Exception Flows made Explicit: An Exploratory Study

Nélio Cacho\(^1\), Francisco Dantas\(^2,3\), Alessandro Garcia\(^2\), Fernando Castor\(^4\)

\(^1\) School of Science and Technology, Federal University of Rio Grande do Norte, Brazil
\(^2\) Informatics Department, Opus Research Group, PUC-Rio, Brazil
\(^3\) Computing Department, State University of Rio Grande do Norte, Brazil
\(^4\) Informatics Center, Federal University of Pernambuco, Brazil

neliocacho@ect.ufrn.br, {afgarcia, fnetog}@inf.puc-rio.br, castor@cin.ufpe.br

Abstract. Most of the exceptions exert a global design impact as they tend to flow through multiple module interfaces of a software system. Exception handling mechanisms in programming languages were originally proposed to improve the structuring, comprehension and robustness of error handling code. These mechanisms are traditionally based on the fundamental assumption that global exception flows should be always implicit. However, it has been empirically found that the implementation of global exception handling in real-life software projects tends to exhibit poor quality. This paper presents an exploratory experiment to assess the benefits and drawbacks of explicit exception flows (or exception channels), as opposed to implicit exception flows. The experiment design involved 15 participants using three alternative mechanisms for exception handling. Our analysis was driven by key indicators of software usability: (i) implementation time, (ii) number of uncaught exceptions, and (iii) number of incorrect answers by the participants.

1. Introduction

Since its early days, exception handling [Goodenough 1975] is considered a powerful technique to structure the parts of a system responsible for handling errors. Exception handling mechanisms are aimed at promoting an explicit textual separation between normal and error handling code, in order to support the construction of programs that are more concise, reusable, evolvable, and reliable [Goodenough 1975]. An exception handling mechanism provides constructs to raise exceptions, which indicate the occurrence of errors, and to handle these exceptions so that, ideally, the error can be corrected and the system can resume its normal execution.

Even though a large number of programming languages implement exception handling mechanisms and many real world systems leverage this technique to structure their exceptional behavior, the expected benefits of exception handling have not yet been fully realized. It is often the case that exception handling code is of poor quality [Robillard and Murphy 2003, Shah et al. 2008], contributing negatively to the overall system reliability and understandability [Robillard and Murphy 2003]. To a large extent, this is a consequence of developers treating exception handling as an afterthought and
only worrying about the main system functionality [Shah et al. 2008]. However, the designs of existing exception handling mechanisms also play an important role in undermining system reliability. These mechanisms fail to capture and clarify the ways in which exceptions affect program execution and allow developers to commit mistakes that are difficult to track and correct.

Exception handling is known to be a global design issue [Garcia et al. 2001] that affects almost all the system modules [Lippert and Lopes 2000], mostly in an application-specific fashion. However, existing exception handling mechanisms are based on the implicit assumption that it is enough to specify the places where a program raises exceptions and the places where it handles them. The main consequence of this approach is that global exceptions introduce implicit control flows [Malayeri and Aldrich 2006a], i.e., if a programmer changes exception-related code, the control flow in apparently unrelated parts of the program may change in surprising ways [Robillard and Murphy 2000]. As a consequence, it becomes difficult to discover where the exceptions raised within a given context will be handled and to trace a handled exception to the place where it was originally raised.

We associate many of the problems of existing exception handling mechanisms with their inability to deal with error handling as a global issue. These mechanisms do not require developers to take exception control flow into account, keeping it implicit and, therefore, a source of potential bugs. This paper presents a study which aims to assess whether defining explicit exception control flows (or exception channels) can make exception handling mechanisms deliver on their original promise. More specifically, it is important to understand whether specifying and enforcing the ways in which exceptions should flow from their raising sites to their handling sites would: (i) accelerate software development activities, (ii) improve the understanding that developers have about their programs, and (iii) result in programs that have fewer bugs.

