the Contextual View, participants responded that the view would be useful for debugging tasks, refactoring tasks, and understanding in detail how changes in exception-handling constructs may affect many other parts of the program. However, two participants had the impression that this view could be very detailed and may provide more information than required, and hence, might be better for dynamic analysis purposes, such as viewing the stack trace.

We then revisited the previous tasks discussed in the first part of the interview and asked the participants to explain how they

would perform these tasks using ENHANCE. All participants agreed that the visualizations made it simple and easy to perform these tasks.

3. In the last part of the interview, we asked the participants to give general comments about the tool, its layout, and its interface and interactions. We received valuable suggestions and remarks. We plan to address these suggestions as part of our future work.

Participants suggested that we change the design from reading the input directly from EPTOOLS to reading the input from an XML file so that the visualizations can be reused for showing error flows in other languages. They also suggested the incorporation of a legend to assist in using the Flow View.

Participants made the following remarks: (1) two participants said they found the Flow View to be very useful, (2) one participant addressed the concern of potential scalability problems, especially with the Contextual View, and (3) all three participants agreed that the tool directs developers toward better programming practices with respect to exception-handling constructs.

The evaluation study shows that some developers tend to ignore exception-handling constructs instead of trying to understand them if they do not have a tool available that supports their attempt in understanding. There are two reasons for this behavior: the constructs are either too complex to be understood or the developers assume that the constructs do not form the main functionality of the program and hence are less important. Developers can be tempted to adopt ignore-for-now approaches. However, developers can use visualization tools such as ENHANCE, if available, to comprehend how their code is structured with respect to exceptions, and thus they can understand the program from the exception-flow perspective.

5 Related Work

There exist a number of research systems that use visualization to support programmers' understanding of exception-handling constructs.

Chang and colleagues were the first to develop a tool to visualize exception-propagation paths. The EXCEPTIONBROWSER [Chang et al. 2002] is integrated in the Jipe environment⁸ and its analysis is built on top of the Barat framework.⁹ The EXCEPTIONBROWSER lets the programmer select a method and then displays a list of uncaught exceptions. Selecting one of those exceptions displays its propagation path as an interactive tree.

JEX [Robillard and Murphy 1999; Robillard and Murphy 2003] is a stand-alone tool that analyzes the flow of exceptions. Because it is not integrated into an environment, it requires more work from the user (e.g., generating a configuration file). The visualizer lets the user select classes of interest and displays the exception flow between their methods and methods from other classes that may cause exceptions to flow into the selected classes. The JEXVIS tool¹ is built on JEX and provides a visualization of the exception structure (exception types that might arise and the handlers that are present) as a flow graph. JEXVIS offers interactive techniques such as panning and zooming to navigate the graph. The tool also highlights selected paths and uses a color coding scheme to distinguish different types of nodes in the graph.

⁸<http://jipe.sourceforge.net/>

⁹<http://sourceforge.net/projects/barat>

The visualization tool EXPO [Sinha et al. 2004] is a plugin for the Eclipse environment and uses EPTOOLS [Sinha and Harrold 2000] to compute information about exception flows. EXPO shows statistical data of exception flows, such as the total number of throw-catch pairs and their minimal and maximal distance, provides information about reaching and reachable types/statements for catch handlers and throw statements respectively, and visualizes the context of throw-catch pairs as a flow graph.

EXTTEST [Fu and Ryder 2005a; Fu and Ryder 2005b] is also a plugin for the Eclipse environment and uses the Soot Java Analysis and Transformation Framework [Vallée-Rai et al. 1999] to compute *exception-catch (e-c) links*¹⁰ and their corresponding exception-flow paths. The plugin provides two tree views to browse the e-c links: the Handlers view lets the user browse the links grouped by try-catch blocks and the Triggers view lets the user browse the links grouped by the fault-sensitive operations. Using an annotated call graph, the tool computes all feasible paths for an e-c link and lets the user explore those paths step-by-step.

ENHANCE differs from all these tools: it provides system-wide quantitative information about exception-flow dependencies across different structural levels (packages, classes, and methods), generates context for exceptions flows by embedding them in a SEESOFT-like [Eick et al. 1992] view, and it uses richer visualization techniques to give perspectives on exception flow from different abstract levels.

6 Conclusions and Future Work

Software engineers are faced with the challenging task of developing, debugging, and maintaining software systems that contain exception-handling constructs. Understanding the complex mechanisms of exception handling in a large software system is key for efficiently maintaining, testing, and debugging the system. However, the implicit-flow nature of exceptions makes it difficult to understand the flow of exceptions in a program. In this paper, we address the problem of understanding exception-related information by presenting a visualization technique that shows information related to exception-handling constructs.

To gather deeper insights into the problems developers face while working with exception-related code, we surveyed 34 software engineers. The survey results and our experiences guided the design of our visualization system, which we implemented as a tool called ENHANCE (ExceptionN HANDling CEntric visualization). ENHANCE implements the three views, which provide information about exception-handling constructs and exceptions' flow from the quantitative, the flow, and the contextual perspectives. We performed a preliminary evaluation of ENHANCE by conducting interviews with three graduate students working in the area of software engineering. The results of the evaluation are promising.

