Journal of Information Technology Education: Innovations in Practice Volume 11, 2012

Students' Understanding of Advanced Properties of Java Exceptions

Rami Rashkovits and Ilana Lavy
The Academic College of Emek Yezreel, Israel

ramir@yvc.ac.il; ilanal@yvc.ac.il

Executive Summary

This study examines how Information Systems Engineering School students on the verge of their graduation understand the mechanism of exception handling. The main contributions of this paper are as follows: we construct a questionnaire aimed at examining students' level of understanding concerning exceptions; we classify and analyse the students' responses to the questionnaire in order to determine their level of understanding of the mechanism; and we discuss the students' reflections concerning exceptions. The students were required to demonstrate their understanding of various aspects of Java exception handling: exception throwing and catching, fluency of code in the presence of exceptions, multiple catch clauses, common and different reactions to the various exceptions, passing an exception up the calling chain, proper use of the exception hierarchy, re-throwing an exception, and overriding a method which throws exceptions. The results obtained reveal that almost all the participants understood the basics of throwing and catching an exception, but only a few demonstrated the highest level of understanding of the exception mechanism. As the level of understanding required to address the questionnaire problems increased, the number of study participants who were able to answer correctly decreased substantially. In fact, most of the study participants were not familiar with all the possibilities encompassed by the exception mechanism and so were not able to use them properly. The students' reflections were classified into the following categories: perceptions concerning the benefits of using exceptions; perceptions concerning the complexity of the exception mechanism; and issues concerning the development environment. According to the results obtained we recommend several modifications to the curriculum with the aim of improving the students' ability to properly utilize the advanced properties of exceptions to produce higher quality software.

Keywords: Exception handling; class hierarchy; novice programmers

Introduction

The exception-handling mechanism in modern object-oriented programming languages provides a means to help programmers build robust software systems by separating the normal control flow of a program from the code sections dedicated to exceptional situations (Robillard & Murphy, 1999, 2003). Nevertheless, recent studies indicate that many computer applications mix normal control flow with error handling (Cabral & Marques, 2006; Garcia, Rubira, Romanovsky, & Xu, 2001). Madden and Chambers (2002) argued that novice programmers had difficulty designing and implementing error-handling con-

Editor: Meliha Handzic

structs in their programs. Moreover, it was found that programmers in the software industry tend to avoid error-handling issues and consider them to be tangential to the programming of the main functionality (Shah, Görg, & Harrold, 2010).

In a previous study (Rashkovits & Lavy, 2011), we examined how third-year Information Systems (IS) students apply advanced properties of exception handling after having studied and practised this issue. The results obtained revealed that the majority of students had difficulty designing and implementing an appropriate exception-handling mechanism. More specifically, the students had difficulty exhibiting high levels of assimilation concerning a proper design for a hierarchy of exceptions.

Object-oriented programming languages provide an exception-handling mechanism which enables the handling of an error to be separated from its detection. The use of this mechanism allows programmers to provide a context-based reaction to errors and to enhance the reuse of software modules. In addition, the exception-handling mechanism makes it possible to establish the proper management of exceptions according to their nature. Specifically, it enables programmers to treat related exceptions jointly or separately. In the broader context of software development, it can be said that a proper use of exceptions may contribute to the overall quality of the code.

The use of the exception-handling mechanism permits a method (i.e., a function) to throw exceptions and pass on information regarding the error to other parts of the program. The thrown exception can be caught by other methods along the calling chain, allowing the detected error to be handled. The information regarding the error which is passed on from the method that throws the exception to the method that catches it may include all the information needed to handle the error. Specifically, it may include a meaningful name, textual description of the error, the location in the code where the error occurred, and other related values.

We assume that the tendency of industry programmers to avoid using exceptions properly originates in their professional education. In order to examine our assumption, we explore students' understanding of the Java exception mechanism. Since we believe that a profound understanding of the exception-handling mechanism is essential for its proper implementation, we examined the following aspects of the Java exception mechanism: throwing exceptions from a method; catching exceptions in the calling method; understanding the implications of exception handling for the fluency of the code; implementing multiple catch blocks to handle different exceptions; passing an exception up the calling chain; catching multiple exceptions which are hierarchically related in a single catch clause; re-throwing an exception; and understanding the rules on overriding a method concerning the exceptions that are allowed to be thrown by the overriding method.

The contributions of this paper are as follows: (a) a four-level assessment scale was constructed to measure students' levels of understanding concerning Java exception handling; (b) a questionnaire was built to map the students' levels of understanding concerning the above issue; (c) we highlight and discuss common exceptions-related misunderstandings demonstrated by our study participants; (d) we present and analyse students' reflections on the issues involved; (e) we suggest modifications to the teaching process to enhance the students' understanding concerning Java exceptions.

The remainder of this paper is organized as follows. The next section provides background information on the exception-handling mechanism in Java, instructional courses in exception handling, and known difficulties. The third section presents the study environment, the questionnaire and its expected solutions, and the four-level assessment scale to measure the levels of understanding. The fourth section presents the results obtained, the common faulty solutions provided, and the students' reflections. The fifth section discusses the possible implications for the educational process. Finally, we present our concluding remarks and discuss directions for future researches.

Theoretical Background

In this section, we present a brief literature review regarding exception-handling mechanisms in Java, the difficulties in understanding the exception-handling mechanism, and instructional courses in exception handling.

Exceptions-Handling Mechanism in Java

The Java programming language comprises a mechanism with which to handle exceptions. When an exception occurs during the execution of a method, the programmer can identify the erroneous situation and raise an appropriate exception. The programmer also has to catch and handle the exceptions raised whenever it is possible to create a solution to the exception which has been detected (if such a thing is available).

Java's principles regarding exceptions

Before using an exception, the programmer should import a predefined one or construct a new one. In Java, exceptions are objects whose classes descend from the Throwable class. Throwable serves as a base class for the Error and Exception classes. Error is used as a base class for serious problems, such as the OutOfMemory exception, which are not usually recoverable, and Exception is used as a base class for logical problems that can often be handled and resolved. Exceptions are usually thrown by the Java library packages or the Java virtual machine (JVM: the Java runtime engine) itself. Exceptions are intended to be extended and thrown by programmers to represent abnormal conditions in the program which require special handling. Programmers can use predefined exceptions defined in the Java language (e.g., FileNotFoundException) or define their own exceptions by extending the Java Exception class or one of its descendants. The Exception class has a message attribute that can be used to describe the exception in text. It also includes methods that allow an inquiry to be conducted into the exception (i.e., type, place, cause, stack trace, etc.).

User-defined exceptions

When the programmer defines a new type of exception, he or she may add attributes and methods to indicate the abnormal situation it represents, if this is useful. The RunTimeException class (derived from Exception) and its subclasses do not need to be checked (i.e., these exceptions do not need to be explicitly handled). All other exceptions are checked and need to be explicitly declared and handled in the program. The following is a simple example of how to define a new user-defined exception:

```
class MyException extends Exception {
    MyException (String msg) {super(msg);}
}
```

MyException has inherited all of the characteristics of Exception, including the ability to be thrown and caught. The constructor allows a textual message to be attached to the thrown exception. The programmer may add supplementary attributes and methods, as in any other Java class.