We conducted an exploratory experiment (Section 4) involving 15 participants and it encompassed the comparison of three different development approaches. Two of them are object-oriented and aspect-oriented implementations, respectively based on Java and AspectJ [Laddad 2003]. Even though they offer some different facilities for exception handling, both of them promote implicit exception control flows (Section 2.2). The third implementation is also aspect-oriented, but it is based on the EJFlow language [Cacho et al. 2008b] (Section 3) – the only innovative programming language of which we are aware that supports explicit exception channels. The gathered results (Section 5) were based on three key indicators of software usability. We also compare our findings with related work (Section 6) and provide some concluding remarks (Section 7).

2. Exception Control Flow and Conventional Programming Techniques

The purpose of this section is to provide basic concepts related to exception handling (Section 2.1) and presented the conventional means to express exception control flows (Section 2.2).
2.1. Exception Handling: Basic Concepts

Exception handling [Goodenough 1975] is a technique for structuring the error recovery code of a system so that errors can be more easily detected, signaled, and handled. Many mainstream programming languages, such as Java, Ada, C++, and C#, incorporate built-in exception handling mechanisms. These languages provide constructs to signal the occurrence of an error (raise or throw an exception) and to associate a set of recovery measures (exception handler) with the error.

When a part of a program raises an exception, the underlying exception handling mechanism is responsible for changing the normal control flow of the computation within the program to its exceptional control flow. Therefore, raising an exception results in the interruption of the normal program activity, followed by the search for an appropriate exception handler (or simply handler) to deal with the signaled exception. After the execution of an exception handler, control returns to the code that immediately follows the handler. An exception handling context (EHCs) is a delimited region in the program where exception handlers are attached. Exceptions raised within an EHC are treated by the associated handlers. For instance, try blocks in Java define EHCs and catch blocks define the respective associated handlers.

2.2. Implicit Exception Flows: Conventional Techniques

Implicit control flows are typically present in object-oriented programs creating exception-specific dependences among its methods, thereby negatively affecting software-engineering tasks such as code development and maintenance [Saurabh Sinha and Harrold 2004]. For instance, implicit control flow makes it difficult to explicitly analyse: (i) if an exception is going to be handled and where, (ii) which alternative handlers are bound to a specific exception, (iii) the list of intended handlers for an exception, and (iv) which components are affected by an exception control flow.

Some languages, e.g. Java, try to alleviate the implicit control flow problems by supporting the definition of an explicit exception interface. They require programmers to state the list of unhandled exceptions that each method signals to its clients, otherwise, the compiler will report an error. The main limitation of exception interfaces is that they do not make it possible to link the raising site of an exception to its handling site. The information provided by exception interfaces is local to each method, whereas exception control flow is inherently global. Hence, they indicate potential propagation paths, but not the one that an exception raised at a certain point in a program takes on its way to its intended exception handler. Moreover, even the limited help exception interfaces provide only applies to some of a program’s exceptions. For example, in Java, the rules for exception interfaces do not apply to the so-called unchecked exceptions. In addition, it is well-known that exception interfaces, as implemented in Java, have serious maintainability issues [van Dooren and Steegmans 2005a].

Aspect-oriented programming (AOP) techniques [Kiczales et al. 1997] emerge as a natural option to promote enhanced modularity and maintainability of programs in the presence of exceptions, since the exceptional behaviour of a system is an inherently crosscutting concern [Laddad 2003]. However, a recent study [Coelho et al. 2008] reported that aspects complicate exception control flow in ways that reduce the overall
system reliability. Languages such as AspectJ [Laddad 2003] promote the syntactic separation of normal and error handling code but fail to support a semantic separation. They only provide constructs for raising and handling exceptions and do not include means for developers to explicitly indicate and enforce, at compile time, the possible ways in which exceptions should flow from a raising site to a handling site. Moreover, AspectJ allows developers to suppress the checks that the Java compiler performs, pertaining to exceptions. As a consequence of these factors, it is easy to lose track of exceptions, which results in unplanned exception handling, unhandled exceptions, and unintentional exception raising [Coelho et al. 2008].

3. Explicit Exception Flows with EJFlow

EJFlow [Cacho et al. 2008b, Cacho et al. 2008a] is a novel exception handling mechanism to make it possible to explicitly define properties governing exception flows. This mechanism introduces two new concepts: explicit exception channels and pluggable handlers. An explicit exception channel (channel, for short) is an abstract duct through which exceptions flow from a raising site to a handling site. The raising sites are loci of computation where exceptions can be raised. The handling sites of an explicit exception channel are loci of computation where exceptions are handled, potentially being re-raised or resulting in the raising of new exceptions. In languages such as Java, both raising and handling sites are methods, the elements in the program from where exceptions are thrown and handled. A pluggable handler is an exception handler that can be associated with blocks, methods, classes and packages.

