Visualization of Exception Handling Constructs to Support Program Understanding

Hina Shah, Carsten Görg, and Mary Jean Harrold

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. We implemented our technique in an Eclipse plugin called EnHanCe and conducted a usability and utility study with participants in industry.

Index Terms

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

◆

1 INTRODUCTION

Object-oriented programming languages, such as Java and 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 [1] and that the mechanisms to handle an exception are not applied locally (within a method) but they are scattered across different methods, classes, and packages [2].

Despite the native support of programming languages, exception-handling constructs and their behaviors at runtime are often the least understood parts of a program [3], [4]. This problem exists for two main reasons: first, exception handling introduces implicit control flow, and second, features of exception handling can be abused or misused by the developers. Implicit control flow introduces additional complexity into object-oriented programs, and thus, increases the probability that developers may overlook important interactions in the program [5]. Abuses or misuses of exception-handling features can lead to code that is difficult to understand, verify, and maintain, or even to faulty code [6].

Researchers have developed different approaches for alleviating the problems introduced by exception-handling constructs. Robillard and Murphy [7] propose a design approach for simplifying the exception structure in Java programs. Sinha and colleagues developed analysis and testing techniques [8], and presented an approach that automates the support for development, maintenance, and testing of programs with exception-handling constructs [5]. Fu and Ryder introduced exception-chain analysis to reveal the architecture of exception handling in Java server applications [9].

Researchers have also created visualization tools to help developers better understand analysis results related to exception-handling constructs. JEX [10], [11] analyzes the flow of exceptions and JEXVIS[1] provides a visualization

---

- *H. Shah, C. Görg, and M.J. Harrold are with the College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332.*
  E-mail: {hinashah,goerg,harrold}@cc.gatech.edu
  An earlier version of this paper appeared in ACM Symposium on Software Visualization, September 16-17 2008, Herrsching am Ammersee, Germany.

[1] A. 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

of the exception structure (the exception types that might arise and the handlers that are present) as a flow graph. ExTest [12], [13] shows the handlers for exceptions, their triggers, and their propagation paths as tree views, and supports navigation of exception-handling code. ExPo [5] computes exception-handling statistics and visualizes the context of throw-catch pairs as a flow graph. Both ExTest and ExPo are plugins for the Eclipse IDE. Finally, ExceptionBrowser [14] 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 [4]. Understanding the complex mechanisms of exception handling in a large software system is key for efficiently maintaining, testing, and debugging the system. Vessey [15] 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 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.

To develop our interactive visualization systems, we used an iterative approach that involved developers in identifying requirements for the system and in evaluating several stages of the creation of the system. In our first step, to better understand the needs of Java developers related to exception-handling constructs and to inform the design of our visualization system, we conducted a survey. The 34 participants in the survey 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. The results of the survey provided us with a set of requirements for the system. In our second step, we used these requirements to create our interactive visualization system for Java exception-handling constructs. Using this visualization system, we conducted a study with 12 participants to evaluate our interactive visualization system. We updated our visualization system based on the comments of the participants. The result is the interactive visualization we present in this paper.

Our new interactive visualization features three different views. A high-level view presents information about throw-catch pairs at the level of packages, classes, and 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 SeeSoft-type view [16] 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. Because we gather exception-related information from a static program-analysis tool, in this paper, we do not consider exceptions generated by the Java virtual machine at runtime. However, in the future, we plan to integrate runtime results that will let our tool address such exceptions.

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 results of a study to evaluate our new visualization

In the next section, we provide background about exception-handling constructs in Java. In Section 3, we discuss the results of the survey as well as design alternatives, and we present our new interactive visualization. In Section 4, we describe the implementation of our interactive visualization. In Section 5, we present our study to evaluate the visualization, and discuss the results of the evaluation. In Section 6, we present related work and in Section 7, we conclude and discuss future work.

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 [17], and information about analysis techniques related to exception flow are presented in References [5], [9], [11].

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). As we stated in the Introduction (Section 1), because we gather exception-related information using a static program-analysis tool, in this paper, we consider only checked exceptions.

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 considerations. 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., lines 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., lines 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., lines 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 executed 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 we used both the requirements we gathered and our past experience to guide these decisions (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 insights about what developers need for better understanding of exception-handling constructs, we conducted a survey among software engineers from industry and academia. We created the initial survey questionnaire based on our past experience, and we later revised it according to the feedback we received from a pilot survey with 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, researcher at an institute, and graduate student in computer science.

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; Appendix A provides the complete Survey Questionnaire. One set of questions concern the usefulness of exception-related *quantitative* information. 70% of the participants expressed the need for viewing exception-dependency information[2] because it 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 concern 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 help 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

---

[2] Structural 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 program construct, such as a package, a class, or a method.

program structure, and a low-level representation that provides contextual details with respect to each exception-flow in the program. Based on our past experience, however, we realized the need for an intermediate view that provides more specific details than the high-level quantitative view but abstracts the 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 follows the information visualization mantra "Overview first, zoom and filter, then details-on-demand," introduced by Shneiderman [18].

Thus, based on our experience and the results of 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 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 discussion, we use a version of the Java program NanoXML[3] 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 *rows* in the matrix represent structural elements containing throw statements and the *columns* represent structural elements containing catch statements. Thus, a cell, [*row-name*, *column-name*] in the matrix, represents throw-catch pairs between the two intersecting structural elements in *row-name* and *column-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) it calculates the range of the number of throw-catch pairs, (2) it partitions this range into five sets of values, and (3) it assigns 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. The number in the circle indicates the actual number of throw-catch pairs between those structural elements. Using color and numbers to encode the data supports two different tasks. The colored circles let the user quickly find outliers or look for patterns, and the numbers in the circles provide the detailed information.

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 the 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.

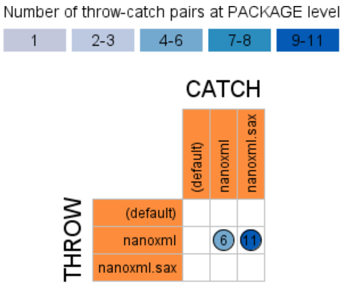
To ensure that the view scales for a reasonable-sized program, we chose a plain design for the Quantitative View and do not display additional information in the circles. Using cells of size 15x15 pixels, the Quantitative View can display up to 50 packages on a standard screen resolution when presenting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

Figure 1[4] shows the Quantitative View for NanoXML at the three different level: (a) the package level, (b) the class level, and (c) the method level. In Figure 1(a) 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 17 exceptions: 6 of them may be caught by catch blocks within the same package (nanoxml), and 11 of them may be caught in another package (nanoxml.sax). The legend above the matrix shows the static color scheme for the program. Figure 1(b) shows the Quantitative View at the class level (for all classes in the program). The view also uses the static color scheme for the program and shows again two circles that indicate that the class

---

[3] http://nanoxml.sourceforge.net/orig/

[4] In this article, we make use of color pictures. Reading the paper on-screen or as a color-printed version will help in understanding the visualization.

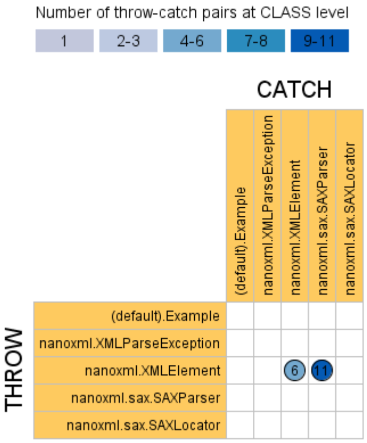


Number of throw-catch pairs at PACKAGE level

1 | 2-3 | 4-6 | 7-8 | 9-11

THROW \ CATCH	(default)	nanoxml	nanoxml.sax
(default)			
nanoxml		6	11
nanoxml.sax			