Throwing an exception

The programmer can use MyException (as well as any other declared exception) in any method that he or she implements, if appropriate. In Java, an exception thrown inside a method must be declared in its signature (checked exception). The following is a simple example of a method throwing an exception:

```
void foo() throws MyException, OtherException {
...
    if (an exception has occurred)
            throw new MyException("something bad happened ...");

    // continue
    ...
}
```

When an exception is thrown, the JVM stops the execution of the running method and looks for the adjacent try-catch block that catches the exception which has been thrown. If the appropriate try-catch block is not found in the running method, JVM uses the program stack to find the caller method and to continue this process until an appropriate handler is found. The program control is then transferred to the beginning of the appropriate catch handler (similar to the goto statement).

Catching an exception

When one method calls another method with a throws declaration, it must either surround the method call with a try-catch block or instead declare on the caller's signature that it may throw the specified exception. The programmer usually handles an exception when he or she can do something about it. Otherwise, he or she defers exception handling to the caller, who may also handle the exception or defer it further.

In order to further increase readability, the programmer can surround several methods with the try-catch block, thus allowing better separation of the normal flow of the code from the code segments dealing with exception handling.

The following is a simple example of a try-catch block used in Java:

```
try {
    foo();
    bar();
    fly();
} catch (MyException my) {
    // execute corrective actions
} catch (OtherException other) {
    // execute corrective actions
} finally {
    // execute some other commands
}
```

In the code segment given above, foo(), bar(), and fly() are executed in a sequence, unless an exception is thrown from one of them. For instance, when OtherException is raised, the JVM skips all further execution commands, and control of the program is then transferred to the beginning of the appropriate catch handler.

Inside the catch clause, the programmer can execute corrective commands, log the exception, notify the user and ask for his or her reaction, and so forth. He or she may also re-throw the exception (or another exception) to be handled elsewhere (by the caller's caller) or ignore it and do nothing. The try statement may also include a finally clause which is always executed whether or not a catch clause is executed. The finally clause is usually used to clean up resources, for example, through file closure.

Hierarchy of exceptions

The programmer may construct a hierarchy of exceptions in which various subclasses may inherit an exception. For instance, assume that ExceptionX and ExceptionY are extending MyException.

The advantages of such a hierarchy are revealed when the programmer catches these exceptions and handles them. The programmer can catch and handle each exception separately, as follows:

```
try {
...
} catch (ExceptionX ex) {
   // execute corrective actions related to ExceptionX
} catch (ExceptionY ey) {
   // execute corrective actions related to ExceptionY
}
```

Alternatively, the programmer can handle both exceptions in the same manner, as follows:

```
try {
   ...
} catch (MyException e) {
   // execute corrective actions related to MyException
}
```

In the second code segment, when ExceptionX or ExceptionY is raised by the methods called inside the try-catch block, the JVM looks for the appropriate handler. It starts with the first clause and continues until one is found. As both are inherited from MyException, the JVM will find the raised exception to be appropriate and call off the search. In both cases, control is transferred to the same handler, and hence one reaction is executed for both exceptions.

To summarize, the exception-handling mechanism makes it possible to assign meaningful names to exceptions, separates the normal flow of the program from the exceptional flow, and implements the errors via objects that may carry additional information regarding the context of the exception. In addition, it allows the programmer to handle exceptions wherever he or she considers it to be appropriate (either close to or far from the exception detection site) and to either deal with each exception separately or manage some of them together. For all of these reasons, the use of this exception-handling mechanism results in more easily understandable and maintainable programs.

Instructional Courses in Exception Handling

The IS 2010 Curriculum Guidelines for Undergraduate Degree Programs in Information Systems (Topi et al., 2010) specify a general course in the development of applications, including topics such as program design for requirements analysis, programming concepts and structures, unit testing, and integration. However, the guidelines do not specify that exception-handling issues should be addressed anywhere in the suggested curriculum. As a consequence, each academic institution has its own interpretation of the time, place, and learning methods which should be devoted to exception handling.

Informal discussions with several lecturers from several universities and colleges revealed that students usually study the issue of exception handling only in their second programming course. In their first programming course they usually study a procedural programming language (e.g., C, Python), learning to handle errors in the old-fashioned way using error flags passed as parameters or return values. In the second programming course they study an object-oriented programming language (e.g., Java, C++). They study exception mechanism only towards the end of course, after studying advanced object-oriented mechanisms (e.g., class design, class hierarchies), which are necessary in order to understand all aspects of exception handling. Many lecturers provide the students with an overall explanation of all aspects of the exception mechanism; however, the students usually do not practise all these aspects as the homework usually refers to only a basic level of throwing and catching exceptions. In addition, issues related to exceptions are usually not a substantial part of the final exam.

Lecturers who teach other courses that use object-oriented programming languages, such as data structures and algorithms, told us that they sometimes use the exception mechanism but only at a basic level. For instance, when trying to add an existing item to a Set object, the add method may throw an exception if the item already exists, and the calling method would catch and handle the problem. Other lecturers avoid using exceptions and continue to use the old-fashioned way of handling errors. For instance, when they implement a Stack object, they provide isEmpty and is-Full methods, and they encourage the students to use these methods when they implement the pop and push methods. The tendency to avoid the use of exceptions sometimes stems from the use of certain textbooks in which traditional error-handling is employed.

Difficulties in Understanding the Exception-Handling Mechanism

Exception handling is perceived as being a relatively difficult task by novice programmers (Madden & Chambers, 2002; Manila 2006). Robillard and Murphy (2000) stated that a lack of knowledge regarding the design and implementation of exceptions can lead to complex and spaghetti-like exception-handling codes. They also claimed that the global flow of exceptions and the emergence of unanticipated exceptions are the main causes of difficulties in designing exception constructs.

When a range of exceptions share a common context, a class hierarchy of these exceptions is desirable. In such cases, there are situations in which the same reaction is needed when either one of the related exceptions occurs, while in other situations, an individualized reaction needs to be applied. The construction of a proper hierarchy of exceptions necessitates a high level of abstraction ability, which is not possessed by all novices or even experienced programmers.

In a previous study (Rashkovits& Lavy, 2011), we found that IS students have difficulty coping with the exception-handling mechanism. In that research, the students were asked to design and implement an exception mechanism for a given problem. The obtained results revealed that the students encountered difficulties in applying advanced properties of exception handling. In the present research, we focused on the students' understanding of the mechanism in order to discover the origins of the poor performances revealed in the previous study.

The Study

We conducted a study which aimed to examine students' understanding of the constituents of the Java exception-handling mechanism. We focused on students in order to explore the impact of the educational process on students' understanding of this mechanism. For that reason we found qualitative research methods to be the most appropriate for the aims of the present study.

Environment and Population

The data were collected during the 2010/11 academic year. The study participants included all fourth (and final) year students on a B.Sc. degree course in an IS Engineering School in a regional academic college. These students were chosen for the study since they were on the verge of starting their vocational careers as software developers. These students had completed the following programming courses: "Introduction to Programming with C", "Object-Oriented Programming with Java", and "Data Structures and Algorithms with Java". In the introductory programming course the students were not exposed to exception handling, as this mechanism is not part of the C language. Therefore they learned to handle errors in the old fashioned way, using error flags and specified return values to indicate problems. In the object-oriented programming course (Java), they learned how to throw, catch, and handle exceptions as well as how to define hierarchies of them. The instructor reported that he spent two consecutive lectures (approximately six academic hours) on issues related to exceptions. In the data structures and algorithm course they had to use library exceptions that are part of the Java Foundation Class (e.g., List, Stack, Queue, etc.). The aim of the present study was to provide a summative assessment of the students' level of understanding concerning various aspects of exception handling. The participants were provided with a questionnaire which included various questions concerning their understanding of throwing and catching of exceptions. The students were given approximately one hour to address the questions. Whilst answering the questionnaire, the students were not allowed to use any supplementary materials and had to rely on their previous knowledge. The questionnaire was given to the students at the end of the systems analysis workshop as one of the course requirements, and hence we consider the data to reflect the students' actual understanding.

