# 3
# Characterizing the Exception Flow in Aspect-Oriented Programs: The Empirical Study

This chapter describes an empirical study whose goal is to evaluate the impact of AOP on exception flows of AspectJ programs, comparing them with its pure Java counterparts. The empirical study is presented according to the steps defined by Wohlin et al. (2000). The study configuration is defined in terms of (i) its goals and hypotheses, (ii) the criteria used for the target systems selection, (iii) the methodology employed to conduct the exceptional code analyses, and (iv) the description of actual execution of our study. The investigation relies on determining in multiple Java and AspectJ versions which exception-handling faults are introduced in AO releases. The consequences of such faults vary from *uncaught* exceptions to *unintended handler actions*.

## 3.1.
## Study Setting

In the *case study* described here, OO and AO versions of three real applications were compared in order to observe the positive and negative effects caused by aspects on their exception flows. Specific procedures were undertaken in order to distinguish AOP liabilities for exception handling implementation from well-known intrinsic impairments of OO mechanisms on exception handling (Miller and Tripathi, 1997) (see Section 3.1.1). These procedures were important to detect whether and which aspect-oriented mechanisms are likely to lead to unexpected and error-prone scenarios involving exception handling.

In this study we opted to conduct a *case study* instead of a *controlled experiment.* The reason is twofold. Firstly, differently from a *controlled experiment*, in a *case study* we have a low level of control over the study situation (the target systems were developed by third party developers). Secondly, while controlled experiments sample over the variables that are being manipulated, *case studies* analyze variables representing typical real situations (Wohlin et al., 2000).

Consequently, if on the one hand there are some factors which may influence the result of the study (e.g. expertise of developers - see Section 3.2.3), on the other hand real development situations can tell us much about the problem being studied.

Thus, the hypothesis of our case study were the following: (i) the null hypothesis (H0) for this study states that there is no difference on the robustness of exception handling code in Java and AspectJ versions of the same system; (ii) the alternative hypothesis (H1) is that the impact of aspects on exception flows of programs can lead to more program flaws associated with the exception flow.

### 3.1.1. Target Systems

One major decision that had to be made for our investigation was the selection of the target applications. We have selected three medium-sized systems to which there was a Java version and an AspectJ version available. Each of them represents a different application domain, and adopts heterogeneous and realistic ways of incorporating exception handling into the code. The target systems were: Health Watcher (HW) (Soares, 2004; Kulesza et al., 2006; Greenwood et al., 2007), Mobile Photo (MP) (Figueiredo et al., 2008) and JHotDraw[5] (JHD) (Deursen et al., 2005; Marin et al., 2007).

#### 3.1.1.1. Health Watcher

The Health Watcher (HW) system (Soares, 2004; Kulesza et al., 2006; Greenwood et al., 2007) is a Web-based application that allows citizens to register complaints regarding issues  in health care institutions. There are 9 versions of HW system available[6], implemented in both OO and AO designs. They vary in terms of the number of functionalities available and some minor design decisions. All versions of HW adopt the Layer architectural pattern (Buschmann et al., 1996) which separates data management, business, communication, and presentation concerns. According to this pattern, the elements from each layer communicate

[5] Project Homepage: http://www.jhotdraw.org/
[6] Homepage that contains the all versions of Health Watcher system source code: http://www.comp.lancs.ac.uk/computing/users/greenwop/tao/

only through well defined layer interfaces. The purpose of a layer interface is to define the set of available operations - from the perspective of interacting client layers - and to coordinate the layer response to each operation. Several design patterns were used to refine each layer of this architecture. Some of them are: the Facade Pattern (Gamma et al., 1995), the Command Pattern (Gamma et al., 1995) and the Persistent Data Collections (PDC) pattern (Massoni et al., 2001). In this study we selected the 1st and the 9th versions to conduct our analysis.

