Students' Strategies for Exception Handling

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

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

Executive Summary

This study discusses and presents various strategies employed by novice programmers concerning exception handling. The main contributions of this paper are as follows: we provide an analysis tool to measure the level of assimilation of exception handling mechanism; we present and analyse strategies to handle exceptions; we present and analyse solutions provided by novice programmers; we classify and analyse the participants' reflections concerning their solutions. Modern programming languages, such as Java, provide the programmer an elaborated object oriented exception mechanism; enable him to handle exceptions in a more convenient way. The aim of this study was to discover the strategies novice programmers are using to handle exceptions and whether they utilise the advantage of the modern exception handling mechanism. For that matter the participants (college students) had to provide a written solution to a given problem with a special focus on exception handling. In addition each of them was interviewed. The analysis of the solutions provided was carried out according to a set of software quality criteria (clarity, modularity and extensibility) adapted to exception handling characteristics. These criteria were used to explain the advantages and disadvantages of the solutions from a software engineering perspective. The solutions provided were also analysed according to a classification of levels of assimilation concerning the structure of exceptions, based on the SOLO taxonomy. The first level of assimilation refers to solutions in which no exception mechanism was used, and the errors were handled in the old fashion way (local handling or return of an error code). The fifth and last level refers to solutions that used adequate hierarchy of exception classes allowing easy extension according to future requirements and enabling the handling of multiple exception altogether or handle each separately. In between these levels there are strategies that used exception mechanisms without the exhausting of its advantages. The results obtained reveal that only few participants (7 out of 56) provided a solution that was classified to one of the two highest assimilation levels, while many (23 out of 56) did not use exception mechanism at all. The rest of the students (25 out of 56) used the exception mechanism poorly (i.e., used only Exception class or did not use hierarchy of exceptions). The participants had difficulties in utilising the advanced exception handling mechanisms and in exhibiting a high level of abstraction with regard to the proper design of a hierarchy of exceptions. The students' statements collected during the interviews were classified into the following categories: misconceptions concerning code quality; misconceptions concerning exception handling; difficulties in understanding the exception handling mechanism; the perceived importance of exception handling; and a lack of programming experience. During the interviews the students provided explanations concerning the reasons they were not fully utilized the exception mechanism. Among them are: lack of practice, too few demonstrated examples, focussing merely on providing a working so-

Material published as part of this publication, either on-line or in print, is copyrighted by the Informing Science Institute. Permission to make digital or paper copy of part or all of these works for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage AND that copies 1) bear this notice in full and 2) give the full citation on the first page. It is permissible to abstract these works so long as credit is given. To copy in all other cases or to republish or to post on a server or to redistribute to lists requires specific permission and payment of a fee. Contact HPublisher@InformingScience.orgH to request redistribution permission.

Editor: Thomas Connolly

lution without considering quality issues. Finally we recommend on some modifications to the curriculum, among them: to expose the students earlier to the exception mechanism; to throw built-in Java exception to address validation errors; and to put more attention to quality issues.

Keywords: Exception handling; class hierarchy; source code quality

Introduction

One of the most important constituents of software systems is exception handling, and a substantial amount of research has been performed in order to develop tools and techniques to assist programmers in incorporating exception handling into their programs (Robillard & Murphy, 1999, 2003).

In this paper, the term "exception" refers to any situation in which some of the rules embedded in a problem are violated. For instance, trying to withdraw money from an overdrawn account will result in an exception. The term "exception handling" refers to a certain mechanism that is designed to detect a violation, to notify someone about the violation, and to perform corrective actions.

In procedural languages such as C, when an exception is detected it is handled locally within the function. Such treatment may lead to complex code in which a function contains many lines of code designed to handle all normal and exceptional flows. Moreover, a uniform method for dealing with exceptions is hard to achieve when the exceptional flow is scattered all over the code. In some cases, the function in which the exception is detected cannot address the problem as it lacks a broader context. In this case, it is reasonable to delegate the task of handling the exception to the caller. Upon detecting an exception, the function returns a specific value to its caller, indicating that a problem has been detected. The caller then uses this value to identify the specific exception and to react accordingly. Alternatively, if the caller itself lacks the ability to handle the exception, it can delegate the task of handling the exception up the calling chain. This kind of treatment can lead to the following problems: the caller code can become far too complex, as when following each invocation of a called function it must check the results and handle exceptions if any are reported, and the values used to identify the exceptions may be arbitrary and meaningless and hence harm the readability of the code.

Object-oriented languages such as Java provide an improved mechanism for handling exceptions. The function in which the exception is detected can pass information regarding the problem that has been detected to other parts of the program. The information that is passed on may include all of the information needed to handle the problem, such as a meaningful name, textual descriptions and related values. Handling exceptions in the proper way may lead to simpler and more modular code, which is extremely important for the maintainability of software.

Although this exception-handling mechanism is a significant improvement on the previous methods, it is still not implemented efficiently by programmers. For instance, programmers throw and catch generic exceptions, resulting in poor exception handling, leading to a decrease in software quality and dependability (Cabral & Marques, 2006; Coelho et al., 2008; Garcia, Rubira, Romanovsky & Xu, 2001). Advanced techniques such as "aspect-oriented programming" also exist, which further reduce the complexity of exception detection and handling and which can cut the code dedicated to exception detection and handling significantly (Lippert & Lopes, 2000).

Large software systems are usually very complex. In such systems, the code devoted to exception handling is extensive and complex. In reality, up to two-thirds of a program is dedicated to exception handling (Garcia et al., 2001). Therefore, using the exception mechanism provided by the software language may contribute significantly to the modularity, readability and maintainability of the code (Filho et al., 2006; Filho, Garcia & Rubira, 2007).

In the process of training novice programmers to deal with large software systems in their future vocation, they should be encouraged to acquire proper programming skills including effective exception handling methods. In the present study, we examined the assimilation of the Java exception mechanism among third year Management Information Systems (MIS) students after they had studied and applied this mechanism. We examined the various strategies the students used when asked to address an exception handling problem. In this paper, we will summarise these strategies and analyse their attributes, advantages, shortcomings and the underlying motivation for using them. In addition, in-depth interviews were conducted in order to understand the students' strategies and their trains of thought.

Theoretical Background

In this section, we will present a brief literature review regarding "design by contract" exception handling, exception handling structures, exception handling mechanisms in Java, the difficulties involved in applying exception handling in programming and instructional courses in exception handling. As the analysis of these findings is based on the students' levels of understanding, we will present the Structure of the Observed Learning Outcome (SOLO) taxonomy as a theoretical basis.

Design by Contract

"Design by contract" is an approach to designing computer software (Mitchell & McKim, 2002). It suggests that software designers should define formal, precise, and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions, and invariants. These specifications are referred to as "contracts". A contract involves the software component as a whole at every level of its refinement, including down to the level of methods.

The main idea of "design by contract" concerns how the components of a software system collaborate with each other, on the basis of mutual obligations and benefits. A method's precondition is a description of an obligation that the calling method has and a benefit that the called method receives, as it frees it from having to handle cases outside of the precondition. A method's postcondition is an obligation for the called method and a benefit for the calling method, as it guarantees results. A class invariant is a guarantee that a certain property is assumed on all class methods entry and exit. When a method detects a violation of a precondition, it should handle the exception.