In order to gain additional information regarding the students' attitudes and their self-perceptions regarding the subject under examination, we conducted informal interviews with 21 students chosen randomly.

Questionnaire and Expected Solutions

In order to assess the students' level of understanding, we designed a questionnaire consisting of several clusters of questions, each addressing a certain level of understanding. The clusters were designed to have an increasing level of complexity regarding various aspects of exceptions. The questions included code fragments and the students were asked to revise them. In addition, the students were instructed to avoid code duplication in their solutions. We elaborate on the levels of understanding according to which we designed the questionnaire in the following section (Analysis methods). To avoid difficulties stemming from the understanding of complex tasks we decided to use a simplified example in which the student could concentrate mainly on the exception mechanism. However, the students were told that the simplified exceptions and reactions to these exceptions represented a realistic and complex problem, and they were expected to handle them appropriately.

The students were presented with the following hierarchy of exception classes both as code fragment and as a schematic diagram (Figure 1):

```
abstract class Problem extends Exception{}
class Bug extends Exception {}
class Baz extends Problem {}
class Boom extends Problem {}
class BazPlus extends Baz {}
```

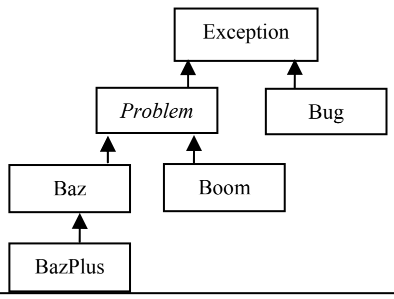

Figure 1: Class hierarchy

In what follows we present the questions and expected solutions organized in clusters.

Cluster 1

The questions (1 and 2) in this cluster require one to demonstrate a basic understanding of the exception mechanism. Question 1 examines the student's ability to throw an exception from a given method and add a proper throws declaration to the method's signature. Question 2 examines the student's ability to surround a block of commands in which an exception might be thrown with a proper try-catch block in the calling method.

Question 1: Please modify method foo() in the given class in such a way that it produces, using Math.random(), a random number between 0 and 1 and throws a BuzPlus exception if the number is lower than 0.5. Please provide a complete implementation for foo().

```
class Class1 {
    void foo(){}
}
```

Expected solution to Question 1:

```
void foo() throws BazPlus {
    if (Math.random() < 0.5)
            throw new BuzPlus();
}
```

Question 2: The method bar() was added to Class1 as follows:

```
void bar(){
    foo();
}
```

Modify bar() in such a way that it will catch the BazPlus exception that might be thrown from foo() and react by printing #.

Expected solution to Question 2:

```
void bar() {
    try {
        foo();
        System.out.println("OK");
    } catch (BazPlus e) {System.out.println ('#'); }
}
```

Without the ability to properly address the above questions one cannot be considered to have the basic knowledge necessary to cope with a single exception. Hence a student who is able to address these questions properly is considered to be at the first level of understanding.

Cluster 2

The questions (3 and 4) in this cluster require one to demonstrate a more advanced understanding of exceptions. Question 3 examines the student's ability to catch and handle two unrelated exceptions independently in the calling method. The student is expected to use two unrelated try-catch blocks, each surrounding a single command. Question 4 examines the student's understanding concerning the need to surround a block of commands with a try-catch block in the calling method, in case the second command depends on the proper termination of the first one. The student is also expected to use a finally clause for the println() method, which must be performed whether or not exceptions are thrown.

Question 3: A new method, fly(), was added to the given class. Method fly() potentially throws an exception of type Boom. In addition, the method bar() was modified as follows.

```
void bar(){
    foo();
    fly();
    System.out.println("OK");
}
```

Modify bar() in such a way that it will catch the exceptions that might be thrown from foo() or fly() and react as follows: print # when BazPlus is caught and print * when Boom is caught. Methods foo() and fly() should be invoked independently, regardless of the proper or erroneous termination of the other method. "OK" should be also printed regardless of the proper or erroneous termination of foo() or fly().

Expected solution to Question 3:

```
void bar() {
    try{
           foo() ;
    } catch (BazPlus e) {System.out.println ('#'); }

    try{
           fly();
    } catch (Boom e) {System.out.println ('*'); }

    System.out.println("OK");
}
```

Question 4: Repeat the previous question, with the following guidelines: fly() should be called only if foo() was terminated properly.

Expected solution to Question 4:

```
void bar() {
    try{
       foo() ;
       fly();
    } catch (BazPlus e) {System.out.println ('#'); }
      catch (Boom e) {System.out.println ('*'); }
      finally {System.out.println("OK"); }
}
```

A student who is able to address Questions 3 and 4 can handle more than one exception properly. He or she knows which commands to include inside the try-catch blocks and when to split the commands into different try-catch blocks and to use the finally clause for commands that need to be performed anyway. A student who is able to address these questions properly is considered to be at the second level of understanding.

Cluster 3

The questions (5, 6, and 7) in this cluster require one to demonstrate additional understanding of the exception mechanism. Question 5 examines the student's ability to catch and handle two unrelated exceptions in different locations along the calling chain. In order to do so, one needs to catch an exception and handle it inside the calling method while passing another exception up the calling chain and handling this second exception there. Question 6 examines the student's ability to handle two related exceptions – one is derived from the other. It requires the student to catch both exceptions in the same method while providing different reactions to each. In Question 7 the problem requirements were modified in such a way that both exceptions have to be caught and handled in the same manner. The student has to identify the hierarchical relationships between the exceptions involved and to utilize these relationships in order to handle both in one catch clause and to avoid unnecessary code duplication.

Question 5: The method zip() was added to the given class as follows:

```
void zip() {
    bar();
}
```

Modify bar() and zip() in such a way that if BazPlus is thrown it will be caught by bar(), which reacts by printing #, and if Boom is thrown it will be caught by zip(), which reacts by printing *.

Expected solution to Question 5:

```
void bar() throws Boom {
    try{
            foo();
            fly();
    } catch (BazPlus e) { System.out.println ('#'); }
}

void zip() {
    try{
            bar() ;
    } catch (Boom e) {System.out.println ('*'); }
}
```

Question 6: A new method, fat(), was added to the given class. The method fat() potentially throws an exception of type Baz. The method bar() was modified as follows.

```
void bar(){
    foo();
    fat();
}
```

Modify bar() and zip() in such a way that when BazPlus or Baz is thrown it should be caught by zip(), which will react by printing # or @ accordingly.

Expected solution to Question 6:

```
void bar() throws Baz {
    foo();
    fat();
}

void zip() {
    try{
            bar() ;
    } catch (BazPlus e) {System.out.println ('#'); }
      catch (Baz e) {System.out.println ('@'); }
}
```

It is worth noting that BazPlus is not included in the throw clause within the signature of bar() since it is derived from Baz, which is already included. In addition, the catch clause of BazPlus precedes the catch clause of Baz since each has a different reaction and if they were reversely ordered the BazPlus catch clause would remain unreachable.

Question 7: Repeat the previous question with the following guidelines: when Boom is thrown it should be caught by bar(), which reacts by printing *; when either BazPlus or Baz is thrown, zip() will catch it and react by printing $.