(a) *Quantitative View at the package level.*

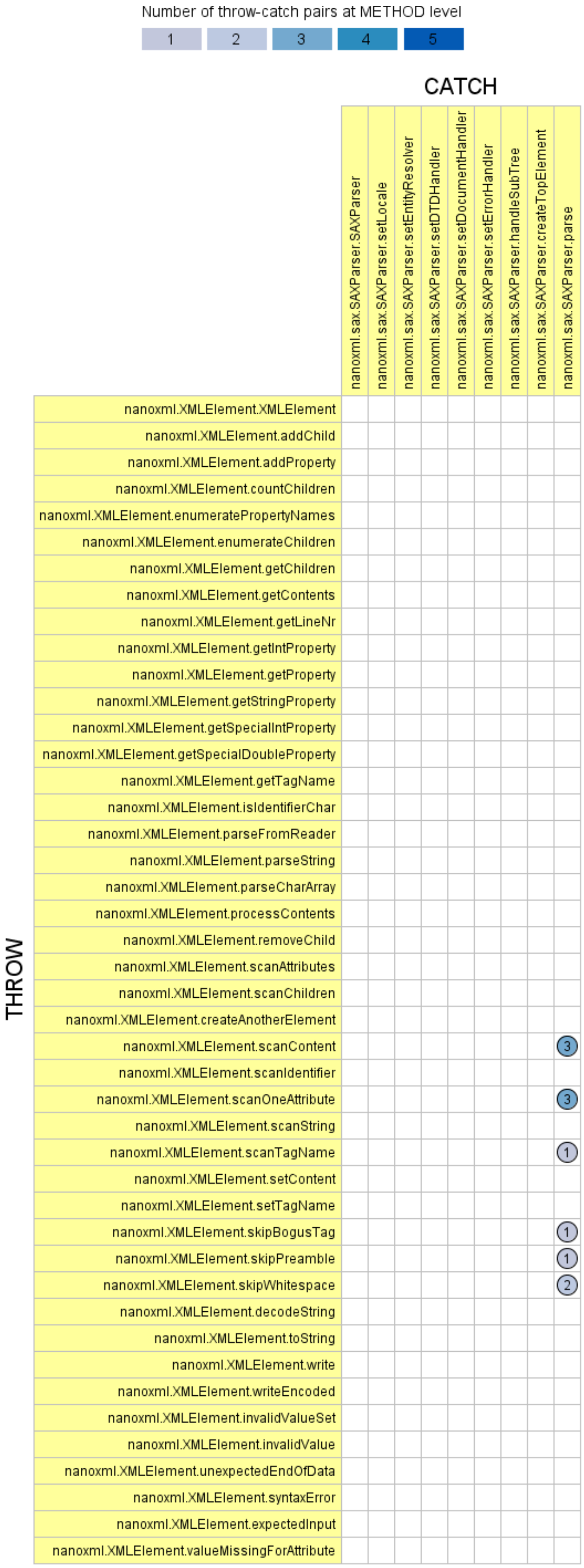


Number of throw-catch pairs at CLASS level

1 | 2-3 | 4-6 | 7-8 | 9-11

THROW \ CATCH	(default).Example	nanoxml.XMLParseException	nanoxml.XMLElement	nanoxml.sax.SAXParser	nanoxml.sax.SAXLocator
(default).Example					
nanoxml.XMLParseException					
nanoxml.XMLElement			6	11	
nanoxml.sax.SAXParser					
nanoxml.sax.SAXLocator					

(b) *Quantitative View at the class level.*



Number of throw-catch pairs at METHOD level

1 | 2 | 3 | 4 | 5

THROW \ CATCH	nanoxml.sax.SAXParser.SAXParser	nanoxml.sax.SAXParser.setLocale	nanoxml.sax.SAXParser.setEntityResolver	nanoxml.sax.SAXParser.setDTDHandler	nanoxml.sax.SAXParser.setDocumentHandler	nanoxml.sax.SAXParser.setErrorHandler	nanoxml.sax.SAXParser.handleSubTree	nanoxml.sax.SAXParser.createTopElement	nanoxml.sax.SAXParser.parse
nanoxml.XMLElement.XMLElement									
nanoxml.XMLElement.addChild									
nanoxml.XMLElement.addProperty									
nanoxml.XMLElement.countChildren									
nanoxml.XMLElement.enumeratePropertyNames									
nanoxml.XMLElement.enumerateChildren									
nanoxml.XMLElement.getChildren									
nanoxml.XMLElement.getContents									
nanoxml.XMLElement.getLineNr									
nanoxml.XMLElement.getIntProperty									
nanoxml.XMLElement.getProperty									
nanoxml.XMLElement.getStringProperty									
nanoxml.XMLElement.getSpecialIntProperty									
nanoxml.XMLElement.getSpecialDoubleProperty									
nanoxml.XMLElement.getTagName									
nanoxml.XMLElement.isIdentifierChar									
nanoxml.XMLElement.parseFromReader									
nanoxml.XMLElement.parseString									
nanoxml.XMLElement.parseCharArray									
nanoxml.XMLElement.processContents									
nanoxml.XMLElement.removeChild									
nanoxml.XMLElement.scanAttributes									
nanoxml.XMLElement.scanChildren									
nanoxml.XMLElement.createAnotherElement									
nanoxml.XMLElement.scanContent									3
nanoxml.XMLElement.scanIdentifier									
nanoxml.XMLElement.scanOneAttribute									3
nanoxml.XMLElement.scanString									
nanoxml.XMLElement.scanTagName									1
nanoxml.XMLElement.setContent									
nanoxml.XMLElement.setTagName									
nanoxml.XMLElement.skipBogusTag									1
nanoxml.XMLElement.skipPreamble									1
nanoxml.XMLElement.skipWhitespace									2
nanoxml.XMLElement.decodeString									
nanoxml.XMLElement.toString									
nanoxml.XMLElement.write									
nanoxml.XMLElement.writeEncoded									
nanoxml.XMLElement.invalidValueSet									
nanoxml.XMLElement.invalidValue									
nanoxml.XMLElement.unexpectedEndOfData									
nanoxml.XMLElement.syntaxError									
nanoxml.XMLElement.expectedInput									
nanoxml.XMLElement.valueMissingForAttribute									

(c) *Quantitative View at the method level.*

Fig. 1: *The Quantitative View showing exception dependencies in* NanoXML*: (a) shows all throw-catch pairs at the package level, (b) shows all throw-catch pairs at the class level, and (c) shows the throw-catch pairs that are thrown in class nanoxml.sax.SAXParser and caught in class nanoxml.XMLElement.*

nanoxml.XMLElement may throw 17 exceptions that may be caught by catch blocks either within the same class or in another class (nanoxml.sax.SAXParser). Figure 1(c) shows the Quantitative View at the method level for throw-catch pairs with throws in class nanoxml.XMLElement and catches in class nanoxml.sax.SAXParser (circles with number 11 in Figure 1(b)). The view uses the dynamic color scheme because there are only a few throw-catch pairs within the same methods.

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 operation.[5]

Because the matrix in the Quantitative View can be large, as seen at the method level in Figure 1(c), we provide multiple filters to address scalability. Developers can (1) filter structural elements by their names (for example, consider only elements that have the term “parse” in their names), (2) reduce the size of the matrix by showing only the rows and columns that have at least one entry, and (3) select specific elements of interest.

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 respect to exception-handling constructs: if all circles at 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 four components of exception handling: exception types, throw statements, catch statements, and finally statements. In addition, some inappropriate coding patterns [5], [19] (e.g., empty catch handlers, deactivations of exceptions in finally blocks, rethrows of exceptions in catch blocks, and exceptions that reach the program’s exit) are highlighted.

The Flow View represents the components using different shapes: triangles for type nodes, squares for throw-statement nodes, circles for catch-statement nodes, and octagons for finally-statement nodes (as an analogy to a stop sign). 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. 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 nodes that indicate an inappropriate coding pattern. Red circles with a white hole in the center represent empty catch handlers (i.e., catch blocks that do not contain any executable statements). Red octagons represent finally blocks that have at least one path that deactivates an exception (for example using a return statement). The filled red circle represents an exit node (indicating that it is a special kind of catch-statement node). Edges from throw nodes that reach this red circle indicate that the exception raised 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 exception-flow graph. If no finally statements are present in the exception flow, 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 are assigned to the middle layer, and all catch-statement nodes are assigned to the bottom layer. Within a layer, our technique sorts the nodes to minimize edge crossings using a heuristic [20].

Figure 2 shows the Flow View for NANOXML when we select both circles in the Quantitative View (Figure 1(a)) and switch to the Flow View. The highlighted path, shown with thicker edges in the figure, shows that an exception of type “nanoxml/XMLParseException” can be thrown from the throw statement at line 1709 in method nanoXML/XMLElement.skipWhiteSpace and this exception can be caught at the catch statement at line 1155 in method nanoxml/XMLElement.scanChildren.

In this view, users can select nodes or edges 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.

---

[5] The *rubberband* operation selects multiple items using the mouse to press and drag a “rubber band.”