![Figure 1. Exception control flow among components of Mobile Media.](image)

EJFlow provides means for developers to define explicit exception channels and pluggable handlers using well-known AspectJ abstractions, namely, pointcuts, advice, and inter-type declarations. To better illustrate the use of EJFlow, Figure 1 describes partially the exception control flows in the Mobile Media application. MobileMedia [Figueiredo et al. 2008] is a mobile application that manipulates photo, music, and video on mobile devices, such as mobile phones. The application uses various technologies based on the Java ME platform, such as SMS, WMA and MMAPI.

Figure 1 presents five components: ImageUtil, ImageAccessor, AlbumData, BaseController, and PhotoViewScreen. The ellipses inside the
components represent methods. The exception handling behaviour is described by a black arrow from \(a\) to \(b\) which indicates that, during the execution of \(a\), an exception can be raised and this exception will be signalled to \(b\). Each arrow also indicates that control flow is passed from one method to the other. The figure explicitly indicates the types of exceptions detected and signalled by components.

### 3.1. Defining Explicit Exception Channels

Several different explicit exception channels can be defined for the application. EJFlow provides a new pointcut designator, \(echannel\), to support the definition of explicit exception channels. This pointcut designator takes a formal parameter, \(et\), consisting of the name of an exception type. It matches any join point where the raised exception is of type \(et\). In the EJFlow pointcut language, named pointcut expressions built up using the \(echannel\) designator are considered explicit exception channels, where the name of the pointcut expression represents the name of the channel. The second parameter of a channel definition identifies its raising site.

```
pointcut rSite1 : within(lancs.mobilemedia.core.ui.datamodel.*);
pointcut EEC1() : echannel(RecordStoreException+, rSite1);
pointcut rSite2 : withincode(public void getImageInfoFromBytes());
pointcut EEC2() : echannel( (IndexOutOfBoundsException) && (ArrayStoreException), rSite2);
```

The two examples above define the raising sites as separate pointcuts that the definitions of \(EEC1\) and \(EEC2\) use (Figure 1). The \(EEC1\) definition captures all exceptions that are subtypes of \(RecordStoreException\) and occur inside the model layer. The second channel \(EEC2\) captures occurrences of two runtime exceptions (\(IndexOutOfBoundsException\) and \(ArrayStoreException\)) raised within the body of method \(getImageInfoFromBytes\). It follows that as long as the exception type is defined using the \(echannel\) designator, EJFlow does not make any distinction between checked and runtime exceptions. Both of them are equally monitored by the defined channels.

### 3.2. Plugging Handlers to Exception Channels

In order to specify the handling site of an explicit exception channel, EJFlow provides the \(ehandler\) advice. This type of advice supports the implementation of pluggable handlers. It encapsulates the exception handling code that is executed when a certain point in an explicit exception channel is reached. Each \(ehandler\) advice consists of: (i) a set of parameters, like any other advice; (ii) a \(boundto\) clause specifying the explicit exception channel to which the handler is bound; (iii) an associated pointcut that determines the join points, within the channel, at which the advice executes; (iv) a \(catching\) clause that indicates the exception to be handled; and (v) a body, the actual handler implementation. The following code snippet presents a simple \(ehandler\) in the context of channel \(EEC1\):

```
void ehandler() boundto(EEC1()):
  withincode(public void showImage()) { ... }
```

This advice handles exceptions flowing through the channel \(EEC1\). The handler is activated when such exceptions are raised within method \(showImage\)().
3.3. Specifying Exception Interfaces of Channels

When a class or program cannot handle all the exceptions that flow through an explicit exception channel, it is necessary to declare these exceptions in the channel’s exception interface. There is an inter-type declaration in EJFlow, named declare einterface, which serves this purpose. The following code snippet illustrates typical examples of exception interfaces:

1: declare einterface: LibException echannel EEC2():
   within(lancs.mobilemedia.core.util.*);
2: declare einterface echannel EEC1():
   within(lancs.mobilemedia.core.ui.datamodel.*);

The first inter-type declaration (line 1) explicitly indicates that LibException exception is part of the exception interface of the channel. Alternatively, the second declaration (line 2) specifies only the explicit exception channel to which the exception interface is associated. This second format is more general. It states that every uncaught exception that flows through channel EEC1 is part of the channel’s exception interface.