Expected solution to Question 7:

```
void zip() {
   try{
         bar() ;
   } catch (Baz e) {System.out.println ('$'); }
}
```

A student who is able to address Questions 5, 6, and 7 can handle properly advanced situations in which exceptions are involved. He or she is able to catch or pass up an exception, distinguish between hierarchically related exceptions by catching them in the right order, and utilize hierarchical relationships between exceptions in order to handle them jointly whenever a single reaction is required for both. A student who is able to address these questions properly is considered to be at the third level of understanding.

Cluster 4

The questions (8, 9, and 10) in this cluster require one to demonstrate a profound understanding of the exception mechanism. Question 8 examines the student's ability to catch and handle a complete subtree of exceptions in a single catch clause using the abstract exception at the root of the subtree (i.e., Problem exception). Question 9 examines the student's ability to catch and rethrow an exception. Question 10 requires one to understand the rules on overriding methods concerning the exceptions that the overriding method can throw.

Question 8: The method bar() was modified as follows.

```
void bar(){
   foo();
   fat();
   fly();
}
```

When either BazPlus, Baz, or Boom is thrown, zip() should catch it and print %.

Expected solution to Question 8:

```
void bar() throws Problem {
   foo();
   fat();
   fly();
}

void zip() {
      try{
            bar() ;
      } catch (Problem e) {System.out.println ('%'); }
}
```

It is worth noting that since all the types of exceptions thrown from bar() are derived from the Problem exception, it is sufficient to declare the Problem exception in the throws clause instead of specifying all types separately, although the Problem exception was not explicitly mentioned in the question and is actually an abstract class. Moreover, since the reaction is identical for all kinds of exceptions that might be thrown from bar(), it is sufficient to catch all these exceptions in a single catch clause referring to their super-class (i.e., Problem exception).

Question 9: Repeat the previous question with the following guidelines: when either Baz or Boom is thrown, bar() will catch it and react by printing * in the case of Boom and @ in the case of Baz. However, if BazPlus is thrown, zip() should catch it and react by printing #.

Expected solution to Question 9:

```
void bar() throws BuzPlus {
   try{
      foo();
      fat();
      fly();
   } catch (BazPlus e) { throw; }
     catch (Baz e) { System.out.println('@'); }
     catch (Boom e) { System.out.println('*'); }
}

void zip() {
   try{
      bar() ;
   } catch (BazPlus e) {System.out.println ('#'); }
}
```

Question 10: Assume that the signature of bar()in Class1 is as follows:

```
void bar() throws Baz, Bug
```

Given the following class,

```
class Class2 extends Class1{
      void bar()throws _________ {} //override
}
```

Which of the following types of exceptions might actually be thrown without causing a compilation error from the overridden bar() in class2?

- ☐ Problem
- ☐ Bug
- ☐ Baz
- ☐ Boom
- ☐ BazPlus

Expected solution to Question 10:

- ☐ Problem
- ✓ Baz
- ✓ Bug
- ☐ Boom
- ✓ BazPlus

Since Baz and Bug are declared by the original bar(), the overriding bar() must obey the override rules and hence it can throw only those exceptions or their derived ones.

A student who is able to address Questions 8, 9, and 10 can properly handle complex situations in which exceptions are involved. He or she is able to use the root of the complete subtree of exceptions in spite the fact that (a) it is an abstract exception and (b) it is not mentioned explicitly in the question. He or she is also able to catch and re-throw a derived exception when necessary and demonstrates mastery of the override rules concerning exceptions that are allowed to be thrown by the overriding method. A student who is able to address these questions properly is considered to be at the fourth level of understanding.

Analysis Methods

The study was conducted on a relatively small number of students in one academic college, and hence we found the qualitative research method to be appropriate for this matter. Nevertheless, we would like to stress that although the research method used is of a qualitative nature, we present numerical results as well, since they convey a certain message. Informal discussions with some instructors from other universities and academic collages revealed similar situations. Hence, we believe that the numerical results indicate a wider phenomenon which is not specific to our students.

After the students had finished answering the questionnaire we conducted informal interviews with 21 of them, in which they were asked to provide reflections concerning their performances on the questionnaire. Using analytic induction (Goetz & LeCompte, 1984) and content analysis (Neuendorf, 2002), and reviewing the entire corpus of data to identify themes and patterns of the focal points of the study, we analysed the students' reflections and came up with a set of categories which we will elaborate on later in the section Students' Reflections on the Questionnaire.

Four-Level Assessment Scale

As was stated in the previous section, the questionnaire was built to assess the students' level of understanding regarding the exception-handling mechanism. As the complexity of error handling in a software system rises, the level of understanding of the exception-handling mechanism required for the design and implementation of a high-quality solution increases. Hence, for the analysis process, we defined four categories relating to the level of understanding of the exception-handling mechanism comprising throwing and catching exceptions in order to assess the students' understanding of the exception-handling mechanism in Java. Each level adds more aspects of the Java exception-handling mechanism, as follows:

1. First level of understanding. The student understands how to throw, catch, and handle a single exception. Specifically, he or she knows how to throw an exception from a method, declare it in the method's signature, and catch the thrown exception inside the calling method in an appropriate catch clause.
2. Second level of understanding. In addition to the understanding stated in the previous level, the student understands how to throw, catch, and handle multiple unrelated exceptions. Specifically, he or she understands that a method that throws unrelated types of exceptions must add to its signature a 'throws' declaration for each exception that might be thrown and that the thrown exceptions can be caught and handled in separate catch clauses referring to these exceptions. He or she also understands that a try-catch block can contain one or more commands, and when an exception is thrown, the fluent execution stops and the control is passed to the appropriate catch clause and continues from there. In addition he or she understands the role of the finally clause.

3. Third level of understanding. In addition to the understanding stated in the previous levels, the student understands how to throw, catch, and handle multiple hierarchically related exceptions. Specifically, he or she understands that a method that throws hierarchically related types of exceptions should add to its signature a 'throws' declaration that refers only to the exception that is located higher up in the hierarchy, omitting the derived ones. Furthermore, he or she knows how to catch and handle exceptions in different locations along the calling chain and understands that hierarchically related exceptions can be caught and handled either in separate catch clauses or in a single catch clause referring to the exception that is located higher up in the hierarchy.
4. Fourth level of understanding. In addition to the understanding stated in the previous levels, the student understands that if exceptions comprising a subtree rooted by an abstract exception share a common reaction, they all can be caught using a single catch clause referring to their abstract root. In addition, an exception can be caught and re-thrown further in the calling chain. Also, the overriding method can throw only those exceptions that are identical to or derived from the exceptions that are declared in the throws clause of the overridden method.

Table 1 presents the mapping of the questions to the levels of understanding presented above.

Table 1: Mapping of questions to levels of understanding

Level of understanding	Questions
1	1,2
2	3,4
3	5,6,7
4	8,9,10

Figure 2 demonstrates the calling chain used in the questionnaire. Method zip() calls bar(), which in turn calls methods fat(), fly(), and foo(). This demonstrates the distance between the location where the exception is caught and the location where it occurred. Catching an exception in the calling method (distance = 1) is considered to be easier than passing it up the calling chain (distance > 1). Hence, in the questions referring to the first and the second levels of understanding, as denoted in Figure 2, the students were asked to catch and handle the exception in the calling method, while in the third and fourth levels they were required to pass some of the exceptions up the calling chain.