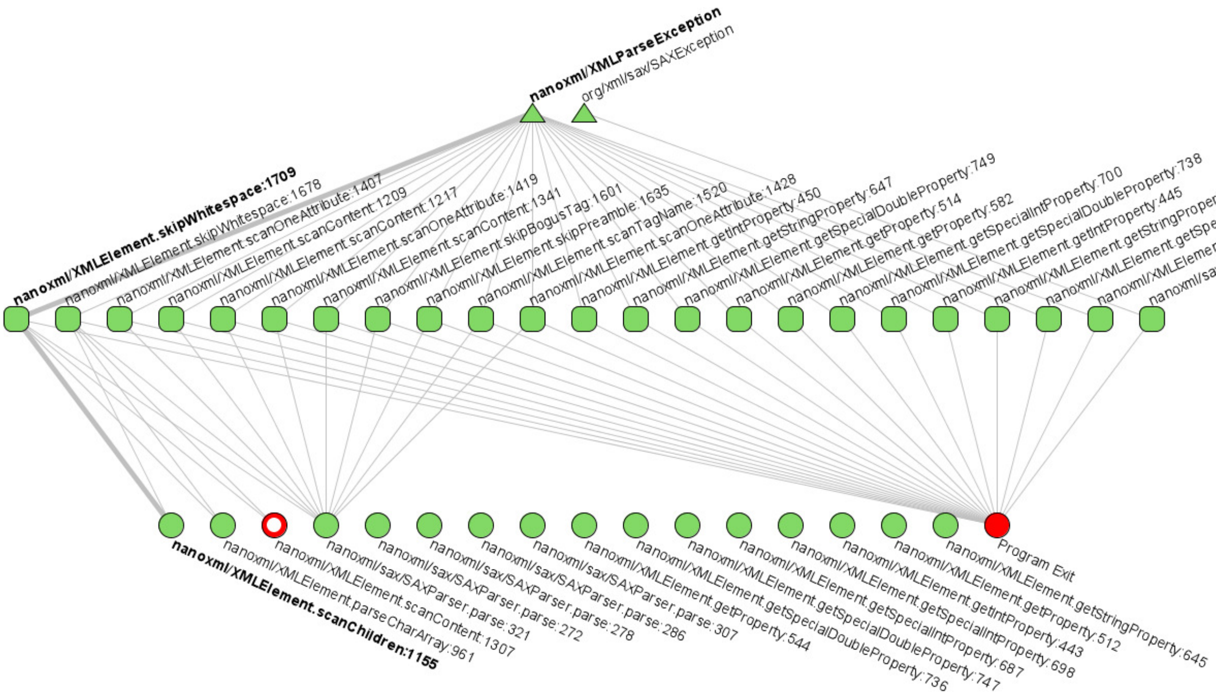

nanoxml/XMLParseException
org/xml/sax/SAXException
nanoxml/XMLElement.skipWhitespace:1709
nanoxml/XMLElement.skipWhitespace:1678
nanoxml/XMLElement.scanOneAttribute:1407
nanoxml/XMLElement.scanContent:1209
nanoxml/XMLElement.scanContent:1217
nanoxml/XMLElement.scanOneAttribute:1419
nanoxml/XMLElement.scanContent:1341
nanoxml/XMLElement.skipBogusTag:1601
nanoxml/XMLElement.skipPreamble:1635
nanoxml/XMLElement.scanTagName:1520
nanoxml/XMLElement.scanOneAttribute:1428
nanoxml/XMLElement.getIntProperty:450
nanoxml/XMLElement.getStringProperty:647
nanoxml/XMLElement.getSpecialDoubleProperty:749
nanoxml/XMLElement.getProperty:514
nanoxml/XMLElement.getProperty:582
nanoxml/XMLElement.getSpecialIntProperty:700
nanoxml/XMLElement.getSpecialDoubleProperty:738
nanoxml/XMLElement.getIntProperty:445
nanoxml/XMLElement.getStringProperty:652
nanoxml/XMLElement.getSpecialIntProperty:689
nanoxml/XMLElement.getProperty:546
nanoxml/sax/SAXParser.setLocale:127
nanoxml/XMLElement.scanChildren:1155
nanoxml/XMLElement.parseCharArray:961
nanoxml/XMLElement.scanContent:1307
nanoxml/sax/SAXParser.parse:321
nanoxml/sax/SAXParser.parse:272
nanoxml/sax/SAXParser.parse:278
nanoxml/sax/SAXParser.parse:286
nanoxml/sax/SAXParser.parse:307
nanoxml/XMLElement.getProperty:544
nanoxml/XMLElement.getSpecialDoubleProperty:736
nanoxml/XMLElement.getSpecialDoubleProperty:747
nanoxml/XMLElement.getSpecialIntProperty:687
nanoxml/XMLElement.getSpecialIntProperty:698
nanoxml/XMLElement.getIntProperty:443
nanoxml/XMLElement.getProperty:512
nanoxml/XMLElement.getStringProperty:645
Program Exit

Fig. 2: *The Flow View showing exception flows in* NanoXML. *The nodes are arranged to minimize the number of edge crossings.*