Exception Handling

Exceptions stem from the system requirements with regard to the problem conditions that the system must follow. These requirements include guidelines regarding legal inputs and class invariants. For instance, in a banking system, the rules of the business refer to conditions for cash withdrawal (e.g., the amounts which can be withdrawn and current balance constraints) (Siedersleben, 2006). In terms of "design by contract", the class invariants of the account would refer to "invariant: balance ≥ 0"; the precondition of the withdrawal method would be "precondition: balance ≥ withdrawal amount", and the postcondition would be "postcondition: balance = balance + withdrawal amount".

The programmer has to implement both the normal flow of the code (e.g., a successful withdrawal) and the exceptional flow (e.g., an illegal withdrawal violating the precondition or the invariants). The withdrawal method should notify its caller whenever the operation fails.

In the scope of this paper, we will present and analyse strategies used by novice programmers (i.e., college students) to address the design and implementation of exception handling.

Exception Handling Structures

The programmer may use control-flow structures in order to detect exceptions (invariants or pre-condition violations) and to notify the calling method about them, as detailed herein.

Error codes. The programmer uses certain return values, for instance -1, to notify the caller that an exception of a specific kind has been detected, as shown in the following example:

```
void foo() {
    ...
    if (some exception has been detected) {
        return -1;
    else if (another exception has been detected) {
        return -2;

    // everything is ok
    // do whatever foo needs to do
    ...
}
```

The following is an example of the caller's code:

```
int code = foo();
if (code == -1) { // exception of type-1 has been detected
    // execute corrective actions;
} else if (code == -2) { // exception of type-2 has been detected
    // execute corrective actions;
} else // successful
    code = bar();
    if (code == -3) { // exception of type-3 has been detected
        return -3; // delegate the task of handling this exception upwards
    } else if (code == -4) { // exception of type-4 has been detected
        // execute corrective actions;
    } else // successful
        // continue with the normal flow
        code = baz();
        ...
    }
}
```

The normal flow of the code segment given above includes the calls for *foo*, *bar* and *baz*. All of the other statements in the code segment above are related to exception handling. The caller itself may not handle some errors and may instead delegate the task of handling them to its caller (see the line highlighted above), and therefore the caller's caller code may appear to be complex as well.

In cases in which the return value of *foo* has already been dedicated to other purposes, the programmer will have to use return values that are not returned by *foo* as normal values or instead use a global exception flag (or a parameter sent by reference) to indicate an exception.

The caller must check after each call to a method whether an exception has been detected and reported (as demonstrated in the code segment above) and handle the specific exception indicated by the called method, if any.

As shown, mixing exception handling with the normal flow results in a complex program. Moreover, this solution is inefficient, as it involves checking for exceptions each time a method is called. Such a program will be barely understandable and difficult to maintain. Therefore, this kind of approach is undesirable.

Local exception handling. In order to avoid the complexity shown above, some programmers prefer to handle exceptions locally (within *foo, bar,* and *baz*) without indicating an exception to the caller. The following is a simple example of local exception handling inside a method:

```
void foo() {
    if (exception has been detected) {
        // execute corrective actions;
        ...
    if (another exception has been detected) {
        // execute another corrective actions;

    // everything is ok
    // do whatever foo needs to do
    ...
}
```

In this mechanism, the caller calls the *foo* and cannot tell whether or not it was successful. The *foo* itself becomes far too complex and its reactions to exceptions are always in the same context, i.e., *foo* cannot be used in other contexts. For instance, if *foo* is part of a console application, it might print an exception message to the console. However, using it in a Graphical User Interface (GUI) application is useless, as no console messages are available. The use of local exception handling only shifts this complexity to the methods themselves, disabling context-based reactions to the exceptions that are detected. Similarly to the previous strategy (error codes), this program is barely understandable and difficult to maintain. Therefore, such an approach is undesirable.

Exception handling mechanisms. Modern programming languages such as Java and C# provide mechanisms with which to address exception handling. Although exception handling mechanisms differ from one language to another (Garcia et al., 2001), they all share common attributes. An exception handling mechanism is used to address both runtime exceptions (emergencies) and user-defined exceptions. The following is a simple example of an exception handling mechanism:

```
try {
    foo();
    bar();
    baz();
} catch (MyException e1) {
    // execute corrective actions;
} catch (AnotherException e2) {
    // execute other corrective actions;
}
```

When an exception occurs inside *foo*, *bar,* or *baz*, they throw a dedicated exception, which can be caught by their caller. The catch clause can then be used to implement some corrective actions that are specific to the exception which has been detected. As shown above, the names of the exceptions are not arbitrary, and the programmer can assign meaningful names to them. Moreover, when using an exception handling mechanism, the normal flow of the program will not get mixed up with the exception handling code, and hence this mechanism is more easily understood and easier to maintain. In the following section, Java's exception handling mechanism will be elaborated upon and its advantages will be discussed.

Exception Handling with Java's Exceptions-Handling Mechanism

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 that 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 *Exception* class. *Exception* is used as a base class for serious problems, such as the *OutOfMemory* exception, which are not usually recoverable. *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 vir-

tual machine (JVM: the java runtime engine) itself. Exceptions are intended to be extended and thrown by programmers to represent abnormal conditions in the program that require special handling. Programmers can use predefined exceptions defined in the Java language (e.g., *IllegalArgumentException*) 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 enables a textual message to be attached to the exception which is thrown. 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 which he or she implements, if appropriate. In Java, the exception which is 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 occured)
      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* clause that catches the exception which has been thrown. If the appropriate *try-catch* clause is not found in the running method, JVM uses the program stack in order 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 one which declares on a potential exception throw, it must either surround the method call with a *try-catch* clause or, instead, declare on the caller's signature that it may throw the specified exception. The programmer usually handles exceptions when he or she can do something about it. Otherwise, he or she defers exception handling up to the caller, who also may handle the exception or defer it further.

In order to further increase readability, the programmer can surround several methods by the *try-catch clause* 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* clause used in Java:

```
try {
   foo();
   bar();
   baz();
} catch (MyException my) {
```

```
    // execute corrective actions
} catch (OtherException othre) {
    // execute corrective actions
} finally {
    // execute some other commands
}
```