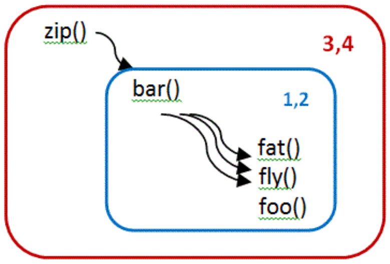

Figure 2: The calling chain

Figure 3 shows, on top of the exception hierarchy used in the questionnaire, the mapping of the exceptions according to the examined levels of understanding. Handling a single exception is considered to be a relatively easy task. Hence, the questions referring to the first level of understanding use only the BazPlus exception, as denoted in Figure 3. Handling multiple unrelated exceptions is considered to be a more complex task. Hence, the questions referring to the second level of understanding use both BazPlus and Boom exceptions. Coping with hierarchically related exceptions is even more difficult, and hence the questions referring to the third level of understanding use Baz, BazPlus, and Boom. At the highest level of understanding, one has to demonstrate ability to cope with a multi-level hierarchy.

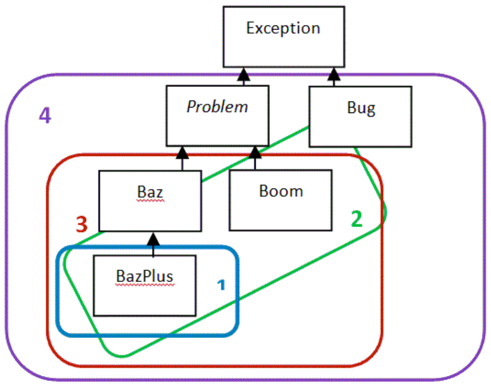

Figure 3: Exception hierarchy and levels of understanding

Results and Discussion

In this section we present an analysis of the students' responses to the questionnaire according to the levels of understanding described earlier. In addition, analysis of the students' reflections offered in the informal interviews regarding their perceptions of the exception mechanism, its constructs, and its use are presented.

Levels of Understanding of the Exception Mechanism

The analysis of the solutions provided by students to the questionnaire is summarized in Figure 4. Each student's level of understanding was set according to the correct answers provided to the various clusters of the questionnaire. As was stated before, the answers given to the questions included in each cluster revealed a certain level of understanding. For example, a student who provided correct answers to Questions 1 and 2 was classified as being at the first level of under-

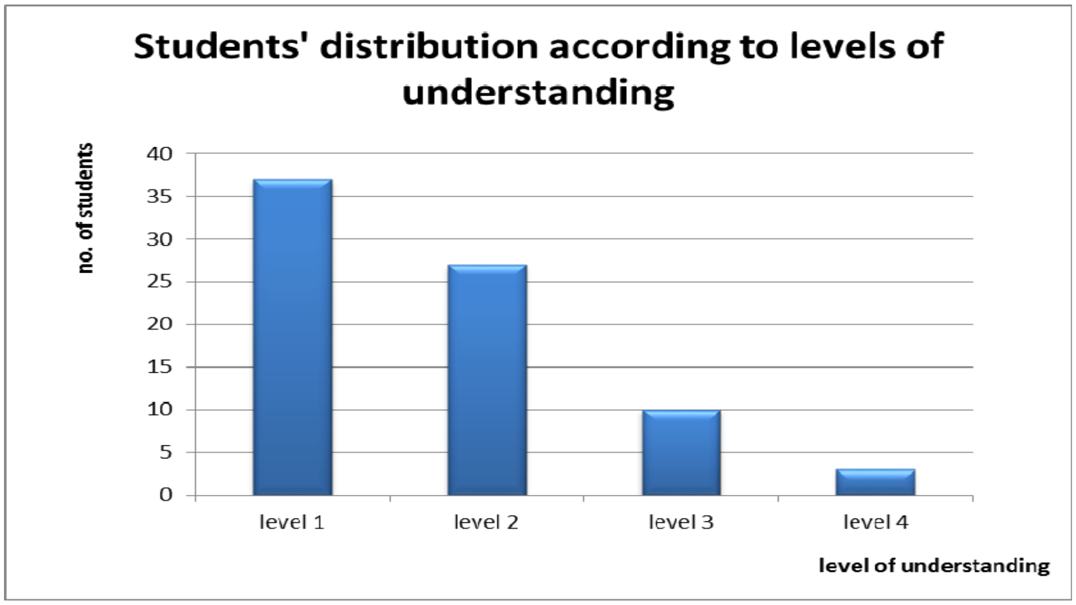

Figure 4: Students' distribution according to understanding levels

standing. If this student also provided correct answers to Questions 3 and 4, then he or she was classified as being at the second level of understanding as well. It is important to note that none of the students who failed to provide correct answers to the questions of a certain cluster succeeded in providing correct answers to the questions of the subsequent clusters.

Level 1

Analysis of the data received from the questionnaire reveals the following: 37 out of 40 students (90.75%) provided correct answers to the first two questions and hence were classified as being at the first level of understanding. The other 3 students did not provide any answers. Most of the solutions provided were similar to the expected one; however, a few students omitted the throws clause from the foo() signature in Question 1. Although such a solution causes a compilation error, we did not consider this solution to be faulty. If the students were using a modern software environment tool during the work on the questionnaire, they would face the compilation error and correct it. Since they addressed the questionnaire using only paper and pencil, they failed to notice the problem. The students had become used to working interactively with sophisticated software development tools that notify them about errors and suggest ways to fix them. The omission of the throws clause can be attributed to the students' tendency to rely on these tools and not necessarily to their lack of understanding concerning the required declaration of a throws clause in this case.

Regarding Question 2, most of the solutions provided were similar to the expected one; however, a few students provided a solution that uses the general Java Exception instead of BazPlus as follows:

```
void bar() {
   try{
          foo();
   } catch (Exception e) {System.out.println ('#'); }
}
```

Although the above solution is syntactically correct, it may lead to unexpected behaviour of the program. For instance, if foo() tries to divide a number by zero, it will raise an exception of type ArithmeticException, which will be caught in the catch clause above, although the reaction of printing # has no connection with this problem. This happens since it is not only BazPlus exceptions that are caught but also all other exceptions that might be thrown inside the try-catch block. This is in line with the findings of Cabral and Marques (2007), who asserted that programmers tend to catch all kinds of exceptions using a catch clause referring to the general Exception class. This kind of solution demonstrates the students' tendency to provide a simplified and effortless solution. The students focused merely on the current requirements, disregarding possible future modifications in which more commands might be added to the try-catch block or more exceptions might be added to the throws clause of foo(). The simplified solution above arises from the students' tendency to invest minimal effort in providing a solution that works. These observations are consistent with those of Clancy (2004), who claimed that many students see program maintenance, modification, and extension as academic exercises and that they are concerned with minimizing the amount of typing they have to do and facilitating superficial changes to program syntax.

To summarize, most students provided correct answers to the first two questions. They demonstrated a basic understanding of the throwing of a single exception and catching it in the calling method.

Level 2

Twenty-seven students out of 40 (67.5%) provided correct answers to the next two questions (3 and 4) and hence were classified as being at the second level of understanding. As for the other 13 students, 8 provided one correct answer and 5 did not provide any correct answers.

The most typical faulty answer to Question 3 was as follows:

```
void bar() {
       try{
          foo() ;
          fly();
          System.out.println("OK");
       } catch (BazPlus e) {System.out.println ('#'); }
         catch (Boom e) {System.out.println ('*'); }
}
```

The problem with the above solution is that fly() will be performed only if foo() does not throw an exception, in contrast to the instructions. The same is true for the println() command, which depends on the proper termination of both foo() and fly(). Students who provided such a solution did not understand the impact of exception throwing on the fluency of the code. They had not assimilated the fact that once an exception is thrown, the program jumps to the closest catch clause that is appropriate for the exception and continues from there. They could not cope properly with try-catch blocks containing more than one command.