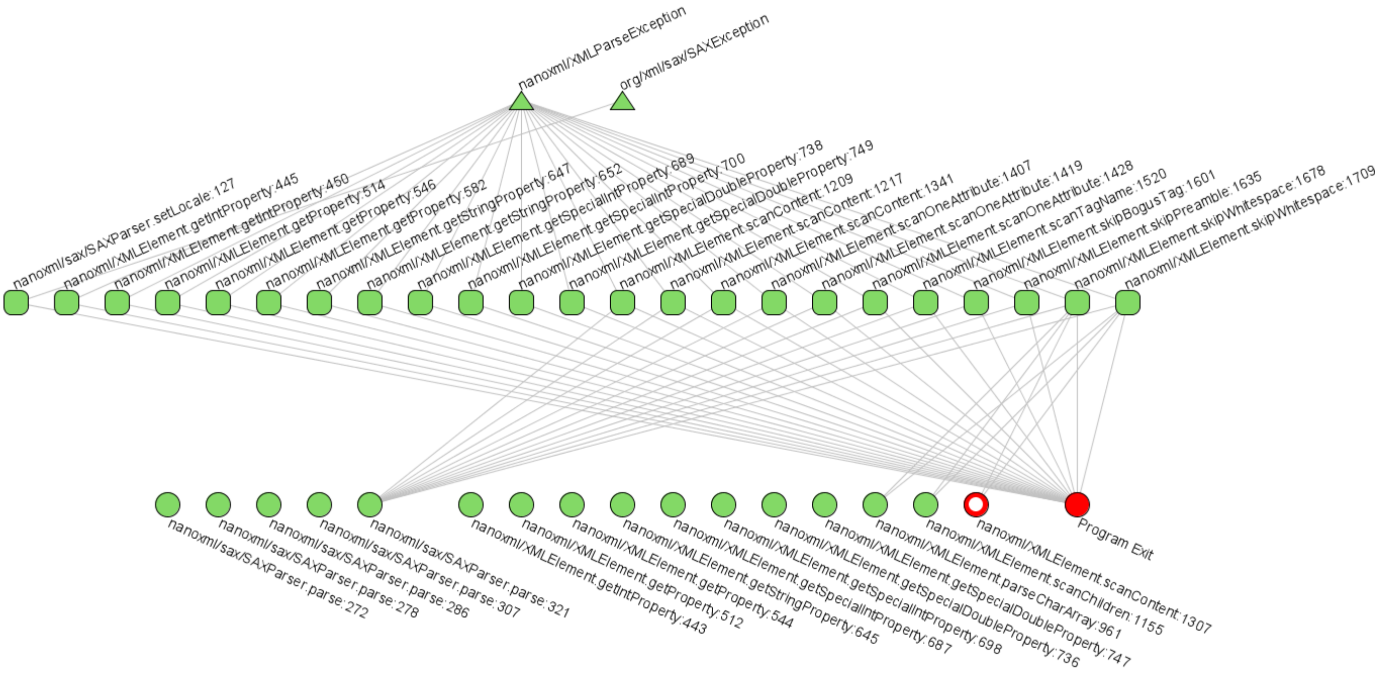

nanoxml/XMLParseException
org/xml/sax/SAXException
nanoxml/sax/SAXParser.setLocale:127
nanoxml/XMLElement.getIntProperty:445
nanoxml/XMLElement.getIntProperty:450
nanoxml/XMLElement.getProperty:514
nanoxml/XMLElement.getProperty:546
nanoxml/XMLElement.getProperty:582
nanoxml/XMLElement.getStringProperty:647
nanoxml/XMLElement.getStringProperty:652
nanoxml/XMLElement.getSpecialIntProperty:689
nanoxml/XMLElement.getSpecialIntProperty:700
nanoxml/XMLElement.getSpecialDoubleProperty:738
nanoxml/XMLElement.getSpecialDoubleProperty:749
nanoxml/XMLElement.scanContent:1209
nanoxml/XMLElement.scanContent:1217
nanoxml/XMLElement.scanContent:1341
nanoxml/XMLElement.scanOneAttribute:1407
nanoxml/XMLElement.scanOneAttribute:1419
nanoxml/XMLElement.scanOneAttribute:1428
nanoxml/XMLElement.scanTagName:1520
nanoxml/XMLElement.skipBogusTag:1601
nanoxml/XMLElement.skipPreamble:1635
nanoxml/XMLElement.skipWhitespace:1678
nanoxml/XMLElement.skipWhitespace:1709
nanoxml/sax/SAXParser.parse:272
nanoxml/sax/SAXParser.parse:278
nanoxml/sax/SAXParser.parse:286
nanoxml/sax/SAXParser.parse:307
nanoxml/sax/SAXParser.parse:321
nanoxml/XMLElement.getIntProperty:443
nanoxml/XMLElement.getProperty:512
nanoxml/XMLElement.getProperty:544
nanoxml/XMLElement.getStringProperty:645
nanoxml/XMLElement.getSpecialIntProperty:687
nanoxml/XMLElement.getSpecialIntProperty:698
nanoxml/XMLElement.getSpecialDoubleProperty:736
nanoxml/XMLElement.getSpecialDoubleProperty:747
nanoxml/XMLElement.parseCharArray:961
nanoxml/XMLElement.scanChildren:1155
nanoxml/XMLElement.scanContent:1307
Program Exit

Fig. 3: *The Flow View showing exception flows in* NanoXML. *The nodes are clustered accordingly to their package and class structure. Nodes within a class are ordered by the line number of the statements they represent.*

The edges between the type-nodes and throw-statement nodes are difficult to see in Figure 2 because of the overlapping labels. The view provides the option to focus on the edges instead of the labels (shown in Figure 8) or to remove the labels completely.

Figure 3 shows the Flow View with a different layout of the graph. In this layout, the nodes are not arranged to minimize edges crossings in the graph, as in Figure 2. Instead, they are clustered accordingly to their package and class structure. Within a class, the nodes are ordered by the line number of the statements they represent. There are two clusters for the type nodes (nanoxml exception type and sax exception type), only one cluster for the throw nodes (they are all in the same class) and three clusters for the catch nodes (nanoxml/sax cluster and nanoxml cluster). The clustered Flow View in Figure 3 shows that the catch statement at line 321 in method

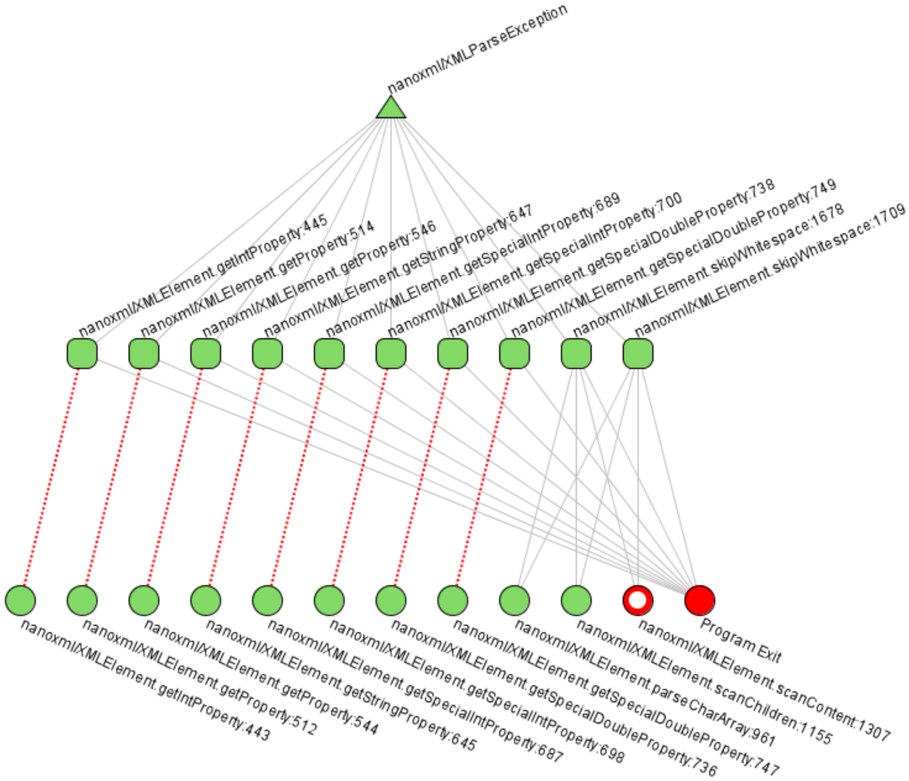

nanoxml/XMLParseException
nanoxml/XMLElement.getIntProperty:445
nanoxml/XMLElement.getProperty:514
nanoxml/XMLElement.getProperty:546
nanoxml/XMLElement.getStringProperty:647
nanoxml/XMLElement.getSpecialIntProperty:689
nanoxml/XMLElement.getSpecialIntProperty:700
nanoxml/XMLElement.getSpecialDoubleProperty:738
nanoxml/XMLElement.getSpecialDoubleProperty:749
nanoxml/XMLElement.skipWhitespace:1678
nanoxml/XMLElement.skipWhitespace:1709
nanoxml/XMLElement.getIntProperty:443
nanoxml/XMLElement.getProperty:512
nanoxml/XMLElement.getProperty:544
nanoxml/XMLElement.getStringProperty:645
nanoxml/XMLElement.getSpecialIntProperty:687
nanoxml/XMLElement.getSpecialIntProperty:698
nanoxml/XMLElement.getSpecialDoubleProperty:736
nanoxml/XMLElement.getSpecialDoubleProperty:747
nanoxml/XMLElement.parseCharArray:961
nanoxml/XMLElement.scanChildren:1155
nanoxml/XMLElement.scanContent:1307
Program Exit

Fig. 4: *The Flow View showing exception flows in* NanoXML *for the catch handlers in class nanoXML/XMLElement. The red dotted lines between catch nodes and throw nodes represent rethrows.*

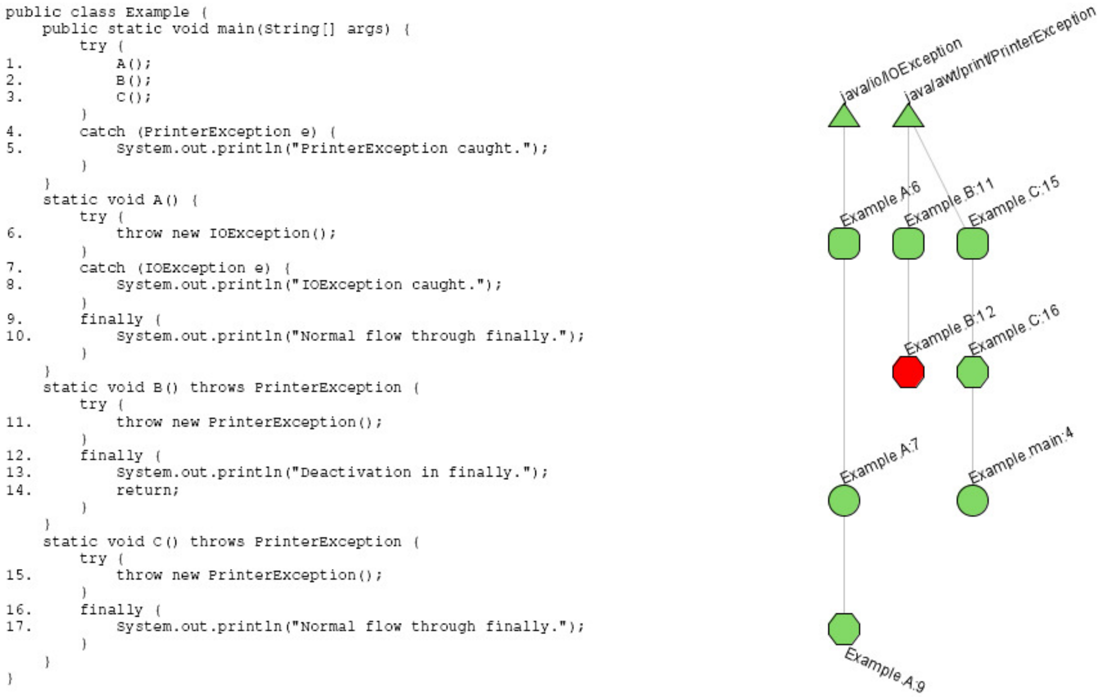
nanoxml/SAXParser.parse may catch 11 throws from class nanoxml/XMLELement and that the two rightmost throw nodes may be caught at three catch nodes, one of them being an empty catch node. Furthermore, the exception from all throws may reach the program exit and there seem to be some unreachable catch handlers.