In the code segment given above, *foo*, *bar,* and *baz* 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. She or he 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, e.g., 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
}
```

When *ExceptionX* or *ExceptionY* are 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 it 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 summarise, the exception handling mechanism in general uses meaningful names for 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 enables the programmer either to deal with each exception separately or to 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.

Difficulties in Applying Exception Handling

Exception handling is perceived as a rather difficult task by novice programmers (Madden & Chambers, 2002). Providing a proper exception handling solution for a given task can be a challenging mission. Identifying, declaring, and handling all of the potential exceptions that a pro-

gram should be able to deal with is time consuming and hence the programmer's productivity is impaired. Exception handling may result in either the termination of the program or in a corrective action being performed. In order to correct a problem using the *try-catch* block, it should be located inside a loop which will increase the complexity of the code (Cabral & Marques, 2007). 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 structures. In order to assist programmers in designing and implementing exception handling, a visualisation tool has been suggested (Shah, Görg, & Harrold, 2008).

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 individualised 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 novice or experienced programmers.

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 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 Israeli universities and colleges reveled that they usually exemplify error-handling on input validation by printing an error message to the console and terminating the program as soon as an error occurs (local reaction inside a method). By the end of the course (or sometimes in a successive course), the students have studied the exception handling mechanism, but the approach of printing and terminating the program has already been established. Consequently, most of the students do not utilise exception handling mechanisms if they are not specifically instructed to do so, and use them poorly when they do. Students find exception handling to be one of the more difficult topics to learn (Manila, 2006).

Mapping Levels of Understanding – The SOLO Taxonomy

In the research literature, there are several taxonomies by which learning processes and levels of understanding are classified (Bloom, 1956). Biggs and Collis (1982) developed a system for classifying the quality of students' work, known as the SOLO taxonomy. The main advantage of the SOLO taxonomy, in relation to other educational hierarchies, is its generality: it is not content-dependent, making it useable across a number of subject areas. The SOLO taxonomy has five levels of understanding that can be encountered in learners' responses to academic tasks (Biggs, 1996):

1. Prestructural — the task is not accessed appropriately, and/or the student has not understood the task;

2. Unistructural — one or several aspects of the task are picked up and used (level of understanding is nominal);

3. Multistructural — several aspects of the task are learned but are treated separately. The student still lacks the "full picture" (understanding is equivalent to knowing about);

4. Relational — the task's components are integrated into a coherent whole, with each part contributing to the overall meaning (understanding in the form of appreciating relationships);

5. Extended abstract — the integrated whole at the relational level is reconceptualised at a higher level of abstraction, which enables it to be generalised to a new topic or area. The integrated whole derived at the previous level is conceptualised at a more abstract level so that it can be used in different settings (understanding in the form of transferring concepts and involving metacognition).The SOLO taxonomy has been used fruitfully both to classify students' work and to identify approaches used in the area of learning course material in post-secondary school settings. For these reasons, this research utilised the SOLO taxonomy to assess students' levels of learning. We used the SOLO taxonomy due to the objective criteria that it provides for measuring students' cognitive abilities (Chick, 1998). Students' knowledge and understanding of class hierarchies and interface classes was accrued incrementally, in a similar way to the measures in the taxonomy.

The Study

During many years of teaching Java programming, we tackled various solutions and strategies used by students concerning exception handling. Due to the importance of suitable exception handling strategies within computer programs, we conducted a study which aimed to examine students' strategies and characteristics concerning this issue. We focused on students rather than industry practitioners in order to explore the impact of the educational process (i.e., the level to which students assimilate the exception handling mechanism in general) on students' implementation of this mechanism before they are influenced by other sources. We were interested in getting insights concerning the various solution strategies used by the study participants when coping with exceptions. Hence we found the qualitative research methods to be the most appropriate for the aims of the present study.

Environment and Population

The data were collected during the academic years 2008-2010. The study subjects were third (and final) year students on a BA degree course in the Management Information Systems Department in an academic college in Israel. Fifty-six students participated in the research, who had all graduated from the following programming courses: "Introduction to programming", "Object oriented programming", and "Data structures and algorithms". In all of these courses, the students were provided with problems which necessitated the implementation of Java's exception handling mechanism.

In order to bring the participants to invest their best efforts in providing the solution, we decided to perform one-on-one interviews. The structure of an interview was as follows. First, the students solved the problem they were given. While solving the problem, the student could ask for clarifications. After solving the problem, the student had to choose from various solutions presented to him or her, justify his or her choice, and explain the differences between this solution and the one the student had provided beforehand. Then, the interviewer asked the student questions concerning various constituent parts of her solution. This method of data collection enabled the receiving of insightful spectrum of the student perceptions and opinions on error handling.

Fifty-six one-on-one interviews were conducted. All of the interviews were conducted by one of the researchers. A typical interview lasted for approximately one hour. During the interview, the students were not allowed to use any supplementary material and had to rely on their previous knowledge. However, if a participant could not remember the exact syntax of a mechanism, he or she could use a pseudo code or ask for the interviewer's assistance.

Research Tools

The problem which was given to the participants and the analytical methods used to analyse the students' exception handling strategies are presented in the following section.

The problem

In order to examine the students' assimilation of the exception handling mechanism in Java, they were asked to address the following problem. The problem that was presented to the students included a description of a large software product which was intended to be distributed to many customers who were then required to customise the software according to their specific needs. These settings were meant to motivate the students to provide a comprehensive, clear, modular, and extensible solution. The problem was as follows:

An industrial machine that performs operational tasks must keep the temperature (t) of its engine within the range [70-75] and the pressure (p) inside within the range [110-120]. Whenever these ranges are exceeded, the machine stops and raises the alarm. The human operator can react in any of the following ways: (1) terminate the current task; (2) open a valve (in order to release pressure) for *sec* seconds; (3) wait for *sec* seconds while doing nothing; or (4) run the machine for the time left to finish the current task. The specific action that takes place when a range violation occurs depends on the specific task being performed and may change accordingly. Corrective action can comprise a combination of several actions, such as opening the valve for 60 seconds, then waiting for 30 seconds, and then re-running the machine for the remaining time.

A computerised control has now been developed by the manufacturer for the machine in order to replace the human operators, and each customer that uses the machine will be able to program it using the Java programming language. A software package will be supplied to the customers containing the *Machine* class which represents the industrial machine. A summary of *Machine*'s methods is presented in Table 1.

Table 1: Application Programming Interface (API) of *Machine* class

Method	*Description*
Machine()	A constructor used to contact the machine equipment
createTask(int sec)	Create a new task with a specified operating time
executeTask()	Run the machine and continuously monitor temperature and pressure. Return upon task completion or upon unusual temperature or pressure (in that case, the remaining time > 0).
terminateTask()	Terminate the current task by setting the remaining time to zero
openValve(int sec)	Open the valve to release pressure for *sec* seconds
wait(int sec)	Wait for *sec* seconds while doing nothing
isDone()	Returns true if time left is zero, false otherwise

The code of all *Machine* methods is not available; however, the *Machine* class is accompanied by the following program with which the customer should encode the operation plan. The following is a simplified example of an operation plan in which the machine is operated for one minute, assuming that the operation is valid (i.e., not featuring unusual pressure or temperature):

```
import Machine;
public class Controller {
    public static void main(String[] args) {
        Machine machine = new Machine(); // contact the machine
        Machine.createTask(60); // initiate a task of 60 seconds
        // execute the task until it is done
        do {
```