Whereas ENHANCE provides a number of unique capabilities that we believe will be useful for software engineers, our work is just the start of what is possible in the area of supporting the understanding of exception-handling constructs using visualizations. Numerous avenues of research and extensions to our tool are possible in future work.

We conducted studies for performing preliminary evaluation of our system to improve its effectiveness. However, a rigorous evaluation of the system is required. We plan to perform additional user studies

¹⁰Given a set P of fault-sensitive operations that may produce exceptions, and a set C of catch blocks in a program, there is an *e-c link* (p,c) between $p \in P$ and $c \in C$ if p may trigger c .

to gain deeper insights into the system's usability and usefulness, and to evaluate its impact on debugging and maintenance tasks.

ENHANCE directly accesses EPTOOLS using its API to gather the analysis results of the exception flow in a program under consideration. We plan to generalize this approach by defining a standard format using XML to represent the analysis results so that ENHANCE becomes independent of any kind of analysis tool. A standard format will facilitate extending ENHANCE to use other analysis tools for exception flows in Java programs, such as Soot [Vallée-Rai et al. 1999], and to visualize exception flows in programs written in languages other than Java.

Additionally, having a standard format facilitates the use of our visualizations for other analysis techniques. Currently, ENHANCE only uses results gathered from static analysis of exception flows but we are planning to use dynamic analysis results to create other views showing the exception behavior of a program at runtime. Then we can also address unchecked exceptions in Java programs.

ENHANCE does not yet visualize information about finally blocks. We plan to extend our views to represent finally-related information and have already begun to integrate a filter for finally statements into our framework. The Flow View will show finally blocks in a fourth layer and the Contextual View will integrate the finally blocks into the exception flow. To alleviate potential scalability problems, we will provide more filtering techniques and implement zooming and panning interfaces for the Contextual View.

We also plan to implement a semantic-based layout strategy for the Flow View that first groups the nodes in each layer according to their hierarchical information (e.g., to what method, class, or package they belong) and then minimizes edge crossings while maintaining the groups. This approach will help to visualize the spread of exception flows in a program.

Acknowledgements

This work was supported in part by awards from National Science Foundation under CCR-0205422, CCF-0429117, CCF-0541049, and CCF-0725202, and Tata Consultancy Services, Ltd. to Georgia Tech. The participants in our survey provided valuable information that improved our visualizations. The anonymous reviewers gave helpful comments on an earlier version of this paper.

References

CHANG, B.-M., JO, J.-W., AND HER, S. H. 2002. Visualization of exception propagation for Java using static analysis. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, 173–182.

EICK, S. G., STEFFEN, J. L., AND ERIC E. SUMNER, J. 1992. Seesoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering* 18, 11, 957–968.

FU, C., AND RYDER, B. G. 2005. Navigating error recovery code in Java applications. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, 40–44.

FU, C., AND RYDER, B. G. 2005. Testing and understanding error recovery code in Java applications. In *Exception Handling in Object Oriented Systems: Developing Systems that Handle Exceptions*, 15–26.

FU, C., AND RYDER, B. G. 2007. Exception-chain analysis: Revealing exception handling architecture in Java server applications. In *Proceedings of the 29th International Conference on Software Engineering*, 230–239.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2005. *Java Language Specification*. Prentice Hall.

LINTERN, R., MICHAUD, J., STOREY, M.-A., AND WU, X. 2003. Plugging-in visualization: Experiences integrating a visualization tool with Eclipse. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, 47–56.

REIMER, D., AND SRINIVASAN, H. 2003. Analyzing exception usage in large Java applications. In *Workshop on Exception Handling in Object Oriented Systems*, 10–19.

ROBILLARD, M. P., AND MURPHY, G. C. 1999. Analyzing exception flow in Java programs. In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 322–337.

ROBILLARD, M. P., AND MURPHY, G. C. 2000. Designing robust Java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2–10.

ROBILLARD, M. P., AND MURPHY, G. C. 2003. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology* 12, 2, 191–221.

RYDER, B. G., SMITH, D., KREMER, U., GORDON, M., AND SHAH, N. 2000. A static study of Java exceptions using JESP. In *Proceedings of the 9th International Conference on Compiler Construction*, 67–81.

SCHAEFER, C. F., AND BUNDY, G. N. 1993. Static analysis of exception handling in Ada. *Software - Practice and Experience* 23, 10, 1157–1174.

SHNEIDERMAN, B. 1996. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, 336–343.

SINHA, S., AND HARROLD, M. J. 1998. Analysis of programs with exception-handling constructs. In *Proceedings of the International Conference on Software Maintenance*, 348–357.

SINHA, S., AND HARROLD, M. J. 2000. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering* 26, 9, 849–871.

SINHA, S., ORSO, A., AND HARROLD, M. J. 2004. Automated support for development, maintenance, and testing in the presence of implicit control flow. In *Proceedings of the 26th International Conference on Software Engineering*, 336–345.

TOLLIS, I. G., BATTISTA, G. D., EADES, P., AND TAMASSIA, R. 1998. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall.

VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., AND SUNDARESAN, V. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, 125–135.

VAN SOMEREN, M. W., BARNARD, Y. F., AND SANDBERG, J. A. C. 1994. *The Think Aloud Method: a Practical Guide to Modelling Cognitive Processes*. Academic Press, London, San Diego.

VESSEY, I. 1985. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies* 23, 5, 459–494.