To learn more about the unreachable catch handlers in class nanoxml/XMLElement, we focus on the flow to catch handlers and also take rethrows into account. Figure 4 shows that all catch handlers in class nanoxml/XMLElement rethrow the caught exception, indicated by the red dotted lines from the catch node to the throw node. Inspecting the source code reveals that all these catch handlers catch runtime exceptions and map them onto nanoXML/XMLParseException using a rethrow.

Because NanoXML does not have finally statements, we use the program in Figure 5 as an example to illustrate how finally statements are integrated in the Flow View. The programs has three methods A, B, and C that are called in the try block of the main method. Method A throws an exception in a try block, the exception is caught in a catch block, the finally block is executed, and the method returns. Method B throws an exception and deactivates it using a return statement in the finally block. Method C throws an exception, executes the finally block, and the exception is caught in the main method.

The Flow View in Figure 5 corresponds to the program. Two additional layers are added for the finally nodes to reflect the two possible flows. If a try block has both a catch and a finally block, and the exception is caught in the catch block, then the finally block is executed after the catch block. If the catch block does not catch the exception (because the type does not match) or no catch block is present, the finally block is executed after the try block and before the control flow leaves the method. The leftmost nodes represent the flow in method A (throw, catch, finally), the nodes in the middle represent the flow in method B (throw, deactivation in finally), and the rightmost nodes represent the flow in method C (throw, finally, catch).

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 (for example the catch block at line 321 in method nanoxml/SAXParser.parse in Figure 3) 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.



```
    public class Example {
        public static void main(String[] args) {
            try {
1.              A();
2.              B();
3.              C();
            }
4.          catch (PrinterException e) {
5.              System.out.println("PrinterException caught.");
            }
        }
        static void A() {
            try {
6.              throw new IOException();
            }
7.          catch (IOException e) {
8.              System.out.println("IOException caught.");
            }
9.          finally {
10.             System.out.println("Normal flow through finally.");
            }
        }
        static void B() throws PrinterException {
            try {
11.             throw new PrinterException();
            }
12.         finally {
13.             System.out.println("Deactivation in finally.");
14.             return;
            }
        }
        static void C() throws PrinterException {
            try {
15.             throw new PrinterException();
            }
16.         finally {
17.             System.out.println("Normal flow through finally.");
            }
        }
    }
```

Fig. 5: *An example program to illustrate different exception flows using finally blocks (left) and the corresponding Flow View (right).*

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 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 throw 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 [16] 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: squares represent throw statements, circles represent catch statements, and octagons represent finally statements. The nodes are colored green unless they are involved in an inappropriate coding pattern in which case they are colored red. 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 6 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. Embedding the exception flow into the view of the entire source code helps the developer to maintain the mental model of the visualization when switching from one flow to another. This embedding also lets the user visualize multiple flows at once and as such supports comparison. For larger programs this approach does not scale, however. Thus, the user can select to view a condensed Contextual View that includes only methods that are involved in the exception flow. An example of this condensed view is shown in Figure 7. In the figure, a throw in method XMLElement.skipWhitespace in package nanoxml reaches the program exit after it is propagated through five other methods; only the methods involved in the propagation are shown in this view.

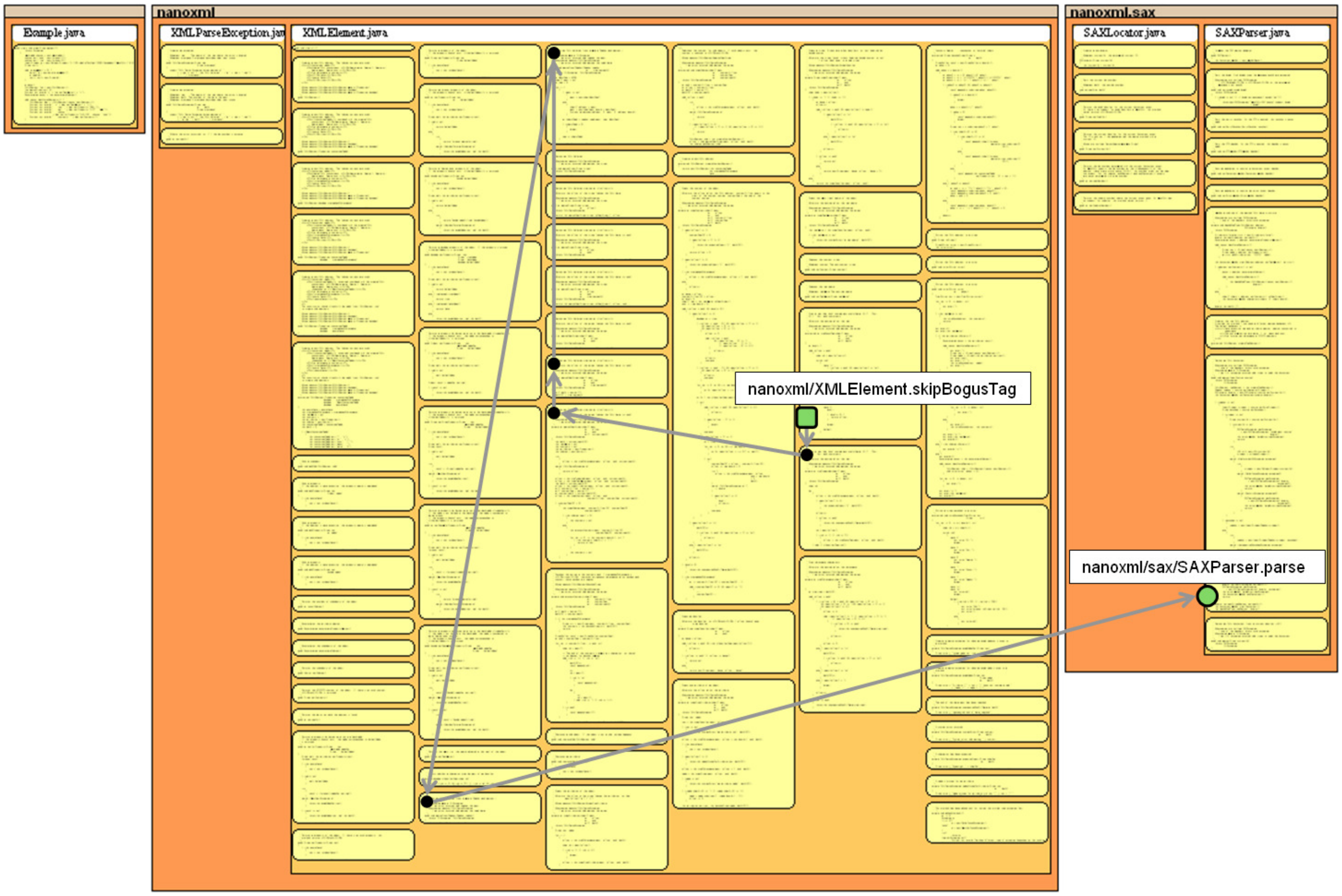


Fig. 6: *The Contextual View showing the exception flow of an exception embedded in the source code. The green square shows the location of the throw statement, the green circle shows the location of the catch statement, and the smaller black circles show the locations of methods along the propagation path.*

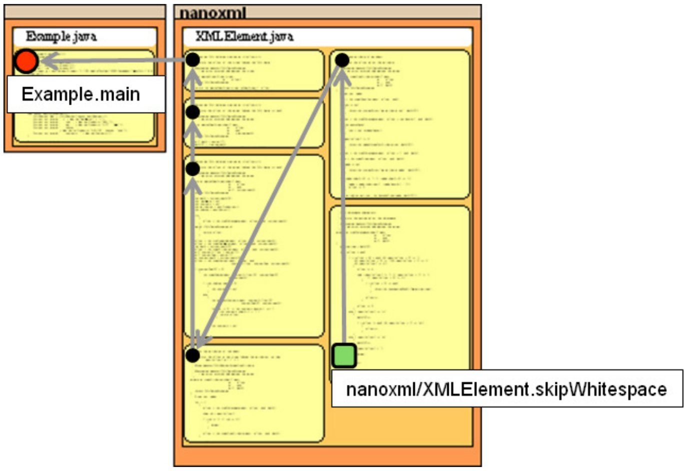


Fig. 7: *The condensed Contextual View showing only the methods that are involved in the exception flow of an exception. The green square shows the location of the throw statement and the red circle indicates that the exception reaches the program exit.*