```
            machine.executeTask(); // execute the task
        } while ( !machine.isDone() ); // continue if time left > 0
    }
}
```

The manufacturer of the machine would like to develop an exception handling mechanism to allow the customer to handle illegal values of temperature and pressure. You are requested to assist the manufacturer to construct a mechanism that allows the customer to add their own policies concerning temperature and pressure violations to the controller outlined above. Although the code of the *Machine* class is unavailable, the manufacturer is willing to modify it according to your recommendations, if you find it to be useful. The manufacturer plans to provide future versions of the automatic controller with extended functionality, hence the solution should be easy to maintain, clear, modular, and extensible.

In order to demonstrate your mechanism, you are requested to write two controller programs (representing possible customer operation plans) for the following cases, presented in Tables 2 and 3.

Table 2: Customer-1 operation plan

Operation time	Temperature violations	Pressure violations
30 minutes	t<70: wait 180 sec, re-execute task	p<100: open valve 90 sec, re-execute task
	t>75: wait 180 sec, re-execute task	p>110: terminate task

Table 3: Customer-2 operation plan

Operation time	Temperature violations	Pressure violations
60 minutes	t<70: terminate task	p<100: wait 60 sec, re-execute task
	t>75: wait 40 sec, re-execute task	p>110: open valve 50 sec, re-execute task

Methods of analysis

For the analysis process, we chose two analytical tools: our interpretation of the SOLO taxonomy (Biggs, 1996) and a selected subset of source code quality criteria (Boehm, Brown & Lipow., 1976). The problem stated in the previous subsection has various possible solutions, each with its own merits and weaknesses. We analysed and classified these solutions and ranked them according to the source code quality criteria (Boehm et al., 1976). We used the SOLO taxonomy to evaluate the students' assimilation of the exception handling mechanism in Java that was exhibited in each solution. The problem was built in such a way that the best solution would demonstrate the highest level of assimilation of the exception handling mechanism and the worst solution would demonstrate the lowest level of assimilation.

Table 4 presents the subset of criteria taken from Boehm et al. (1976) regarding source code quality that was used to evaluate the extent to which each level of assimilation met the specified requirements which appeared in the problem description concerning understandability, modularity, efficiency, and conciseness. We selected three criteria to address the maintainability requirements stated in the problem. We have adapted the criteria's description to the context of exception handling.

Table 4: Evaluation criteria

Criterion	Description
Clarity (C)	By clarity, we mean that the program contains meaningful names, and does not contain large number of control paths, making it simple and easy to follow. For instance, for the sake of fluency, the normal flow of code should be separated from exception handling code.
Modularity (M)	By modularity, we mean that the program is comprised of building blocks, each responsible for a certain function. Each building block is built from smaller building blocks, down to the basic functions, enabling modifications at the appropriate locations without undesirable side effects. For instance, minimisation of excessive or redundant code is desirable.
Extensibility (E)	By extensibility, we mean that the program can be easily extended to address new requirements. For instance, future versions of the software package may include bug fixes and new functions (e.g. more exceptions can be detected and highlighted), and these versions can be easily adapted by customers.

Based on the SOLO taxonomy, we defined five categories relating to the level of assimilation in the design and implantation of an exception handling mechanism in the solution:

1. First level of assimilation - The student handles the exception locally within the method (where the exception is detected) without the use of an exception handling mechanism or uses error codes to notify its caller. Although such a solution might work, it is undesirable as it does not use an exception handling mechanism at all, and it does not address the maintainability requirements of the problem. This category fits into the prestructural level in the SOLO taxonomy (elaboration in subsections S1 – S6).

2. Second level of assimilation – The student uses the Java exception mechanism in a simplified manner; namely, the student separates the normal flow of code from the exception handling mechanism using one exception class (e.g., *java.lang.Exception)* for all kinds of exceptions. However, this solution lacks the ability to declare specific exceptions raised in the program. Although this solution is better than the previous one, as it involves separating the normal from the exceptional flow, it still does not properly address all of the maintainability requirements. This kind of solution makes it hard to track the specific type of the exception when it is caught. Therefore, it is hard to write a distinct solution to each exception. This category fits into the unistructural level in the SOLO taxonomy (elaboration in subsection S4);

3. Third level of assimilation – The student defines specific exceptions, referring to the exceptions stemming from the problem. These user-defined exceptions directly inherit the Java *Exception* class. The hierarchy of exceptions which is provided is simplified, as it does not allow multiple exceptions to be handled together, and it forces the programmer to catch and handle each exception separately or to handle them all together (using one catch block). This category fits into the multistructural level in the SOLO taxonomy (elaboration in subsection S3);

4. Fourth level of assimilation - The student constructs an adequate exceptions class hierarchy which addresses only the requirements of the problem, including more levels of abstraction, and uses it properly. However, the student does not provide a refined solution addressing future extensions of the problem's requirements. This category fits into the relational-structural level of the SOLO taxonomy (elaboration in subsection S2);

5. Fifth level of assimilation - The student constructs an adequate hierarchy of exceptions, including more levels of abstraction, and uses it properly. The student provides a refined solu-

tion addressing future extensions of the problem's requirements, not just stemming directly from the current requirements. This category fits into the extended-abstract level in the SOLO taxonomy (elaboration in subsection S1).

Results and Discussion

In this section, we will present the various strategies provided by the study participants, followed by a discussion based on an examination of these solutions according to the criteria outlined above and an adequate level of abstraction. It is worth noting that a thread-based solution could address the problem with some additional benefits. However, as the participants were not familiar with this option, we will not present it herein.

Strategies

In the following subsections, we will show how the students' solutions were classified according to the following categories: refined multi-level hierarchy of exceptions (S1); multi-level hierarchy of exceptions (S2); single level hierarchy of exceptions (S3); Java exceptions (S4); delegating the task of exception handling to the caller using error codes (S5); and handling exceptions locally (S6).

S1: Refined multi-level hierarchy of exceptions

In this solution, the participants defined a three-level hierarchy of exceptions, as shown in Figure 1. This solution defines a proper hierarchy of exceptions according to inheritance guidelines (Meyer, 1997). The exception *MachineOperationException* is derived from Java *Exception.* The exceptions *PressureException* and *TemperatureException* are derived from *MachineOperation-Exception*. *LowPressure* and *HighPressure* are derived from *PressureException*. *LowTemperature* and *HighTemperature* are derived from *TemperatureException*.

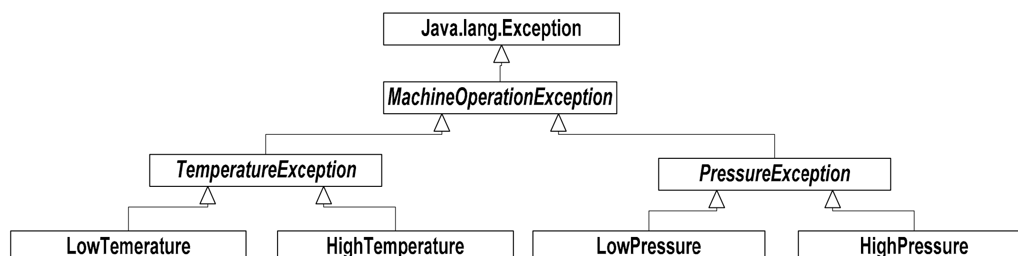

Figure 1: Refined multi-level class hierarchy of exceptions

From the policies presented in Tables 2 and 3, one can see that customer-1 uses an identical policy for handling both *LowTemperature* and *HighTemperature*, while customer-2 uses different policies for each. In order to avoid duplicate code in each 'catch' clause, it is desirable to create a base exception (*TemperatureException*) for the *LowTemperature* and the *HighTemperature* exceptions, enabling both exceptions to be caught together or separately, as desired. As for pressure, one can see in Tables 2 and 3 that both customers use different exception handling policies for *LowPressure* and *HighPressure* exceptions. Hence, a common base class for these exceptions is not mandatory, and these exceptions could directly extend *MachineOperationException*. However, doing so would hide the fact that both exceptions are related to a common concept of "pressure constraint violation". Other customers may use an identical reaction policy for both, and so, similarly to the solution provided for the temperature-related exceptions, a common base class for *LowPressure* and *HighPressure* is provided, *PressureException*.

In this solution, the *executeTask* method, which is the manufacturer's responsibility, should be used to continuously monitor the pressure and temperature in the machine and to throw concrete exceptions (the bottom level of the hierarchy) in the case of unusual data. However, its signature does not have to declare the throwing of all four concrete exceptions. Instead, it can declare the following:

void executeTask() throws MachineOperationException

This signature declares only one abstract exception instead of four concrete ones, ensuring that the solution is clear and extensible (i.e., future versions can add other exceptions derived from *MachineOperationException*).

The following is an example of a controller code which presents this kind of solution and implements customer-1's policies (Table 2), as given by the participants who provided this kind of solution.