As for Question 4, some of the students did not properly apply the requirement to perform fly() only if foo() is terminated properly. They modified foo() to return a Boolean value, indicating whether or not it ended properly, and bar() relies on this indicator when invoking fly(), as follows:

```
// return true if foo ended properly, false otherwise
boolean foo() { … }

void bar() {
       try{
              if (foo()) fly();
       } catch (BazPlus e) {System.out.println ('#'); }
         catch (Boom e) {System.out.println ('*'); }
       System.out.println("OK")
}
```

As shown, the sequence of commands is still surrounded by a try-catch block, which again hints at misunderstanding of the impact of thrown exceptions on code fluency. This kind of solution is interesting, as it mixes both traditional error-handling (e.g., error flags) and exceptions. It may stem from the fact that the students studied and practiced traditional error handling before they learned the exception mechanism. Although the exception-handling mechanism is nevertheless considered to be more efficient, some of the students continued their old habits and used Boolean indicators, even in the presence of exceptions. As for the finally clause, many students did not include it in their solution, but left the println() command outside the try-catch block as the last command of bar(). However, since such a solution also addresses the requirements, we accepted it as a correct one, although they did not use the finally clause as expected.

To summarize, out of the 37 students who provided correct answers to the questions of Level 1, only 27 managed to provide correct answers to the two questions of Level 2. These students demonstrated an ability to work with more than one exception at a time and a good understanding of the code fluency in the presence of exceptions. The other students failed to cope with the questions and demonstrated that they had not assimilated the exception-handling mechanism effectively.

Level 3

Ten students out of 40 (25%) provided correct answers to the next three questions (5, 6, and 7) and hence were classified as being at the third level of understanding. Of the other 30 students, 5 provided two correct answers, 10 provided one correct answer, and 15 did not provide any correct answers.

Among the answers to the three questions at Level 3, several typical faulty answers were provided by the students. With regard to Question 5, some students did not add a throws clause to the signature of bar(), referring to the Boom exception, which is passed up to its caller. This omission can be attributed to the students' habit of relying on the automatic correction of the development environment and hence not paying enough attention to these syntax details.

As for the handling of exceptions along the calling chain, many students failed to correctly transfer the handling of Boom from bar() to zip(). Some of them caught and handled the two exceptions involved in bar(). This may indicate a lack of understanding that when an exception is thrown, Java seeks an appropriate catch clause up the calling chain until one is found. Hence, the catch clause can be located anywhere between main() and the method from which the exception was thrown, according to the programmer's decision. This finding is in line with that of Cabral and Marques (2007), who found that the majority of programmers tend to catch and handle exceptions one level up from where they are thrown.

The following solution was provided to Question 6, in which a redundant declaration of BazPlus was added to the throws clause of bar() in addition to Baz as follows:

```
void bar() throws Baz, BazPlus
```

Although this solution can be compiled and run properly, it may indicate the students' lack of understanding of the exception hierarchy concerning the throws declaration, since BazPlus is derived from Baz. This finding is in line with Rashkovits and Lavy (2011), who found that students have difficulties in using, designing, and implementing hierarchies of related exceptions.

Some students provided the following solution in which they reversed the order of the catch clauses in zip():

```
void zip() {
   try{
         bar()
   } catch (Baz e) {System.out.println ('#'); }
     catch (BazPlus e) {System.out.println ('@');}
}
```

This faulty answer may point to a lack of understanding of the exceptions hierarchy and its use. The lack of understanding is demonstrated by the student's inability to recognize the meaning and consequences of reversing the order of the catch clauses in the solution. Since BazPlus is derived from Baz, it will be caught in the first catch clause, and therefore the second catch clause remains unreachable. Modern Java compilers give notifications concerning such reverse order, notify the programmer about the error, and suggest possible solutions.

Regarding Question 7, some of the students gave the following solution, in which a redundant catch clause referring to the BazPlus exception was added:

```
void bar() {
       try{
             bar()
       } catch (BazPlus e) {System.out.println ('$');}
         catch (Baz e) {System.out.println ('$');}
}
```

This answer may again point to a lack of understanding of exceptions hierarchy concerning the catching location of hierarchically related exceptions. Specifically, when a common reaction to hierarchically related exceptions is required, one catch clause referring to the class located higher in the hierarchy is sufficient. This kind of error may originate in the students' tendency to act in a 'safe' way. That is, they explicitly address all the exceptions specified in the question without investing any additional effort to reach a more elegant solution.

To summarize, out of the 27 students who provided correct answers to the questions of Level 2, only 10 managed to provide correct answers to the three questions of Level 3. These students demonstrated an ability to handle different exceptions along the calling chain, to pass an exception up to the calling method, and to catch and handle exceptions in separate catch clauses or jointly in one catch clause, using the exceptions hierarchy appropriately. The other students failed to cope with the questions, and showed that advanced use of exception handling that efficiently uses hierarchies is beyond their technical skills.

Level 4

Only 3 students out of 40 (7.5%) provided correct answers to the next three questions (8, 9, and 10) and hence were classified as being at the fourth level of understanding. Of the other 37 students, 3 provided two correct answers, 11 provided one correct answer, and 23 did not provide any correct answers.

Among the students who were classified as being at the third level of understanding, and hence proved that they understood how to cope with hierarchically related exceptions, only a few were able to address Question 8 correctly. In this question the students were expected to notice that all the three exceptions involved are derived from the abstract Problem exception, which was not mentioned in the question, and therefore required additional effort.

Many students duplicated the required reaction to the three exceptions involved into separate catch clauses instead of handling them jointly using Problem, as follows:

```
void zip() {
       try{
              bar()
       } catch (BazPlus e) {System.out.println ('$');}
         catch (Baz e) {System.out.println ('$');}
         catch (Boom e) {System.out.println ('$');}
}
```

To be able to answer Question 8 correctly, one has to demonstrate advanced understanding of class hierarchy in the context of exceptions and its consequences. This may be attributed to the fact that Problem was declared as an abstract class, and students were not confident that they can use it as an ordinary exception. Another possible explanation is that students were not aware of the fact that Problem is the common ancestor of BazPlus, Baz, and Boom and can be used to catch them all.

As for Question 9, one of the students provided the following solution in which the Java reflection mechanism was used:

```
void bar() throws BazPlus {
      try{
             foo() ;
             fat();
             fly();
          } catch (Boom e) {System.out.println ('*');
```

```
                catch (Baz e) {
                        if (e instanceof BazPlus) throw;
                        else System.out.println ('#');
                }
        }

        void zip() {
               try{
                      bar();
               } catch (BazPlus e) {System.out.println ('@'); }
        }
```

Although this kind of solution is correct, it is less elegant than the expected solution since it complicates the solution in that the Baz catch clause handles two types of exceptions and distinguishes between them using the instanceof operator. However, catching the exception in the proper place and using the instanceof operator indicates the student's understanding of the exceptions hierarchy. Therefore, the above solution was considered as a correct one.

Many students failed to catch and re-throw the BazPlus exception within bar() and provided a variation of the following solution:

```
void bar() throws BazPlus {
   try{
          foo() ;
          fat();
          fly();
   } catch (Boom e) {System.out.println ('*');
            catch (Baz e) {System.out.println ('#');
}

void zip() {
   try{
          bar();
   } catch (BazPlus e) {System.out.println ('@'); }
}
```

This solution demonstrates again that the students had not comprehended that an exception is caught whenever an appropriate catch clause is found, even though it refers to an ancestor class. Specifically, the students did not understand that the BazPlus exception is caught by the catch clause referring to Baz and therefore will not reach the catch clause in zip() unless it is re-thrown.