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 [5]. 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

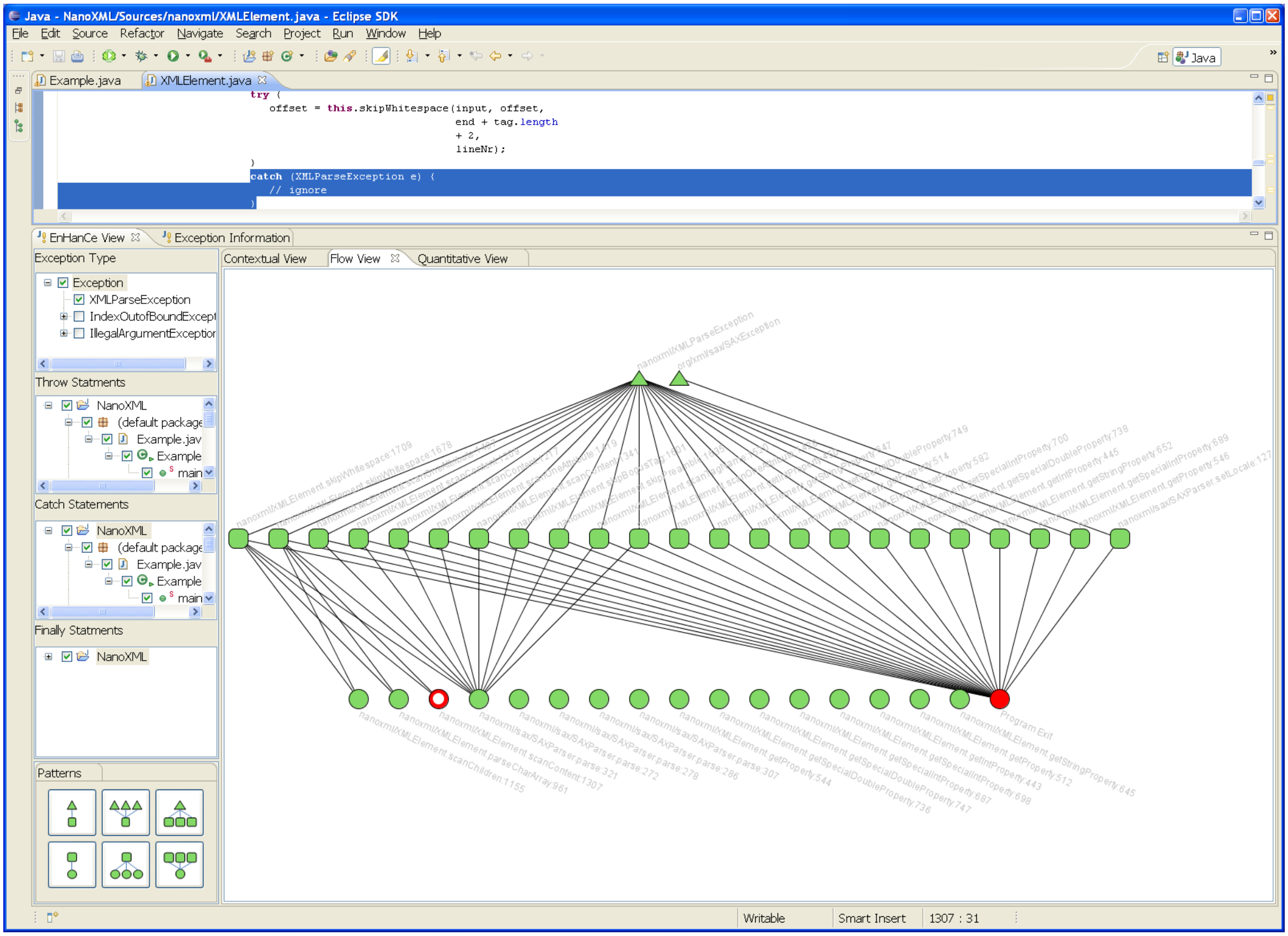
To evaluation our new visualization, we developed a prototype that we call EnHanCe (ExceptioN HANdling CEntric visualization). EnHanCe implements the three views we presented in Section 3 as a plugin framework to Eclipse.[6] EnHanCe is implemented in Java, and has more than 3000 LOC, eight packages, 51 classes, and 381 methods. We used the Eclipse plugin environment[7] to implement the framework of our plugin and the Draw2D and Graphical Editing Framework (GEF)[8] 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 [21]. In the current plugin design, the three views are integrated as three separate tabs in a single Eclipse view.

Figure 8 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.

---

[6] http://www.eclipse.org/
[7] http://www.eclipse.org/pde/
[8] http://www.eclipse.org/gef/


Java - NanoXML/Sources/nanoxml/XMLElement.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Example.java
XMLElement.java
EnHanCe View
Exception Information
Exception Type
☑ Exception
☑ XMLParseException
☐ IndexOutofBoundExcept
☐ IllegalArgumentExceptior
Throw Statments
☑ NanoXML
☑ (default package
☑ Example.jav
☑ Example
☑ main
Catch Statements
☑ NanoXML
☑ (default package
☑ Example.jav
☑ Example
☑ main
Finally Statments
☑ NanoXML
Patterns
Contextual View
Flow View
Quantitative View
nanoxml/XMLParseException
org/xml/sax/SAXException
nanoxml/XMLElement.scanChildren:1155
nanoxml/XMLElement.parseCharArray:961
nanoxml/XMLElement.scanContent:1307
nanoxml/sax/SAXParser.parse:321
nanoxml/sax/SAXParser.parse:272
nanoxml/sax/SAXParser.parse:278
nanoxml/sax/SAXParser.parse:286
nanoxml/sax/SAXParser.parse:307
nanoxml/XMLElement.getProperty:544
nanoxml/XMLElement.getSpecialDoubleProperty:736
nanoxml/XMLElement.getSpecialDoubleProperty:747
nanoxml/XMLElement.getSpecialIntProperty:687
nanoxml/XMLElement.getSpecialIntProperty:698
nanoxml/XMLElement.getIntProperty:443
nanoxml/XMLElement.getProperty:512
nanoxml/XMLElement.getStringProperty:645
Program Exit
Writable
Smart Insert
1307 : 31

```
try {
   offset = this.skipWhitespace(input, offset,
                                end + tag.length
                                + 2,
                                lineNr);
}
catch (XMLParseException e) {
   // ignore
}
```

Fig. 8: *A screenshot of* EnHanCe *in the Eclipse environement shows the Flow View for* NanoXML.

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 8 shows the Flow View for NanoXML with the focus on the edges. The view shows that five of the six patterns discussed in Section 3 exist in NanoXML (the *multiple types to single throw* pattern is not present). 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 [8]. However, the visualization is not restricted to any particular analysis tool, and it can be extended easily to use other analysis tools that provide the required information.

5 EVALUATION

To evaluate our visualization technique, we conducted user studies. Initially, we conducted a *pilot study* to get feedback for designing the detailed study's interview guide with respect to what questions to ask, what areas to cover, and what participants' space to explore. Based on this feedback, we revised our study guide to produce the *detailed study guide*, and used it to conduct the study. This section discusses the study design and the study results.

5.1 Study Design

We first describe the the interview guide design, we then discuss the participant details, and finally, we present the study method.

5.1.1 Interview Guide Design

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 whether ENHANCE can help them to better understand exception-handling constructs; Appendix B gives the complete Interview Guide for the study. 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. Some of the questions are
   - What approach do you follow to understand exception-flow information in the program?
   - While attempting to understand a program, how do you go about tracking the probable path of an exception that is under consideration?
   - Are you satisfied with the way you approach program understanding with respect to exceptions?

   Then, we explained a scenario and asked the participants to explain how they would solve three tasks: (1) finding dependencies between structural elements, (2) changing exception-related code and determining the change impact on the rest of the program, and (3) understanding the entire propagation path of an exception (from its throw to its catch).
2) In the second part of the interview, we first demonstrated how to use the ENHANCE plugin within the Eclipse environment (by running the tool on the NANOXML sample program).[3] We then introduced each view and let the participants explore and navigate the views. Then, for each view, we asked the developers questions concerning their opinions about the views.
   Some of the questions we asked are
   - For each view, how can that view be used to support the comprehension of exception-flow-related tasks?
   - Can you think of any tasks you would perform using this view?
   - Do you think the view is intuitive?

   We then revisited the previous scenario and tasks discussed in the first part of the interview and asked the participants to explain how they would perform these tasks using ENHANCE views.
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 also asked for feedback on the interview guide and the questions asked.