Figures 3 and 4 present representative slices of the 9th version in both OO and AO architecture designs. In the OO and AO designs of HW system, the GUI layer is implemented using Java Servlets[7]. The Servlet executes commands (implemented according to the Command Pattern) which access `HealthWatcherFacade`. This facade works as a portal to access the business collections (e.g., `ComplaintRecord` and `EmployeeRecord`), which are the elements responsible for accessing the Data layer. Figure 3 also illustrates how some concerns are spread over system elements in OO design.

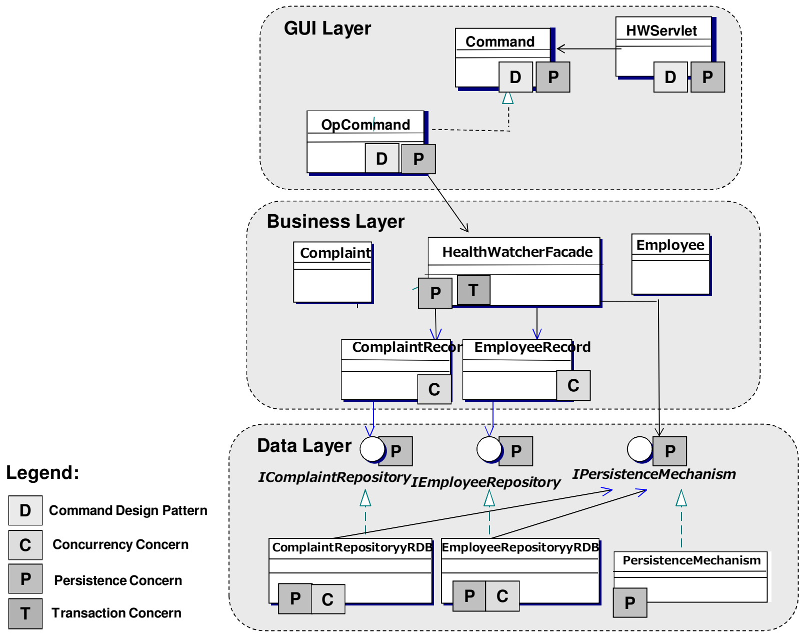

**Figure 3.** The OO design of Health Watcher system (version 9).

[7] Although the original version of Health Watcher system presented in (Sergio, 2004) implements the distribution layer, this concern was not used in our study. This is the reason why it is not represented on pictures that depicts HW architecture.

In the AO design presented in Figure 4 some concerns that were tangled and scattered in the OO decomposition counterpart were "*aspectized*" (i.e., refactored to aspects). Basically, in the AO release of the HW system presented below, crosscutting concerns relative to persistence, transaction management, and concurrency control were represented as aspects. Moreover, the exception handling concern of every crosscutting concern was also represented a set of aspects (e.g., `HWPersistenceExceptionHandler` and `HWTransactionExceptionHandler`) as illustrated in Figure 4. Such *exception handling aspects* (also called *handler aspects*) intercept the points in the code where exceptions thrown by the corresponding crosscutting concerns should be handled.