The declare einterface clause avoids the need to specify the exception interface of each method that acts as a raising or intermediate site within a channel. Therefore, in the example of Figure 1, with the use of the two declarations above, exceptions LibException and RecordStoreException do not need to be declared in the throws clauses of methods getImageInfoFromBytes, loadImageBytesFromRMS, and loadSingleImageFromRMS. The compiler still performs the static checks and, therefore, the compiler issues an error message if one of these declarations is removed.

Figure 2 illustrates the utilization of some EJFlow abstractions necessary to describe the exception handling behaviour of the Mobile Media application (Figure 1). Notice that for each layer of the Mobile Media architecture, it is possible to see clearly where the explicit exception channels are defined and where they are handled. For instance, Lines 7-8 state that Model layer does not handle the explicit exception channel defined in Util layer. In addition, Model layer defines a new channel (EEC1) which is declared in the exception interface of the channel. Such a channel is handled in Control (Line 11) and View (Line 13) layers. In summary, Figure 2 shows that EJFlow model provides the means to specify, in a local manner (Figure 2), non-local
information pertaining to exception flows of an entire component or the software architecture implementation.

4. Evaluating Usuability of Exception Channels: Study Settings

Despite the preliminary modularity-specific advantages already perceived from EJFlow usage [Cacho et al. 2008b], there is no knowledge about the impact of exception channels on important usability aspects. There is a need to understand how EJFlow constructs eased or not the implementation of specific maintenance tasks. We have compared the implementation time and robustness of three different implementations of the same software system. We undertook a experimental study involving a group of fifteen novice students. Our experimental method was based on the exploratory evaluation method described by Soares [Soares 2004] and Walker et al [Walker et al. 1999]. This section describes the configuration of our study including the study subjects, experimental phases, and the hypotheses.

4.1. Target Systems

The initial decision in this study entailed the selection of the targeted applications. We have chosen two applications: HealthWatcher (HW) [Soares et al. 2002] and Mobile Media (MM) (Section 3). Both applications were selected because they met a number of relevant criteria for our intended evaluation. First, they are real and non-trivial systems with existing Java and AspectJ implementations. HW and MM systems are particularly interesting in terms of exceptional behaviour since the overwhelmingly majority of exceptions are checked and exception interfaces are widely used.

Second, these two applications involve a number of concerns and technologies common in day-to-day software development, such as GUI, persistence, concurrency, RMI, Servlets and JDBC. They were already implemented in Java and AspectJ. Third, each design and implementation choice for both OO and aspect-oriented (AO) solutions of MM and HW has been extensively discussed and evolved in a controlled manner. Both the OO and AO designs of the HW and MM systems were developed with modularity and changeability principles as main driving design criteria. Finally, the first HW release of the Java implementation was deployed in March 2001. Since then a number of incremental and perfective changes have been addressed in posterior HW releases. These changes have allowed us to observe the impact of typical types of exception handling modifications in such application domains.

4.2. Experiment Participants

Fifteen participants took part in this study, and they were last-year undergraduate students taking a course on advanced topics of programming languages. Their level of experience provided a good match for this study as we were interested in assessing the ability of EJFlow adopters to correctly grasp and use the notion of exception channels. The course was also specifically offered to perform the study. The participants were aware that they were participating of an experimental study and that their data would be used in the study analysis. In addition, they participants answered a questionnaire in order to characterize their expertise before the study. More information about their expertise is presented in Section 4.4.
4.3. Hypothesis

The definition of our hypotheses relies on some symbols used through the rest of this section. They denote the metrics to be collected and analysed: IT(Implementation time), NE (Number of exceptions uncaught) and WA (Number of questions incorrectly answered). IT is the amount of time spent by developers realising the tasks. NE quantifies the number of exceptions associated with the wrong handlers in the programs produced by the participants. WA was counted in terms of wrong answers to questions in a program comprehension activity discussed later.