5.1.2 Participant Details

In the pilot study we interviewed three graduate students from the software-engineering group at the Georgia Institute of Technology. Each participant in the pilot study has between one and four years of industry experience in Java software-development. For the detailed study, we interviewed eight software developers. These developers were summer interns at a large multinational organization when the study was conducted. All the participants had experience with Java. Additionally, most of them also had experience developing in programming languages such as C/C++, and during the interviews, they provided information about exception-handling approaches they generally adopted, irrespective of the languages they used. The intern participants had between one and ten years of prior industrial software-development experience. We concentrated the study on developers with considerable experience in Java because currently our visualization tool, ENHANCE, supports Java programs.

5.1.3 Study Method

We used the results of the pilot study [22] to guide our detailed study. Also, based on the feedback received in the pilot study, we revised the semi-structured interview guide for the participants by iteratively modifying it.(The interview guide is attached as appendix B.) After we completed the design of the interview guide, we contacted potential participants by emailing invitations to software developers who had Java experience.

Before beginning the interview with each participant, we explained the goals of the study, and informed the participants that their information would be anonymous and that they could stop the interview at any time, without being required to give a reason. Additionally, we asked permission to audio record the conversation. Before leaving, we asked their permission to send emails to them in case we required any clarifications at a later time. Each interview lasted for approximately one hour. Throughout the interviewing process, we applied the "think-aloud" approach [23].

We adopted a straight-forward data-driven thematic approach for the analysis. We analyzed each interview to produce a summary, and then read through the entire set of field notes multiple times to find cross-cutting sets of common and distinct themes.

5.2 Study Results

In this section we discuss the study results. We first discuss the feedback we got for each view and then we present the general comments about the visualization.

5.2.1 Quantitative View

Participants' comments on the Quantitative View, as explained, are basically related to (1) its layout and (2) the usage of the information that it provided. First, one participant stated that the layout provided quick access to the location of the throw and catch statements in the program. However, the participant thought that the layout of the throw-catch pairs was nonintuitive: in the version of the Quantitative View that we used for the study, the information about the throw-catch pairs was given in a column-row layout. The participant thought that the column-row layout was nonintuitive because it was opposite of the more commonly used row-column layout. We had also received this feedback through informal conversations with other teams members from our lab. Hence, we changed this layout in our current view and now throw-catch pairs are presented in a row-column manner.

Second, seven of the eight intern participants thought it was difficult to understand the Quantitative View and its usage. Participants thought that the dependency-relation information that the view provides might be more useful for project managers than for developers like themselves. However, one participant disagreed, and suggested that the view may be used for understanding the exception concentration in the program and getting an overview of the project with respect to exception handling. Another participant indicated one pattern in the view that he would use to determine dependency information among modules with respect to exception handling: circles on the diagonal on the package view from the top-left to bottom-right (which indicates whether the exceptions can flow across packages).

Participants also made some general suggestions on the visualization design, including

1) showing more useful information, such as putting numbers of throw-catch pair within the bubbles
2) showing information about the total number of pairs at the end of the columns and rows
3) giving some information about the complexity of the module in terms of lines of code (e.g., show complexity information for packages that may be small but have more exceptions)

We incorporated the first suggestion into our current visualization, and plan to consider the other two for our next version of the visualization tool.

5.2.2 Flow View

The participants found the Flow View quite useful. They provided general comments related to the layout, the patterns, and the usage of the view. Regarding patterns, many participants made interesting conclusions about the program based on the exception patterns revealed in the view. Also, participants mentioned that the empty catch block representation was quite useful for refactoring. One participant pointed out that there was one catch handler in the NANOXML's flow view that was handling many of the exceptions in the program.

Regarding the layout, many participants were concerned about the cluttering of the view for large programs, and thus suggested better arrangement mechanism to reduce cross cutting edges. To address this issue, in our latest version of this view, we have provided a facility to select the layout based on structural elements (elements within packages, classes, methods grouped together) or based on minimal cross-cutting edges. Additionally, one participant mentioned that he expected the view to be other way round—from top to bottom, catch-throw-type instead of type-throw-catch—and he explained by saying that the flow of the program is upwards from the catch towards the throw. However, we decided not to change the view based on this one comment.

Participants also had comments about the usage of the view. For instance, they thought that the view would be useful for refactoring, debugging, cleaning up code by facilitating the removal of unreachable catch blocks, ensuring that exceptions are handled in the correct manner, and checking whether exceptions are handled within a package or across packages.

One suggestion that a few participants made was related to the color coding. They suggested that the color coding could be based on the types of exceptions being handled (i.e., have separate colors for each type of exception). We will consider this for our future versions of the visualization.

5.2.3 Contextual View

The participants found the Contextual View very intuitive, and mentioned that it fit well with their mental model of program. With respect to usage, some of the participants mentioned that the view will be helpful for debugging tasks, for providing interclass dependency information based on the number of crossings across classes, and for showing useful flow information of the exceptions. One participant suggested the extension of the view to show dynamic information about exception flow. We are considering this for future work.

However, a few participants raised concerns about the scalability of the visualization for large programs, and how the view would be helpful when the exception flow paths are lengthy. Also, they raised concerns that when multiple throw-catch pairs are shown, the views could get cluttered. They recommended displaying only the relevant methods in the view and not displaying all the methods to address the scalability problem. We incorporated this suggestion into the visualization, and improved the tool to provide additional filters to selected specific structural elements for display in the view.

5.2.4 General Comments

In addition to asking specific questions, we collected general comments about the tool from the participants.

- One participant suggested having another view that shows statistical information of exception flows such as the average number of hops that an exception takes before reaching a catch block, the maximum distance between a throw and catch, and the minimum distance between them.
- Many participants expressed the desire to have direct interaction between the visualization and the code, by facilitating a way to double click on the visual elements in the view to switch to the corresponding code in the program.

We are considering these suggestions for a future version of our tool.

6 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 [14] is integrated in the Jipe environment[9] and its analysis is built on top of the Barat framework.[10] 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 [10], [11] 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 tool[1] 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.

The visualization tool EXPO [5] is a plugin for the Eclipse environment and uses EPTOOLS [8] 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.

EXTEST [12], [13] is also a plugin for the Eclipse environment and uses the Soot Java Analysis and Transformation Framework [24] to compute *exception-catch (e-c) links*[11] 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 [16] view, and it uses richer visualization techniques to give perspectives on exception flow from different abstract levels.

7 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 experience guided the design of our visualization system, which we implemented as a tool called ENHANCE (ExceptioN 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 study with 12 software developers

---

[9] http://jipe.sourceforge.net/

[10] http://sourceforge.net/projects/barat

[11] Given 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 evaluate our visualization, and reported the results. We also incorporated many of the suggestions that the developers made into the visualization presented in the paper. We now need to perform a rigorous evaluation of the system. We plan to perform additional user studies to gain deeper insights into the system's usability and usefulness, and to evaluate its impact on debugging and maintenance tasks.

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.

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 [24], 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.

ACKNOWLEDGMENTS

This work was supported in part by awards froms 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 and our study provided valuable information that guided the development and improvement of our visualization.