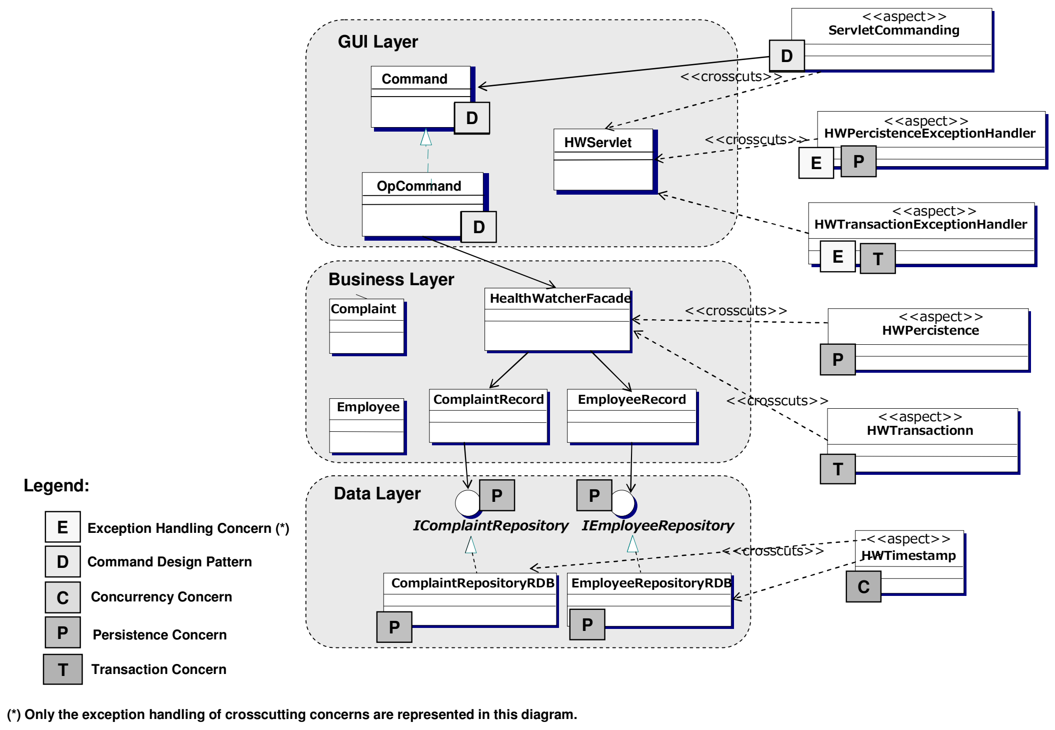

**Figure 4.** The AO design of Health Watcher system (version 9).

### 3.1.1.2. Mobile Photo

The Mobile Photo (MP) is a software product line (SPL) of applications that manipulate media (e.g., photo, music and video) on mobile devices, such as cell phones (Figueiredo et al., 2008). There are 6 releases of this SPL available. All SPL versions adopt the same architecture style, varying in terms of the number of functionalities available and design decisions taken in each version. In this study we selected the 4th and 6th versions. The 4th version allows the manipulation of a single type of media (i.e., photo) and the 6th allows the manipulation of photos and audio files. The manipulation tasks available in MP comprise the following: to sort a list of media, to choose the favorites, to copy a media and to send SMS messages including a specific media.

Figure 5 illustrates the OO design of the 6th version of MP. It adopts a model-view-controller architecture style. When an item on the screen is selected, a command is executed by a specific controller, then the operation is performed on the model layer and the screen is updated to reflect the changes. Each set of commands is organized in a specific controller, for instance the `PhotoViewController` accounts for the operations related to the photo view concern.