Each of these metrics has three variations, each of them corresponding to Java (J), AspectJ (A), and EJFlow(E). For example, there is implementation time metric defined for the Java ($IT_J$), AspectJ ($IT_A$), and EJFlow ($IT_E$). The definitions of the following hypotheses use these symbols. The main hypothesis is the null hypothesis that states there is no difference in using either Java, AspectJ or EJFlow. Therefore, we expect the study findings will reject this hypothesis. There are three null hypotheses, one for each metric of interest. The following text presents first a composition of these three null hypotheses. In the following, alternative hypotheses are defined to be accepted when each of the corresponding null hypotheses is rejected.

**Null hypothesis** ($H_{01,3}$): The implementation time (1), the number of exceptions uncaught (2), and the number of questions incorrectly answered (3) using EJFlow is not different from using Java or AspectJ approaches.

**Alternative hypothesis** ($H_{11,3}$): The implementation time (1), the number of exceptions uncaught (2), and the number of questions incorrectly answered (3) using EJFlow is lower than using Java and AspectJ approaches.

4.4. Study Phases

This study was divided into three major phases: (1) preparation, (2) experimental procedures, and (3) data collection. In the phase three, in order to obtain the number of uncaught exceptions for each subject’s implementation, we performed an exception control flow analysis whose main goal was to identify the exceptions that were not handled and likely leading to system failure. We have used an “aspect-aware” static analysis tool [Coelho 2009] to derive exception control flow description of all 45 prototypes produced by the subjects. More details and extra information about this phase can be found in [Cacho 2008b]. A description of the first two phases follows.

**Preparation.** The first step before starting the study was to apply the questionnaire to the subjects in order to collect information about their experience. The student expertise was divided in six different categories and each one was matched with a score varying from 1 to 6. Students with experience only in the graduate course received 1 whereas those who have more than 6 years of experience gained 6 points. Other scores (2-5) represent intermediary categories. Figure 3 depicts a brief overview of the participants’ experience. It is interesting to notice that most of the subjects had their first contact with AspectJ in this course. On the other hand, all subjects used exception handling mechanisms in the past two years.
In the first half of the course we introduced and discussed the concepts of AOP, good exception handling practices, AspectJ, EJFlow, and specific design patterns used in the HW system. We provided exercises about AspectJ, EJFlow and RMI, in order to give the subjects knowledge about these technologies, in particular to the ones that did not have enough contact with them. We provided several documents to the participants, including the selected HealthWatcher’s class diagram and guidelines on how to use AspectJ and EJFlow. In addition, we conducted a pilot study where the students were asked to move try-catch blocks from the distribution-specific HW modules to aspects using both AspectJ and EJFlow approaches. The purpose of this pilot study was to train the participants in the programming environment and to address any doubt about how to use the AOP approaches under assessment.

<table>
<thead>
<tr>
<th>Task</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>The intent of this task was to assess whether or not is easy to change the handling site of existing exception control flows. This scenario corresponds to the second release of HW system.</td>
</tr>
<tr>
<td>Adapter</td>
<td>This task focus on identifying how subjects grasped the notion of exception channels and exception interfaces. This scenario corresponds to the fifth release of HW system.</td>
</tr>
<tr>
<td>Exception Handling Improvement</td>
<td>This task intends to assess how easy is to change the exception handling behaviour by adding more specific error recovery handlers. This scenario corresponds to the tenth release of HW system.</td>
</tr>
<tr>
<td>Comprehension of a short program</td>
<td>The last task aims to assess how easy is to understand the exception control flow looking only to a small part of the program.</td>
</tr>
</tbody>
</table>

Table 1. Study Tasks

**Experimental Method.** The study was executed during the second half of the course, according to the schedule in Table 1. The tasks were conducted in the order presented by the table. Subjects were randomly assigned to three groups, named Java, AspectJ and EJFlow. Each task, except for the last, consisted of the following steps. First, the experimenter introduced to the subjects the overall goal and format of the experiment. A printed copy of the implementation scenario with task descriptions was given to each subject. The experimenter then ensured that the subjects could use the programming environment to edit, compile and run the HW.
The subjects were given 4 hours to perform the tasks. The experimenter was present during this period and available to answer questions about the tasks and the programming environment. At the end of each scenario implementation, which was either when the subjects finished the task, or the end of time limit, the subjects were asked to fill out a questionnaire regarding the perceived usability of the used approach. Then, we collected the generated source code and the questionnaire. We have also recorded the time necessary for each subject to finish the task.