REFERENCES

[1] S. Sinha and M. J. Harrold, "Analysis of programs with exception-handling constructs," in *Proceedings of the International Conference on Software Maintenance*, 1998, pp. 348–357.
[2] B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah, "A static study of Java exceptions using JESP," in *Proceedings of the 9th International Conference on Compiler Construction*, 2000, pp. 67–81.
[3] C. F. Schaefer and G. N. Bundy, "Static analysis of exception handling in Ada," *Software - Practice and Experience*, vol. 23, no. 10, pp. 1157–1174, 1993. [Online]. Available: citeseer.ist.psu.edu/schaefer93static.html
[4] H. Shah, C. Görg, and M. J. Harrold, "Why do developers neglect exception handling?" in *Proceedings of the 4th International Workshop on Exception Handling*, Nov 2008, pp. 62–68.
[5] S. Sinha, A. Orso, and M. J. Harrold, "Automated support for development, maintenance, and testing in the presence of implicit control flow," in *Proceedings of the 26th International Conference on Software Engineering*, 2004, pp. 336–345.
[6] D. Reimer and H. Srinivasan, "Analyzing exception usage in large Java applications," in *Workshop on Exception Handling in Object Oriented Systems*, 2003, pp. 10–19.
[7] M. P. Robillard and G. C. Murphy, "Designing robust Java programs with exceptions," in *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2000, pp. 2–10.
[8] S. Sinha and M. J. Harrold, "Analysis and testing of programs with exception handling constructs," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 849–871, 2000.
[9] C. Fu and B. G. Ryder, "Exception-chain analysis: Revealing exception handling architecture in Java server applications," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 230–239.
[10] M. P. Robillard and G. C. Murphy, "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*, 1999, pp. 322–337.
[11] ——, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM Transactions on Software Engineering and Methodology*, vol. 12, no. 2, pp. 191–221, 2003.
[12] C. Fu and B. G. Ryder, "Navigating error recovery code in Java applications," in *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*, 2005, pp. 40–44.
[13] ——, "Testing and understanding error recovery code in Java applications," in *Exception Handling in Object Oriented Systems: Developing Systems that Handle Exceptions*, 2005, pp. 15–26.
[14] B.-M. Chang, J.-W. Jo, and S. H. Her, "Visualization of exception propagation for Java using static analysis," in *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, 2002, pp. 173–182.
[15] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985.
[16] S. G. Eick, J. L. Steffen, and J. Eric E. Sumner, "Seesoft-a tool for visualizing line oriented software statistics," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, 1992.
[17] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java Language Specification*. Prentice Hall, 2005.
[18] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *Proceedings of the IEEE Symposium on Visual Languages*, 1996, pp. 336–343.
[19] D. Reimer and H. Srinivasan, "Analyzing exception usage in large Java applications," in *Proceedings of the ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms*, Jul. 2003, pp. 10–19.
[20] I. G. Tollis, G. D. Battista, P. Eades, and R. Tamassia, *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998.
[21] R. Lintern, J. Michaud, M.-A. Storey, and X. Wu, "Plugging-in visualization: Experiences integrating a visualization tool with Eclipse," in *Proceedings of the 2003 ACM Symposium on Software Visualization*, 2003, pp. 47–56.
[22] H. Shah, C. Görg, and M. J. Harrold, "Visualization of exception handling constructs to support program understanding," in *Proceedings of the ACM Symposium on Software Visualization*, 2008, pp. 19–28.

[23] M. W. van Someren, Y. F. Barnard, and J. A. C. Sandberg, *The Think Aloud Method: a Practical Guide to Modelling Cognitive Processes*. Academic Press, London, San Diego, 1994.

[24] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, “Soot - a Java bytecode optimization framework,” in *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, 1999, pp. 125–135.

APPENDIX A
SURVEY QUESTIONNAIRE

1) What is your current role? [Student / Software Developer / Project Manager / Test Engineer / Other (specify your role)]
2) How many years of experience do you have in Java software development? [less than 1 year / 1-2 years / 2-3 years / 4-5 years / more than 5 years]
3) What was the average size of the projects on which you worked? [Small (less than 5 KLOC) / Medium (5 KLOC–50 KLOC) / Large (greater than 50 KLOC)]
4) Rate your expertise with Java exception-handling in the following areas:
    a) Developing code related to exception-handling constructs [Novice / Less Experienced / Moderately Experienced / More Experienced / Expert]
    b) Reviewing / Refactoring code related to exception-handling constructs [Novice / Less Experienced / Moderately Experienced / More Experienced / Expert]
    c) Testing code related to exception-handling constructs [Novice / Less Experienced / Moderately Experienced / More Experienced / Expert]
5) Consider debugging code that has thrown an exception. Do you think that stack-trace information is sufficient for debugging? [Yes / No] If No, what other information would you like to have with respect to exceptions?
6) Have you experienced code where a thrown exception is handled by a finally block instead of a catch block (e.g., return statement within finally blocks)? If Yes, is this a good usage of 'finally' constructs? [Yes / No / Maybe/ Don't know]
7) Do you think the following information in a graphical format would help in exception-handling tasks, such as debugging and coding?
    a) Displaying where, in the code, the exception can originate (i.e., thrown) [Yes / No / Maybe]
    b) Displaying where, in the code, the exception can be caught [Yes / No / Maybe]
    c) Displaying what type of exception (e.g., FileNotFoundException) can be thrown [Yes / No / Maybe]
    d) Displaying which finally statements would execute before the exception flow reaches the catch block [Yes / No / Maybe]
    e) Displaying the complete flow of the exception (i.e., propagating through different methods because of 'throws' constructs) [Yes / No / Maybe]
    f) Displaying information about where exceptions would be rethrown [Yes / No / Maybe]
    g) Is there any other information you think would be useful?
8) Is it useful to have information about the impact the exception-handling code will have on the rest of the program (e.g., impact of adding new catch block, removing a finally block)? [Useful / Neutral / Useless]
9) Is information such as the number of throw-catch pairs within (across) a method/class/package useful? [Useful / Neutral / Useless]
10) Is understanding of the concentration (distribution) of the exceptions within a method/class/package useful? [Useful / Neutral / Useless]
11) Is exception-dependency information useful (i.e., package A is exception-dependent on package B if A throws an exception caught by B)? [Useful / Neutral / Useless]
12) Is information such as the number of exceptions of a particular type within a method/class/package useful (e.g., the number of occurrences of catch statements handling SQLException type exception)? [Useful / Neutral / Useless]
13) Is there any other useful quantitative data that you would like to see?
14) Would code-coverage information about exception-handling be useful for testing purposes (code coverage describes the degree to which the source code of a program has been executed by test cases)? [Useful / Neutral / Useless]
15) Would test-case classification based on the cardinality of throw and catch statements be useful (e.g., one-throw reaching multiple catches, multiple throws reaching one catch)? [Useful / Neutral / Useless]
16) Are there any other potential problems/issues related to exceptions handling that should be addressed?
17) Are there other suggestions you would like to make?

APPENDIX B
SEMI-STRUCTURED INTERVIEW GUIDE

Motivation/problem definition

1) Would you tell me something about your research work?
2) Have you worked on relatively large Java projects (more than 20KLoc)? Yes/ No
3) In the past, have you ever encountered code that uses exception handling constructs (e.g., Java try, throw, catch, finally)? [Yes / No]
4) What approach do you follow to understand exception-flow information in the program?
5) Suppose you are given the task of understanding the complexity of different structural elements (explain what these are) with respect to exception handling. What do you think will give you the best information to understand this?
6) How would you try to get that information?
7) Suppose you are given the task of understanding the information related to exception flow in a program so that you can make changes in the program. How would you go about this task? Please think aloud and tell me all the steps you would take to approach this problem.
8) While attempting to understand a program, how do you go about tracking the probable path of an exception that is under consideration?
9) Are you satisfied with the way you approach program understanding with respect to exceptions? [Yes / No / Maybe]
10) Do you think that some kind of visualization would be beneficial for performing the above mentioned tasks more efficiently? [Definitely / Maybe / Neutral / Maybe not / Definitely not]
    a) If Yes, what kind of visualization? Have you encountered any visualization for exception-handling data?
    b) If No, would you explain why not?

Prototype tool evaluation: Explain the prototype tool, introduce each view (not the details), current interactions

Quantitative View

1) Can you think of any tasks you would perform using this view (ask this first so as not to lead the participant)?
2) What other uses do you think this visualization can have?
3) Do you think the view is intuitive (i.e., is the view understandable)?
   [Definitely / Maybe / Neutral / Maybe not / Definitely not]
4) Revisiting the previous task of understanding the dependency among different structural elements with respect to exception handling, do you think this view will help you get this information? [Yes / No / Maybe]
    a) If No, what is missing?
    b) If No, could you mention what you found difficult to understand?

Flow View

1) Can you think of any tasks you would perform using this view?
2) What other uses do you think this visualization can have?
3) Do you think the view is intuitive (i.e., is the view understandable)?
   [Definitely / Maybe / Neutral / Maybe not / Definitely not]
4) If not, could you describe what you found difficult to understand?
5) Revisiting the previous task of understanding which other exception-handling constructs will be affected by modifying some part related to exception-handling constructs, do you think this view will help you get this information?
   [Yes / No / Maybe] If No, what is missing?
6) Do you see any patterns?

Contextual View

1) How do you think you would use such a view? For what kind of tasks would you use such a view?
2) What other uses do you think this visualization would have?
3) Do you think the view is intuitive (i.e., is the view understandable)?
   [Definitely / Maybe / Neutral / Maybe not / Definitely not] If not, would you describe what you found difficult to understand?
4) Revisiting the previous task of understanding the entire path that an exception travels from the point where it is thrown to the point where it is caught, along with all the method hops, do you think this

view will help you get this information?
[Yes / No / Maybe] If no, what is missing?

Overall suggestions for the tool

1) How is the layout within Eclipse?
2) What do you think about the interactions? Do you have any suggestions to improve/add interactions?
3) Are there any improvements /suggestions you would recommend for such an interface (ask them to give any kind of feedback/criticism)?
4) Are there any other suggestions or comments you would like to make?