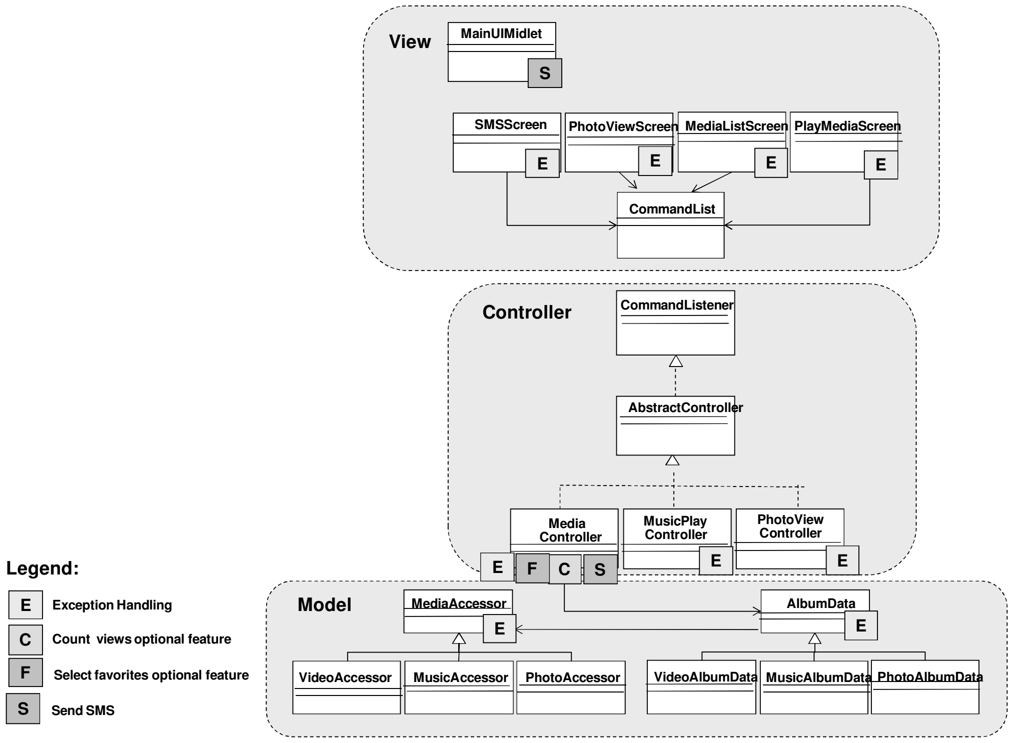

**Figure 5.** The OO design of Mobile Photo System (version 6).

It the AO design some concerns were implemented as aspects as illustrated in Figure 6. The Exception handling concern was partially aspectized, and *handler aspects* (i.e., aspects responsible for handling exceptions signaled in the system by elements of the base code or other aspects) were defined per layer. This system has used the catalog of best practice defined in (Filho et al., 2007) to guide error handling code modularization. Some functional requirements comprising the manipulation of different kinds of media (i.e., photos and audio files) were also implemented as aspects: to sort a list of medias, to choose the favorites, and to copy a media and sending SMS.