In order to undertake the last task we followed the experimental method defined by [Wiedenbeck et al. 1999]. As this task consisted in comparing the comprehension of the exception handling behaviour of a program, we have used a different application, called Mobile Media, for the assessment. In fact, we have used a simplified implementation of Mobile Media. We were very careful in order to make it sure that the three versions of the implementation were equivalent and relied on good representative solutions.

In this task subjects studied the simplified version of Mobile Media and answered questions about it in two trials. Fourteen questions about the exception handling behavior were developed. The questions were the same for Java, AspectJ and EJFlow implementations. Each question targeted a particular kind of information which might describe the understanding of the exception handling behaviour. Subjects kept the same assignment to the three groups defined for the previous tasks. This task consisted of a number of preparatory steps similar to the previous tasks. After receiving a hard copy listing the program, the subjects were allowed to study the listing of the program for 15 minutes. After the study period, the program description was removed. Then the subjects received the set of 14 comprehension questions to be answered in 5 minutes.

After the first question set, the subjects’ answer sheets were removed. Then they were asked to carry out another activity and were also given a few minutes to rest. The purpose of this interval was to discourage subjects from simply duplicating their previous answer from memory in the second trial. Again the program was returned to the subjects with a clean answer sheet. Subjects were given additional 15 minutes to answer the same questions set again, this time with the program made available to them.

5. Study Results

Tables 2 and 3 present each subject’s time (IT), in minutes, and the number of uncaught exceptions (NE) observed during the implementation of the first, second and third tasks, respectively. The difference (Diff) should be read as an increase or decrease in time with respect to the time spent using the Java approach. For example, Table 2 presents a decrease of 11% in the implementation time when using the AspectJ approach, and a decrease of 14.8% in the implementation time when using the EJFlow approach.

We can infer some interesting conclusions from the table’s data. In the first task, the subjects were required to move handling sites from a set of servlets based on the implementation of the Command design pattern [Gamma et al. 1995]. With the AO approach, the subjects performed this task faster than the subjects using the Java approach. In the AO implementation, it was only necessary to change some pointcuts to set up the new handling sites, whereas in Java the subjects needed to add new handlers.
Nevertheless, the subjects using the AO approach should carefully change the handling site in order to avoid the accidentally creation of uncaught exceptions.

Despite the small difference (3%) in the overall mean of time between both AO approaches, it is possible to see the importance of reliability checks supported by EJFlow in the reduction of uncaught exceptions. With EJFlow, subjects could only finish their tasks after addressing all the compilation errors related to uncaught exceptions issued by the compiler. By contrast, AspectJ subjects were not bothered by the compiler once since AspectJ does not perform exception control flows analysis to notify the number of uncaught exceptions. As consequence, the AspectJ results show that the faster the subjects performed the tasks, the higher the number of uncaught exceptions.

In the second task, the subjects were required to understand and change the exception flow enforcement. As described in Section 2, the knowledge about exception interfaces in Java and AspectJ is scattered through many methods, whereas EJFlow maintains this information localised in a few aspects. This explains why the EJFlow group was 20% faster than the Java and AspectJ approaches.

Finally the last task was the hardest one, which can be noticed by the mean values for amount of time spent with each approach. The subjects should understand the exception handling policy for the persistence mechanisms and introduce the implementation retrieval-based exception handling. Again, the EJFlow group performed respectively 27% and 16% faster than the Java and AspectJ groups. The reliability checks provided by EJFlow also ensured that all exceptions defined at raising sites were...
properly handled at the handling sites. This led to none exception being uncaught. Despite this difference between the performance of EJFlow and the other two approaches, the subjects assigned to the EJFlow group did not have superior programming skills, which can be confirmed by their academic expertise (Figures 3). This suggests that EJFlow abstractions are intuitive to use and effective to avoid uncaught exceptions.