Almost all of the students failed to provide a correct solution to Question 10. Some marked all the presented exceptions, while others specified only Baz and Bug, which appeared in the question. The solution in which all exceptions were marked indicated a lack of understanding that the overriding of a method must conform to its signature including its throws clause. The solution in which only Baz and Bug exceptions were marked indicated a lack of understanding that the overriding method is also allowed to throw exceptions that extend Buz and Bug.

To summarize, out of the 10 students who provided correct answers to the questions of Level 3, only 3 managed to provide correct answers to the three questions of Level 4. These students demonstrated an ability to catch many kinds of exception using their abstract superclass and to catch and re-throw exceptions up the calling chain to prevent them from being caught in a more general catch clause referring to ancestor exception as well as a profound understanding of the override mechanism concerning the exceptions that are allowed to be thrown from the overriding method. The other students failed to cope with the questions and showed that their knowledge concerning Java exception was not complete.

The results obtained reveal that almost all of the study participants understood the basics of throwing and catching an exception, but only a few demonstrated the highest level of understanding of the exception mechanism. As the level of understanding required increased, the number of participants who demonstrated mastery decreased. In fact, most of the study participants were not familiar with all the possibilities encompassed by the exception mechanism and so were not able to use them properly. Specifically, the students demonstrated difficulties in copying with the following: use of multiple exceptions; flow of control in the context of exceptions; handling exceptions further up the calling chain; catching and handling hierarchically related exceptions; and overriding methods that throw exceptions.

Students' Reflections on the Questionnaire

After the students had finished answering the questionnaire, we conducted informal interviews with 21 of them, in which they were asked to state their perceptions regarding the exception mechanism, its use, and its constructs in the context of the questionnaire they had just completed. Using analytic induction (Goetz & LeCompte, 1984) and content analysis (Neuendorf, 2002) and reviewing the entire corpus of data to identify themes and patterns of the focal points of the study, the students' reflections were classified into the following categories: perceptions concerning the necessity of the exception mechanism; perceptions concerning the complexity of the exception mechanism; and issues concerning the development environment. In this section, we elaborate on the students' reflections regarding each of these categories.

Perceptions concerning the benefits of using the exception mechanism

Some students provided reflections similar to the following concerning the benefits of using the exception mechanism in programming:

> Student A: "Usually I avoid the use of exceptions. Although I'm familiar with this mechanism I rarely use it. Therefore, I could not remember the details of it."
>
> Student B: "During the course, the exception mechanism was not intuitive at all. Moreover, it seems very complex and more suitable for large and complex software. For the programs I had to provide during my studies it was not necessary to use it."
>
> Student C: "I couldn't understand the logic underlying the separation between the error detection and its handling. Why not handle an error as soon as it occurs? To me it seems more natural to avoid this separation."
>
> Student D: "I consider myself a good programmer in C language, in which this mechanism does not exist, and yet the programs work just fine. I don't like this mechanism and it seems awkward to me. I usually avoid using it, unless I'm forced to do so as in this questionnaire".

The students explain their negative perception concerning the exception mechanism by stating that it is non-intuitive. This may be attributed to the separation of the detection and the handling of an error. They find it easier to handle an error at the spot where it is detected, ignoring the fact that it may harm the code modularity. Other modules depending on a module that handles errors by itself using certain reactions (e.g., exiting the program, displaying an error message) are bound to the reactions used by that module, and the possibility of handling these errors differently is prevented. As a consequence, instead of using the above module, these modules would develop their own one, with their specific reactions to errors embedded in it, repeating the same problem. Such a scenario results in a non-modular code. In order to develop the students' awareness of the benefits of using exceptions, they should be exposed to various tasks in which the advantages of using exceptions are more significant. In addition, the students' prior familiarity with procedural programming languages that do not use exceptions also contributes to their negative perceptions concerning this mechanism. This is in line with the results of Shah, Görg, and Harrold (2010), who found that novice programmers tend to ignore exceptions because of their complex nature.

Perceptions concerning the complexity of the exception mechanism

Many students provided reflections similar to the following concerning the complexity of specific components of the mechanism:

> Student E: "The issue of class hierarchy is rather blurred to me. In the context of exceptions, it is even more of a blur. I never really understood why one cannot use the general Java exception for all purposes".
>
> Student F: "When I was asked to transfer the catching and handling of the exceptions to zip() I didn't know what to do. Usually I catch and handle exceptions immediately when they occur, and I do not convey them further. Therefore I could not provide the desired solution".
>
> Student G: "When I wasn't sure which catch clauses I had to include in the solution, I preferred to stay on the safe side, and added all of them to the solution, although I was aware of the code duplication that this imposes."
>
> Student H: "I find it very confusing when I have to catch or throw more than one exception at a time. Memorizing the rules concerning the correct order of the catch clauses is frustrating. I prefer to deal with one exception at a time."

The students raised several issues which can be attributed to the complexity of the exception mechanism. The concepts of class hierarchy and method override are part of the object-oriented paradigm which necessitates a certain level of abstraction and hence is difficult to comprehend (Or-Bach & Lavy, 2004; Sim & Wright, 2001). Also the fact that when an exception is thrown the control is passed to the nearest catch clause up the calling chain requires a profound understanding of the fundamentals of computing (memory stack, flow of control, etc.) and, hence, is not easy to grasp. The variety of options available in the throwing and catching mechanism (e.g., which exceptions to throw, where to catch them, and the option of handling multiple exceptions jointly) raises the complexity of the exception mechanism, and demonstrating mastery of it requires high order thinking. In order to develop the students' ability to use the advanced properties of exceptions, they should be exposed to complex tasks in which such advanced properties are required.

Issues concerning the development environment

Some students provided reflections similar to the following concerning their working habits when they write programs that include exceptions.

> Student I: "I don't remember the correct syntax for using the exceptions. When I write programs in Java, the Eclipse environment completes all the necessary parts of the try-catch block and throws clauses."
>
> Student J: "Usually when I need to override a method I do not pay attention to the throws clause in its signature. If there's a problem with my implementation, I expect the development environment to notify me, and I usually use their suggestions regarding the required correction without paying attention to its details."

Modern development environments provide the students with technology tools that facilitate the process of programming. Among them is the automatic completion of code fragments (e.g., suggesting automatic wrapping of code fragments with try-catch blocks, instant indication of faulty throws clause declaration, automatic creation of new exception classes, etc.). In addition to their obvious advantages, these tools cause their users to develop dependency on them. This dependency may cause the students to automatically accept the tools' suggestions without thinking further about the consequences. Moreover, the tools usually provide immediate suggestions that do not take into consideration advanced design properties. Since the students tend to accept these suggestions without investing time and effort to provide a better solution of their own, the code they provide is poor, and the benefits of exceptions are not utilized.

Instructional Implications

Mastery of exception handling enables one to design robust and clear computer programs. Unfortunately, many students in our study demonstrated only basic understanding of the issue and failed to demonstrate profound understanding of its advanced properties. In-depth examination of the curriculum reveals that insufficient time is devoted to this issue and more attention should be paid to it. Usually, exceptions are taught towards the end of the second programming course (object-oriented programming) after advanced object-oriented topics such as class inheritance and polymorphism have been learnt. Obviously, poor understanding of these concepts results in difficulties in understanding the advanced properties of exceptions (i.e., exceptions' hierarchies). In addition, most of the examples used by instructors during the learning process are short and simplified (for example, see Lewis, Loftus, Struble, and Cocking, 2003). We believe that although simple examples are useful to effectively teach the topic's principles, they fail to convey the significance of the advanced properties of the exception handling mechanism for the code quality. Additionally, we believe that students are not sufficiently exposed to exceptions in contexts other than the object-oriented programming course.