```
public class Controller {
    public static void main(String[] args) {
        Machine machine = new Machine(); // contact the machine
        Machine.createTask(108000); // initiate a task of 30 minutes

        do{
            try {
                machine.executeTask(); // execute the task
            } catch(TemperatureException e){
                    machine.wait(180);
            } catch(LowPressure e) {
                    machine.openValve(90);
            } catch(HighPressure e) {
                    machine.terminateTask();
            }
        } while ( !machine.isDone() );
    }
}
```

Quality analysis: An analysis of this strategy, based on the evaluation criteria outlined in the analytical methods section, reveals the following: (C) This solution meets the clarity criterion. It is simple and easy to follow. Each exception is represented by a unique exception with a meaningful name; the separation of the discovery of an exception from its handling contributes to the fluency of the code. (M) Each exception can be caught and handled individually. In addition, the use of intermediate level exceptions enables several exceptions to be handled at the same time (in one catch block), avoiding the repetition of code. (E) The *Machine* class (the manufacturer's code) and the *Controller* class (the customer's code) are independent components. Each can be modified separately without undesirable side effects. As long as new versions of the *Machine* class provide an interface which is compatible with the previous version, the *Controller* class will continue to work properly.

Hierarchy analysis: In this solution, the students constructed an adequate class hierarchy of exceptions, including the required levels of assimilation, and used it properly. Moreover, they added the intermediate level of *PressureException*, despite the fact that the production plans provided to them (Tables 2 and 3) did not necessitate its definition. This solution may fit into the fifth level of assimilation in our SOLO-based taxonomy.

Frequency and analysis: Only three of the participants in this study provided this solution. This might be explained by the fact that in order to provide such a solution, the student has to exhibit a high level of assimilation of the exception handling mechanism. In addition, to be able to construct an adequate class hierarchy, the student must demonstrate a high level of abstraction. Previous research has shown that few students are able to do this (Lavy, Rashkovits, & Kouris, 2009; Or-Bach & Lavy, 2004).

S2: Multi-level hierarchy of exceptions

In this solution, the participants defined a class hierarchy of exceptions in which *LowTemperature* and *HighTemperature* are derived from the intermediate-level exception *TemperatureException*, as in S1. However, the *LowPressure* and *HighPressure* exceptions are derived from *MachineOperationException,* as shown in Figure 2.

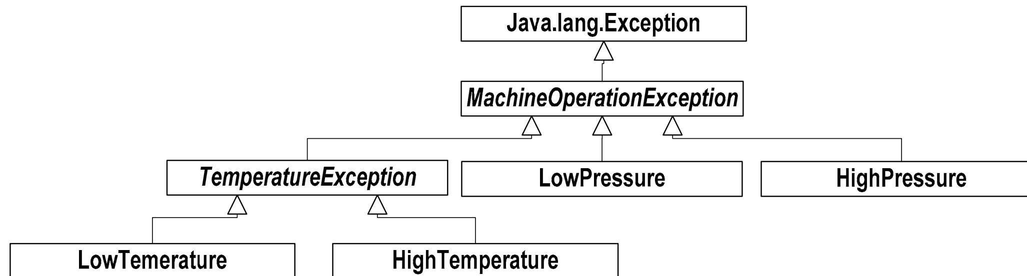

Figure 2: Multi-level class hierarchy of exceptions

The code segment demonstrating the controller in S1 can also be used to demonstrate S2.

Quality analysis: An analysis of this solution based on the evaluation criteria will be similar to that of S1, except that its modularity decreases. Namely, both "pressure" exceptions which are related to a common concept of a "pressure constraint violation" do not share a dedicated intermediate-level exception, meaning that customers who wish to react identically to both low and high pressure would have to catch two different exceptions and handle each one separately, producing excessive and redundant code.

Hierarchy analysis: In this solution, the participants constructed an adequate class hierarchy of exceptions which addresses only the requirements of the problem. However, they did not provide a refined solution addressing other customers' policies concerning pressure violations. This solution may fit into the forth level of assimilation in our SOLO-based taxonomy. Five participants provided this solution. This might be explained by the fact that students find it difficult to design mechanisms for intangible policies and are used to solving only the problem in hand.

S3: Single level hierarchy of exceptions

In this solution, the students defined a hierarchy of exceptions, each one representing a specific exception. Unlike S1, all of the exception classes are derived directly from the Java *MachineOperationException* class, without an intermediate level of refinement, as shown in Figure 3.

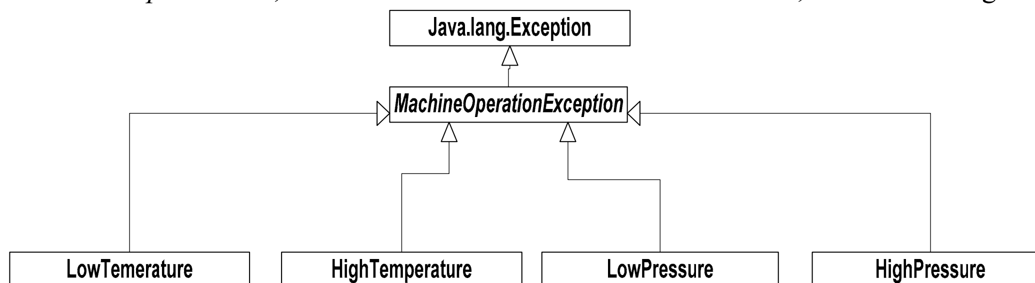

Figure 3: Single-level class hierarchy of exceptions

The following is an example of a code segment presenting this kind of solution and implementing customer-1's policy, as provided by the study participants:

```
public class Controller {
   public static void main(String[] args) {
```

```
        Machine machine = new Machine(); // contact the machine
        Machine.createTask(108000); // initiate a task of 30 minutes

        do{
            try {
                machine.executeTask(); // execute the task
        } catch(LowTemperature e){
                machine.wait(180);
        } catch(HighTemperature e){
                machine.wait(180);
        } catch(LowPressure e) {
                machine.openValve(90);
        } catch(HighPressure e) {
                machine.terminateTask();
        }
    }
}
```

The exception handling code for "low-temperature" and "high-temperature" is identical. However, as no common base class exists to represent temperature exceptions, the code must be duplicated.

Quality analysis: An analysis of this strategy according to the evaluation criteria will be similar to that of S2, except that its modularity is weaker. The lack of intermediate-level exceptions for both pressure and temperature violations forces the controller to catch and handle each exception separately, even in cases where a common reaction would be desirable.

Hierarchy analysis: In this solution, the participants defined specific exceptions by referring to the exceptions that stemmed from the problem. These user-defined exceptions inherited the *MachineOperationException* class. The class hierarchy of exceptions which was provided is somewhat simplified, as it does not allow multiple exceptions to be handled together, and it forces the programmer to catch and handle each exception separately or to handle them all together by catching *MachineOperationException*. Hence, this solution may fit into the third level of assimilation in our SOLO-based taxonomy.

Frequency and analysis: Ten study participants provided this kind of solution. This might be explained by the fact that most students care primarily about the correctness of the solution and neglect issues such as modularity.

S4: Java exception

In this strategy, the *executeTask* method detects exception situations and raises exceptions when appropriate. However, no special exception class was built for each type of exception. Instead, a general Java *Exception* is raised with an appropriate message attached to it, describing the exception. The signature of the *executeTask* method is as follows: *void executeTask() throws Exception.*

The following is an example of a code segment presenting this kind of solution and implementing customer-1's policies, as provided by the study participants:

```
public class Controller {
    public static void main(String[] args) {
        Machine machine = new Machine(); // contact the machine
        Machine.createTask(108000); // initiate a task of 30 minutes

        do{
            try {
                machine.executeTask(); // execute the task
            } catch(Exception e) {
                String msg = e.getMessage();
                if (msg.equals("low-temperature")
                    || msg.equals("high-temperature")) {
                    machine.wait(180);
                }
                else if (msg.equals("low-pressure")) {
                    machine.openValve(90);
                }
                else if (msg.equals("high-pressure")) {
                    machine.terminateTask();
```

```
                }
            }
        } while ( !machine.isDone() );
    }
}
```

Quality analysis: An analysis of this solution based on the evaluation criteria reveals the following: (C) In this solution, the exception handling code is separated from the normal flow, but it is far too complex, as the specific type of exception being raised is unknown. All of the exceptions are caught and handled in the same catch block, and the handler must explore (using many if/else statements) the accompanying message in order to identify the exception and execute the appropriate reaction. As a result, the clarity becomes blurred. Moreover, the *executeTask* method declares an *Exception* that is unspecific and is therefore less clear. (M) As the potential reactions to each exception are coded in the same *catch* block, the customer may have difficulties in modifying an existing policy according to new rules without harming other reactions. Moreover, if the customer makes a mistake in parsing (e.g., omitting a space letter), she or he might mishandle some exceptions. Therefore, this solution is not modular. (E) The *Controller* class relies on the actual text of the accompanying message to identify the specific exception. If the *Machine* class would change the text in future versions, all the customer's controllers would need to change accordingly. As a result, extensibility is damaged.

Hierarchy analysis: In this solution, the students used the Java exception handling mechanism in a simplified manner. In other words, they separated the normal flow of the code from exception handling by using the *Exception* class for all types of exceptions. However, this solution lacks the ability to declare specific exceptions when referring to the specific violation reported by the called method. This kind of solution makes it difficult to track the specific exception type when an exception is caught, since all exceptions are caught as *Exception*, and one has to parse their message to identify the type. Such a solution was also found in the work of Cabral and Marques (2007), and therefore it is difficult to provide a distinct solution for each exception. Hence, this solution may fit into the second level of assimilation in our SOLO-based taxonomy.

Frequency and analysis: A total of 15 participants provided this kind of solution. This might be explained by the way in which this exception mechanism has been taught. Namely, when the throw-catch mechanism was taught, *Exception* was caught and handled using error-print and program termination. This kind of example encourages the use of a single-catch clause that catches all exceptions, prints the attached message, and exits the program regardless of the exception in hand. Although later on the students are taught to declare and use user-defined exceptions, they sometimes adapt this simplified exception handling mechanism.

S5: Error codes

This solution does not use an exception handling mechanism. Instead, it uses arbitrary error codes which are returned by the *executeTask* method to notify the caller of the various exceptions. In this solution, the controller program examines the value returned from the *executeTask* method in order to validate its termination. If the return code is equal to a certain predefined exception code, the controller program applies the appropriate corrective actions. The following is an example of a code segment which presents this kind of solution and implements customer-1's policies, as provided by the study participants:

```
public class Controller {
    public static void main(String[] args) {
        Machine machine = new Machine();    // contact the machine
        Machine.createTask(108000); // initiate a task of 30 minutes

        do{
            int code = machine.executeTask(); // execute the task
            if (code != 0) //an exception occurred
                switch (code) {
```

```
                    case -1:   // low-temperature
                    case -2: // high-temperature
                        machine.wait(180);
                        break;
                    case -3: // low-pressure
                        machine.openValve(90);
                        break;
                    case –4: // high pressure
                        machine.terminateTask();
                        break;
                }
            }
        } while ( !machine.isDone() );
    }
}
```

Quality analysis: An analysis of this strategy based on the evaluation criteria reveals the following: (C) In this solution, the exception handling code and the normal flow are mixed together. This solution must examine the exception code returned by the *executeTask* method. Many redundant if/else statements are executed until the desired reaction is found, and hence this solution is not clear. (M) The exception handling code segment is located in the same *switch* block, and hence modifying one reaction policy may result in software bugs in other reactions. Moreover, if the customer gets confused (e.g., by considering -1 to represent a high temperature exception instead of a low temperature exception), she or he might mishandle some exceptions. Therefore, this solution is not modular. (E) The *Controller* class depends on the actual code returned from *executeTask*. If the *Machine* class changes these error codes in future versions, all of the customer's controllers would need to change accordingly. As a result, its extensibility is damaged.

Hierarchy analysis: The students handled the exceptions in the *Controller* class by using arbitrary error codes instead of exceptions. Although such a solution might work, it is undesirable as it incorporates the normal flow of the code with the exception handling code, and thus makes the controller program difficult to understand, complex and non-efficient. Such a code is difficult to design, develop, test and maintain. This solution may fit into the first level of assimilation in our SOLO-based taxonomy. Therefore, we may say that the students who used this strategy are in the prestructural level.

Frequency and analysis: Four study participants provided this kind of solution. This might be explained by the learning order of programming building blocks. Namely, exception mechanisms constitute an advanced subject which is usually taught at the end of an object-oriented programming course, forcing the students to handle exceptions by using exception codes. Moreover, most of the examples of code which students are given are related to specific contexts and are not intended for reuse in other contexts. Therefore, students get used to handling exceptions in such a manner and have difficulties in adapting to exception mechanisms.

S6: Handling exceptions locally

This solution does not use an exception mechanism. Instead, it mixes exception handling code with the normal flow inside the *executeTask* method. This *executeTask* method does not produce an exception code. Instead, it includes many if/else statements to detect exception situations and to handle them immediately, without notifying the caller. In this solution, the students left the simplified *Controller* code which was presented to them unchanged.

The following is an example of a code segment which presents this kind of solution for the *executeTask* method, implementing customer-1's policies, as provided by the study participants:

```
public class Machine {
    ...
    public static void executeTask() {
        ...
        if (getTemperature()<70 || getTemperature()>75) {
```

```
            machine.wait(180);
        }
        else if (getPressure()<100) {
            machine.openValve(90);
        }
        else if (getPressure()>110) {
            machine.terminateTask();
        }
        ...
    }
}
```

Quality analysis: An analysis of this strategy based on the evaluation criteria reveals the following: (C) As in the previous solution, the exception handling code and the normal flow are mixed together. Therefore, this solution is not clear. (M) The exception handling code segment is located in the same block as the normal flow. There is no separation between these functionalities, and hence modularity is not achieved. Modification to a policy might result in a serious bug in the operation of the machine. Therefore, this solution is not modular. (E) In this solution, the manufacturer must provide a unique *Machine* class to each customer, and whenever a customer wishes to modify the machine policy he or she has to alter that class. The manufacturer cannot distribute a new version of the *Machine* class unless it adapts each customer's policies. Therefore it is not extensible.

Hierarchy analysis: The students handled the exception locally within the *executeTask* method without the use of an exception handling mechanism. Similarly to the previous solution (S5), such a solution might work, but it is undesirable for the same reasons. The only difference between solutions S5 and S6 is the location of the complex code: while S5 handles exceptions within the controller program, S6 does it inside *executeTask* which is part of the *Machine* class. Therefore, as with S5, This solution may fit into the first level of assimilation in our SOLO-based taxonomy.
Frequency and analysis: Nineteen students provided this kind of solution. The explanations are identical to those for S5.

Overview of the Results

A summary of the quality analysis of the six solutions provided is demonstrated in Table 5.

Table 5: Summary of source code quality criteria

Solution/criterion	*Clarity*	*Modularity*	*Extensibility*	*Level of assimilation of exception handling*	*No of students*
S1	✓	✓	✓	Fifth	3
S2	✓	✓̶	✓	Fourth	5
S3	✓	✓̿	✓	Third	10
S4	✓̶	-	✓̶	Second	15
S5	-	-	✓̿	First	4
S6	-	-	-	First	19

As shown in Table 5, the best solution (S1), which was of the highest quality according to the source code quality criteria, also demonstrated the highest level of assimilation of the exception handling mechanism. In the other solutions (S2-S6), there is a gradual decrease in both the source code quality and the level of assimilation of the exception handling mechanism.

Table 5 depicts the solutions' distribution among the study participants. The most interesting result is the low number of students who demonstrated good exception handling solutions (S1, S2). This is in line with the findings of Madden and Chambers (2002), who claimed that exception handling is perceived as a difficult task by novice programmers. Although the students were gra-

duates of three programming courses, and hence were familiar with the exception handling mechanism, almost half of them chose to ignore it and used old-style exception handling instead (S5, S6). This may imply that the students had difficulties in assimilating the exception handling mechanism. The relatively low number of students who provided S5 (error codes) stems from the fact that this solution is, in fact, a variation of S6. In both solutions, the programmer uses if/else structures in order to identify and handle the exceptions.

Students' Reflections on the Task

After the students had finished the task, we conducted an informal interview with each of the participants. We asked them to reflect on the problem solving process and asked questions concerning their exception handling within the solution they had provided. Using analytic induction (Goetz & LeCompte, 1984) and content analysis (Neuendorf, 2002), 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: misconceptions concerning code quality; misconceptions concerning exception handling; difficulties in understanding the exception handling mechanism; the perceived importance of exception handling; and a lack of programming experience. In the following section, we will elaborate on the students' reflections regarding each of these categories.

Misconceptions concerning code quality

Some of the students' reflections referred to misconceptions concerning code quality:

Ran: *"My solution addresses the problem. Does it matter how I wrote the program?"*

Ania: *"I wrote the solution to the problem as fast as I could. I was not occupied with future consequences on maintenance and therefore I did not separate the exceptions' detection from their handling. I did not think it was important."*

When the students were asked why they did not use the Java exception handling mechanism in their solutions, they answered in a similar way to Ran. During their studies, students often develop a habit of investing minimal effort in every task they are given. As the design and implementation of an exception handling mechanism was neither immediate nor trivial, they avoid using it, justifying their behaviour by saying that their solutions met the requirements of the problem. Ania's reflection strengthens Ran's arguments. The students' tendency to dedicate minimal time to solving a programming problem derives from their misassumption that this indicates superior programming abilities. The consequences of this misassumption result in them concentrating solely on the functional requirements and ignoring clarity, modularity, and extensibility, which are very important to a solution's quality. This is in line with the findings of Clancy (2004), who claimed that many students are concerned with minimising typing and see code modifications and extensions as an academic exercise.

Misconceptions concerning exception handling

Some of the students' reflections referred to difficulties in understanding the mechanism for throwing and catching exceptions:

Ruth: *"When I detect an exception that might arise in the program, the most proper place to handle it is in the same module where it is detected. I don't understand why I should separate the handling from the detection. Maybe this is why I do not use exceptions and why instead I use conditional structure".*

Ben: *"I know that duplicating code is undesirable, but sometimes I have no choice but to do so. For instance, when I must print the same exception message in multiple exceptions. I do not think that using exceptions can assist in avoiding such duplications. Anyway I do not know how."*

Ruth and Ben's reflections describe a common reaction among the students who did not use an exception handling mechanism in their solutions. This avoidance of using exceptions is a result of a lack of understanding of what exceptions are and what their advantages are over other techniques. As one student said, *"Exceptions are complex, and I can do without them, so why bother?"*

Difficulties in understanding the exception handling mechanism

Some of the students' reflections related to their conceptions concerning the exception handling mechanism:

Josef*: "The* java.lang.Exception *is the easiest to use, as it does not necessitate the definition of new exception classes. Instead, I use the message attribute attached to the* java.lang.Exception *to declare what the problem was. When I catch the exception, I can tell by the message content what the problem was."*

Dalia: *"When we learned the exception handling mechanism, the lecturer explained the hierarchy of the exceptions in Java, and demonstrated how to derive user-defined exceptions from* java.lang.Exception*. However, many of the examples presented by the lecturer to demonstrate throwing and catching exceptions were based on* java.lang.Exception*. Therefore, this idea was the first one that came to my mind. Maybe if I thought more about the problem, I would come up with a better solution."*

Yaron: *"To me, exceptions are concerned with severe errors such as a lack of memory, problems with privileges, I/O problems and the like. I do not think that exceptions are the proper way to handle violations of business rules."*

In many cases, even when the students decided to use the exception handling mechanism, they did not derive exceptions from *java.lang.Exception* as they should have done. Instead, they used the *java.lang.Exception* itself and used the message attribute to differentiate one exception from another. When they were asked to justify their choice of solution, they answered similarly to Josef and Dalia. The *java.lang.Exception* provided them with a basic exception handling mechanism, and they felt that this solution was good enough without extending it. Josef and Dalia's reflections imply a superficial assimilation of the exception handling mechanism. Yaron, on the other hand, pointed out another aspect. Throwing and catching exceptions is sometimes presented to the students in the context of a "system" (e.g., I/O exceptions), and hence students may get the wrong impression that this mechanism is designed purely for infrastructure errors.