The results of the comprehension tasks help to understand why the EJFlow group was faster than any other approach for the three tasks. Table 3 shows the number of wrong answers provided by the subjects during the fourth tasks. Notice that in the first trial, subjects answered the questions with the program hidden, but in the second trial subjects answered them with the program visible. Thus, scores from the first trial reflect the exception handling representation solely based on the study of the program. On the other hand, the second trial posed a focused task of verifying the proposition (defined in the question) in the program text. Thus, it measured the extent to which subjects were capable to trace the exception control flows and make inferences from them in order to answer the questions.

The Java subjects did not find it easy to trace back the exception control flow in the first trial. The EJFlow group, by contrast, had an overall mean of 28% fewer wrong questions than the Java group. Thus the EJFlow group was more able to capture the general idea of the exception control flow in few minutes. Moreover, the decrease in scores for the Java and EJFlow groups from first to second trial can be attributed largely to the mechanisms provided by these approaches to represent the exception control flows. In Java the subjects could use the exception interface and try/catch blocks to follow the calls until the appropriate handling site. EJFlow subjects could use the notion of explicit exception channel and pluggable handlers to easily understand the exception control flows. At the end, the EJFlow approach performed better as its overall means had 54% fewer wrong questions than the Java one.

Unlike EJFlow and Java groups, the AspectJ group did not improve from the first to the second trial. This indicates that AspectJ subjects had more difficulty than any other subject in tracking back the exception control flows. This was also observed even when they were given a comprehension task. As described in Section 2, one of the problems of AspectJ, like other aspect-oriented program languages, is that it provides mechanisms to separate handlers from the normal behaviour code, but no appropriate support to attach global exception handlers. The lack of explicit relationships between raising and handling sites seems to be the most plausible reason for the high number of incorrect answers given by the AspectJ group.

According to the observed results, all the tasks rejected the null hypotheses for implementation time, number of uncaught exceptions, and number of incorrect questions. The alternative hypotheses stated that the implementation time, the number of uncaught exceptions, and the number of wrong answers are reduced for prototypes using EJFlow. In fact, we observed that when there is a significant difference in favour of EJFlow implementations.

Less implementation time was required for EJFlow implementations than with other approaches. In addition, EJFlow led to fewer uncaught exceptions and incorrect
answers. This indicates that the use of EJFlow abstractions helps to increase implementation productivity and reduce error proneness.

5.1. Lessons Learned

The study results provide sensible evidence that purely textual separation between normal and error handling code is not enough to achieve the expected benefits of modular software. According to [Parnas 1972], modular software systems should be easier to modify and understand than non-modular systems. However, according to the results of the study, AspectJ did not simplify maintenance tasks (which resulted in a large number of uncaught exceptions) and yielded programs that are harder to understand. Even though the implementation of exception handling in Java is, at the same time, scattered and tangled with code implementing other concerns, the Java group fared better than the AspectJ one. We believe that this was the case because the exception handlers in a Java program are more directly associated with its methods (and therefore with its method call chain).

The results that the EJFlow group obtained also raise the question of whether general-purpose AOP is really the best approach for the construction of modular software. EJFlow programmers outperformed the AspectJ ones mainly because EJFlow provides mechanisms and automated support for developers to deal with the specific problems created by exceptions. Further research is still necessary to assess whether the benefits obtained by using a domain-specific AOP language targeting error handling also apply to other crosscutting concerns. Even though AOP started out focusing on domain-specific languages for modularizing crosscutting concerns [Kiczales et al. 1997], this subject has not received much attention until recently (e.g. [Fabry et al. 2008]).

5.2. Threats to Validity

The conclusions obtained in this study are restricted to the involved exception handling policies and the target software systems. Hence, the obtained results regarding advantages and drawbacks of EJFlow should not be directly generalised to other contexts. However, this study allowed us to make a first useful evaluation on whether the utilisation of EJFlow abstractions could improve software understandability and maintainability.