Therefore, to facilitate the students' understanding of the advanced properties of the exception handling mechanism we suggest the use of spiral learning (Harden & Stamper, 1999). Instead of teaching the whole mechanism at once we suggest that the learning process be decomposed into several iterations, each corresponding to other related issues, as follows:

(1) First, teach the basic principles of catching and handling pre-defined exceptions during the first programming course, as no sophisticated object-oriented constructs are required for this purpose, and plenty of pre-defined library exceptions are available for this matter. In this way, the students will become acquainted with exceptions earlier, and gain more practice in using them;

(2) In the second programming course, after gaining sufficient knowledge of and practice in using basic object-oriented constructs (i.e., classes and methods), present the concept of throwing predefined exceptions (i.e., library exceptions) from a method and catching them along the calling chain;

(3) After sufficient knowledge of class hierarchy and practice in using it have been gained, elaborate on defining new exceptions derived from Exception and rehearse the throwing and catching of exceptions;

(4) After sufficient knowledge of polymorphism and practice in using it have been gained, elaborate on catching hierarchically related exceptions jointly or separately while rehearsing all the concepts previously taught again;

(5) In advanced programming courses (e.g., data-structures, algorithms, distributed systems), incorporate tasks involving the use of advanced properties of exceptions in order to demonstrate their importance and relevance in other contexts.

Concluding Remarks

In this paper we have presented and analysed college students' understanding concerning various aspects of exception handling. The results obtained reveal that most of the study participants understood the concept of Java exception handling at a basic level. However, the majority of them had difficulty understanding the advanced properties of the concept and its possible uses in depth. Although the students had been taught and had used the exception-handling mechanism, they did not properly understand the advanced exception-handling mechanisms offered by the Java programming language. The students had difficulty exhibiting high levels of understanding concerning the following: use of multiple exceptions; flow of control in the context of exceptions; handling exceptions further up the calling chain; catching and handling hierarchically related exceptions; and overriding methods that throw exceptions. These results are consistent with previous research regarding the object-oriented design capabilities of novice programmers (Lavy, Rashkovits, & Kouris, 2009; Or-Bach & Lavy, 2004; Sim & Wright, 2001).

In a previous study (Rashkovits & Lavy, 2011) that was conducted in another regional college, we found that students had difficulty designing and implementing exceptions. To examine the possible reasons for these difficulties, we conducted the current study in another regional college, where the understanding of various properties of exceptions was examined. The findings revealed by the present study support our assumption that these difficulties originate from the students' lack of understanding of the advanced properties of the exception-handling mechanism, which is one of the fundamental constituents of software design and implementation. Informal discussions with colleagues from other colleges and universities revealed similar problems.

To confirm our findings we believe that further research with a large number of participants from different institutes should be conducted. In addition, we plan to conduct a comparative research in which the suggested modifications to the learning process will be examined and comparisons will be made with the results obtained in the current study. Another possible future research can focus on additional aspects that were not addressed in the current study, such as exception handling in multi-thread programs and client-server environments.

References

Cabral, B., & Marques, P. (2006). Making exception handling work. *Proceedings of the 2nd Conference on Hot Topics in System Dependability, 9*. Seattle, WA.

Cabral, B., & Marques, P. (2007). Exception handling: A field study in Java and .NET. *ECOOP 2007, LNCS 4609 – Object-oriented programming*. Berlin, Heidelberg: Springer-Verlag.

Clancy, M. (2004). Misconceptions and attitudes that interfere with learning to program. In S. Fincher & M. Petre (Eds.), *Computer science education research* (pp. 85–100). Lisse, The Netherlands: Taylor & Francis.

Garcia, A. F., Rubira, C. M. F., Romanovsky, A., & Xu, J. (2001). A comparative study of exception handling mechanisms for building dependable object-oriented software. *The Journal of Systems and Software*, *59*, 197–222.

Goetz, J. P. & LeCompte, M. D. (1984). *Ethnography and qualitative design in educational research*. New York: Academic Press.

Harden, R. M., & Stamper, N. (1999). What is a spiral curriculum? *Medical Teacher, 21*, 2.

Lavy, I., Rashkovits, R., & Kouris, R. (2009). Coping with abstraction in object orientation with special focus on interface class. *The Journal of Computer Science Education*, *19*(3), 155–177.

Lewis, J., Loftus, W., Struble, C., & Cocking, C. (2003). *Java software solutions, AP version*. Boston, MA, USA: Addison-Wesley Longman.

Madden, M., & Chambers, D. (2002). Evaluation of student attitudes to learning the Java language. *Proceedings of the Inaugural Conference on the Principles and Practice of Programming* (pp. 125–130). Dublin, Ireland.

Manila, L. (2006). Progress reports and novices' understanding of program code. *Proceedings of the 6th Koli Calling Baltic Sea Conference on Computing Education Research* (pp. 27–31). Uppsala, Sweden. Koli Calling.

Neuendorf, K. A. (2002). *The content analysis guidebook*. Thousand Oaks, CA: Sage Publications.

Or-Bach, R., & Lavy, I. (2004). Cognitive activities of abstraction in object-orientation: An empirical study. *The SIGCSE Bulletin, 36*(2), 82–85.

Rashkovits, R., & Lavy, I. (2011). Students' strategies for exception handling. *The Journal of Information Technology Education (JITE), 10,* 183–207. Retrieved from http://www.jite.org/documents/Vol10/JITEv10p183-207Rashkovits939.pdf

Robillard, M. P., & Murphy, G. C. (1999). Analyzing exception flow in Java programs. *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering. Lecture Notes in Computer Science, 1687*, 322–337. New York: Springer-Verlag.

Robillard, M. P., & Murphy, G. C. (2000). Designing robust JAVA programs with exceptions. *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 25*(6), 2–10. New York: ACM Press.

Robillard, M. P., & 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. New York: ACM Press.

Shah, H., Görg, C., & Harrold, M. J. (2010). Understanding exception handling: Viewpoints of novices and experts*. IEEE Transactions on Software Engineering, 99*, 150–161.

Sim, E. R., & Wright G. (2001). The difficulties of learning object-oriented analysis and design: An exploratory study. *Journal of Computer Information Systems, 42*(4), 95–100.

Topi, H., Valacich, J. S., Kaiser, K., Nunamaker, J. F., Sipior, J. C., de Vreede, G. J., & Wright, R. T. (2010). Curriculum guidelines for undergraduate degree programs in information systems. *ACM/AIS task force*. Retrieved 12 May 2012 from http://blogsandwikis.bentley.edu/iscurriculum/index.php/IS_2010_for_public_review

Biographies

Rami Rashkovits is a Lecturer at the Academic College of Emek Yezreel since 2000 in the department of Management Information Systems. His PhD dissertation (in the Technion) focused on content management in wide-area networks using profiles concerning users' expectations for the time they are willing to wait, and the level of obsolescence they are willing to tolerate. His research interests are in the fields of distributed systems as well as computer sciences education.

Ilana Lavy is an associate professor with tenure at the Academic College of Emek Yezreel and is the department head of Management Information Systems. Her PhD dissertation (in the Technion) focused on the understanding of basic concepts in elementary number theory. After finishing doctorate, she was a post-Doctoral research fellow at the Education faculty of Haifa University. Her research interests are in the field of pre service and mathematics teachers' professional development as well as the acquisition and understanding of mathematical and computer science concepts. She has published over seventy papers and research reports (part of them is in Hebrew).