The perceived importance of exception handling

Many of the students made statements similar to the following:

Dan: *"We studied exception handling late in the course. The lecturer showed us several simple examples in which exception handling was demonstrated. Then we had to implement these guidelines in one or two homework assignments**.** In the following year, we were not required to define new exceptions, although we caught and handled Java's exceptions such as ClassCastException, so we assumed that we could mange without it."*

Yael: *"When we first studied input validations, we used conditional statements and when the input was incorrect we printed out an error message and usually terminated the program immediately. We used this technique for quite a long time until the exception handling mechanism was*

presented as a better option. However, I did not understand the advantages of using exceptions, and I continue to handle errors in the old fashioned way."

Dan raised an important issue concerning the hidden message conveyed to the students by teaching them about the exception handling mechanism at the end of the semester. The time and the duration of learning about this subject teaches the students faulty concepts regarding its importance. While learning about this subject, the students are provided with simple examples in which the exception handling mechanism is demonstrated. Due to the superficiality of the examples, the advantages of the mechanism (hierarchy) are not obvious. Yael, on the other hand, raised another issue concerning learning habits. Working for a long period of time and using one way of thinking caused the students to become fixated on this way of working and, when presented with a better way, they prefer to keep on working in the old way.

Lack of programming experience

Some of the students' reflections addressed the reasons why they did not implement the exception handling mechanism in their solutions:

David: *"When I see a program that uses exceptions I can see its advantages. However, I did not use it in my solution as I'm not used to thinking that way yet. I hope in the future that I will gain more experience and be able to provide better solutions."*

Miriam: *"I think I understand why exceptions are better then other error-handling techniques. But I feel that I need more practice to gain confidence implementing it. I perceive exceptions as an advanced technique which requires a deep understanding which I have not yet gained."*

Jacob: *"Often, we were told to assume that the input given in the problem is correct, and that there's no need to validate it. Therefore, I got used to ignoring exceptions such as invalid values, and didn't practice it enough."*

David and Miriam justified their choice to avoid using exceptions by their lack of programming experience. The consequences of teaching exceptions as an isolated issue may be that the students lack practice and are unable to perceive the broader context. Jacob added another argument that concerns a learning method in which, in order to allow students to focus on an issue, they are told to neglect exceptional inputs. As a result, the students may acquire improper programming habits.

Conclusions

In this paper, we have presented and analysed college students' strategies concerning exception handling. The results which have been obtained reveal that the majority of students have difficulties in designing and implementing an appropriate exception handling mechanism. Although the students had been taught and had used exception handling mechanism, they did not properly utilise the advanced exception-handling mechanisms offered by modern programming languages. The students had difficulties in exhibiting high levels of assimilation concerning a proper design for a hierarchy of exceptions. These results are in line with previous research regarding the object-oriented design capabilities of novice programmers (Lavy et al., 2009; Or-Bach & Lavy, 2004; Sim & Wright, 2001).

The design and implementation of exception handling strategies requires previous knowledge of the inheritance mechanism and polymorphism. Consequently, this cannot be taught at the beginning of a programming course. However, the use of this mechanism via existing Java exceptions (such as *java.util.IllegalArgumentException*) does not require a knowledge of these advanced programming topics, and we believe that it should be taught earlier on in the course, along with conditional statements. This way, students will be able to develop better programming habits with regard to exception handling. As soon as an inheritance mechanism is studied, it is desirable to

apply it to exceptions in order to enable students to define exceptions of their own when appropriate. In addition, we believe that providing students with examples that require the design and implementation of a multi-level hierarchy of exceptions might help them to develop their abstraction abilities in general, and to use an exception handling mechanism in particular.

The classification of the solution strategies according to our SOLO-based five levels of assimilation may seem rigid; however it provided a coherent view of the solution spectrum which suits the common range of solutions requiring various levels of abstraction. Finally, we believe that further research with a large number of participants should be conducted in order to establish our results.

References

Biggs, J. B. (1996). Enhancing teaching through constructive alignment. *Higher Education, 32*(3), 347-364.

Biggs, J. B., & Collis, K. F. (1982). *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome).* New York: Academic Press.

Bloom, B. S. (1956). *Taxonomy of educational objectives, the classification of educational goals – Handbook I: Cognitive domain*. New York: McKay.

Boehm, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative evaluation of software quality. *Proceedings of the International Conference on Software Engineering, pages 592-605. IEEE Computer Society Press, October*. Los. Alamitos, CA

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.* Springer-Verlag, Berlin Heidenberg.

Chick, H. (1998). Cognition in the formal modes: Research mathematics and the SOLO taxonomy. *Mathematics Education Research Journal, 10*(2), 4-26.

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.

Coelho, R., Rashid, A., Garcia, A., Ferrari, F., Cacho, N., Kulesza, U., Staa, A., & Lucena, C. (2008). Assessing the impact of aspects on exception flows: An exploratory study. *Proceedings of the 22nd European Conference of Object-Oriented Programming* (pp. 207-234). Paphos, Cyprus: Springer.

Filho, F. C., Cacho, N., Figueiredo, E., Maranhao, R., Garcia, A., & Rubira, C. M. F. (2006). Exceptions and aspects: The devil is in the details. *Proceedings of the 14th ACM SIGSOFT International Symposium Foundations of Software Engineering* (pp. 152-162). Portland, OR, USA.

Filho, F. C., Garcia, A., & Rubira, C. M. F. (2007). Exception handling as an aspect. *Proceedings of the 2nd Workshop Best Practices in Applying Aspect-Oriented Software Development* (pp. 1-6). Vancouver, BC, Canada.

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.

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.

Lippert, M., & Lopes, C. V. (2000). A study on exception detection and handling using aspect-oriented programming. *Proceedings of the 22nd International Conference on Software Engineering* (pp. 418-427). Limerick, Ireland.

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.

Meyer, B. (1997). *Object-oriented software construction* (2nd ed). Englewood Cliffs, NJ: Prentice-Hall.

Mitchell, R., & McKim, J. (2002). *Design by contract: By example*. Redwood City, CA, USA: Addison Wesley Longman.

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.

Robillard, M. P., & Murphy, G. C. (1999). Analyzing exception flow in Java program. *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 (pp. 322–337) vol. 1687.* Springer-Verlag, New York, NY.

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, (pp 2-10), 25*(6). 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.* ACM Press, New York, NY.

Shah, H., Görg, C., & Harrold, M. J. (2008). Visualization of exception handling constructs to support program understanding. *Proceedings of the ACM Symposium on Software Visualization* (pp. 19–28). Munich, Germany.

Siedersleben, J. (2006). Errors and exceptions – Rights and obligations. In C. Dony et al. (Eds.), *Exception handling* (pp. 275-287). Springer-Verlag, Berlin Heidenberg.

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 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 department. 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 distributes systems as well as computer sciences education.

Ilana Lavy is a Senior Lecturer with tenure at the Academic College of Emek Yezreel since 2000 in the department of Management Information Systems department. 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 sixty papers and research reports (part of them in Hebrew).