One issue that limits our findings is the fact that the experiment participants were undergraduate students taking an undergraduate course on advanced programming language topics. Novices and experts programmers are different between them because they have different levels of proficiency in software development process. However, against the expectation, some existing tasks relate that, in some cases, the contrast between novices and experts programmers performance is not significant [Wiederbeck et al. 1999]. In order to reduce such a confounding factor, we randomly distributed the subjects to groups instead of defining blocks, which would decrease the number of samples to be compared. Despite being isolated in three groups (Java, AspectJ, EJFlow), all the subjects used the same application as target. The application was designed, implemented, and presented in a way which was familiar to all subjects from their experience in their courses. Only language features taught in the course have been used; they had already used those features in their own programming assignments.
Other criticism to this study might be that the EJFlow implementation was based on a pre-existing OO version. We tried to reduce the negative or positive impact of such a refactoring by ensuring that coding styles, good design practices, and implemented functionalities were exactly the same to their counterparts in Java and AspectJ. Hence, we think these factors did not undermine the validity of the results as the evaluation purpose was also to broadly understand the issues surrounding the use of EJFlow abstractions.

6. Related Works

There are several papers that propose extensions to existing programming languages in order to improve the quality of exception handling code and, as a consequence, the quality of the normal code. One the one hand, the Lore language [Dony 1988] allows the definition of different levels of granularity for EHCs, based on static program elements such as classes, method declarations, and exception classes. On the other hand, Cui and Gannon [Cui and Gannon 1992] argue that, in object-oriented languages, it makes sense to associate exception handlers to runtime objects. Our approach generalizes these approaches by using the pointcut language of AspectJ, plus the extensions we propose, to select both fine- and coarse-grained EHCs to which exception handling advice can be associated. A complementary idea is to extend existing languages with constructs that act as implementation-level specifications of error handling behavior [van Dooren and Steegmans 2005b]. In this approach, it is possible to specify that a method signals the same exceptions as another method, usually one it invokes. The ExnJava [Malayeri and Aldrich 2006b] is not really a language extension (though it implements a very lightweight modification to the semantics of Java’s throws clause). Instead, it is a tool comprising several features, such as a module system and exception specific refactorings, that aim to support the specification of exceptions.

In the light of software usability, [Corritore and Wiedenbeck 2001] provided empirical dates about the degree of comprehension-related activities proceed in repeated maintenance tasks. Again [Corritore and Wiedenbeck 1999, Susan Wiedenbeck and Corritore 1999] inspected the comprehension of expert and novice programmers in the procedural and object-oriented programs evolution. Even only working with novice programmers in our experiment we find some similarities with their work in the both cases. For instance, We encounter a kind of evolution of the mental representation over time after several modifications of the same program and find that OO programmers tend to migrate from a top-down to a bottom-up orientation during the software modification process. Also, during our experiment the major specific difficulties were related to the comprehension of the application structure and to the inheritance of the functionality as reported by [Amela Karahasanovic and Thomas 2007].

At the same line of thinking, [Robert J. Walker and Guerraoui 1999] provide some insights into the usefulness and usability of aspect-oriented programming and its impacts in programming tasks. Basically, their initial validation was focused in debugging and change operations. However, their study did not investigate how system architecture and the degree of programmers’ expertise can induce the software process development. [Soares and Borba 2005] defined guidelines to restructure object-oriented software in order to modularize concurrency control using aspect-oriented programming. Our work differs from them once we are concerned about issues related
to exception handling and flow control. Besides, we also take in consideration the programmers’ mental representation.

7. Conclusions

This work presents a study that aims to evaluate if a novel exception handling model improves the maintainability and robustness of software systems. The idea was to validate whether the programmers’ understanding increases with the use of this technique and also if they are able to produce programs more robust. We have observed that the use of EJFlow in our experiments was useful to foster the development of readable and reliable software systems. More specifically, our data analysis has provided significant evidence that the use of exception channels and their attached handlers can lead to the creation of exception handling behaviour with superior changeability and robustness. As future work, there are some open issues: (i) additional empirical studies, based on other usefulness and usability indicators, are required to assess further benefits and drawbacks of EJFlow, and (ii) support reasoning and specification of exception channels throughout the software lifecycle.

References