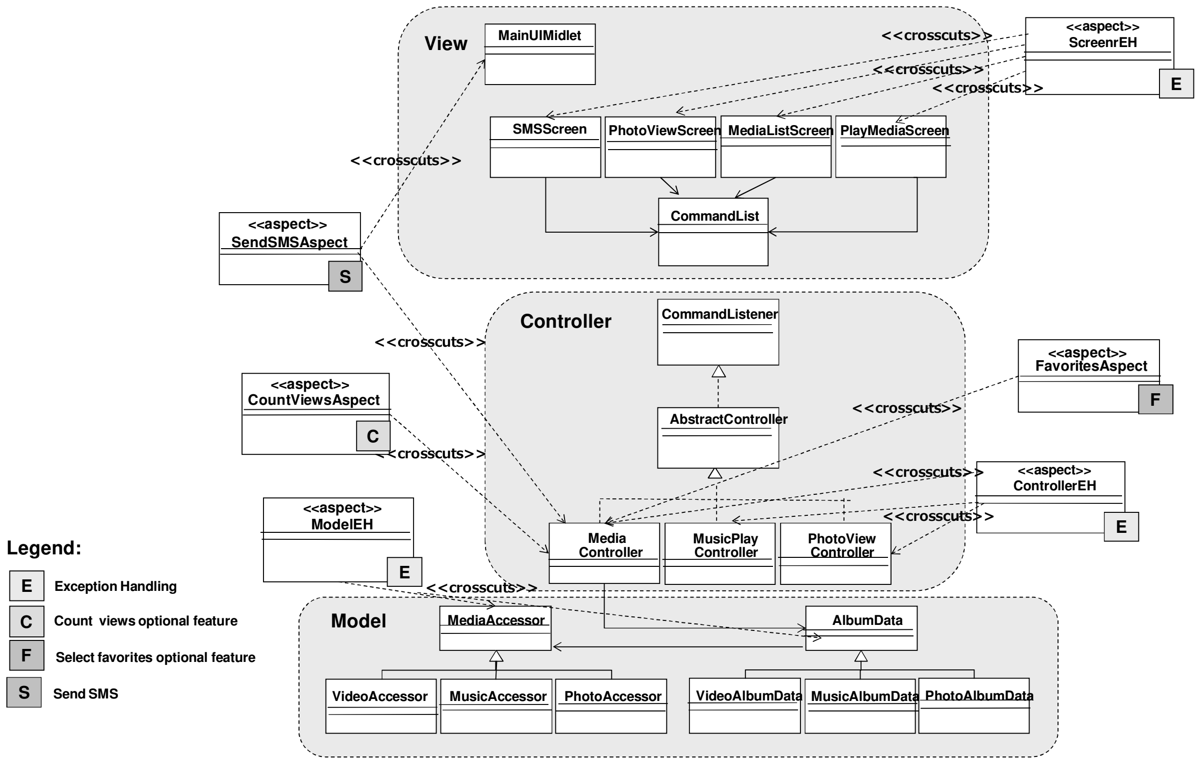

**Figure 6.** The AO design of Mobile Photo System (version 6).

### 3.1.1.3. JHotDraw

JHotdraw framework is an open-source framework for building graphical drawing editor applications. Drawing editors are used to visually arrange graphical figure objects on a drawing area, and are present on nearly every computer. Figures 7 and 8 present an overview of the OO and AO design[8] of JHotdraw; both adopt the Model-View-Controller architectural pattern (Buschmann et al., 1996).

[8] JHotdraw comprises a Java Swing and an Applet interface, but in our study, we have focused in the java Swing version JHotdraw.

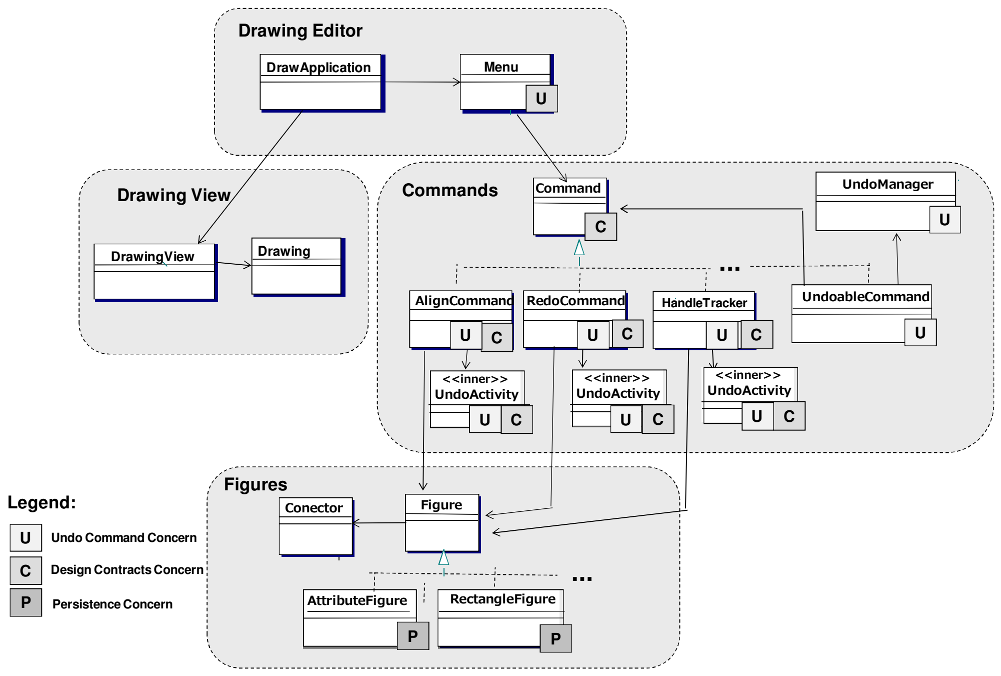

**Figure 7.** The OO design of JHotDraw.

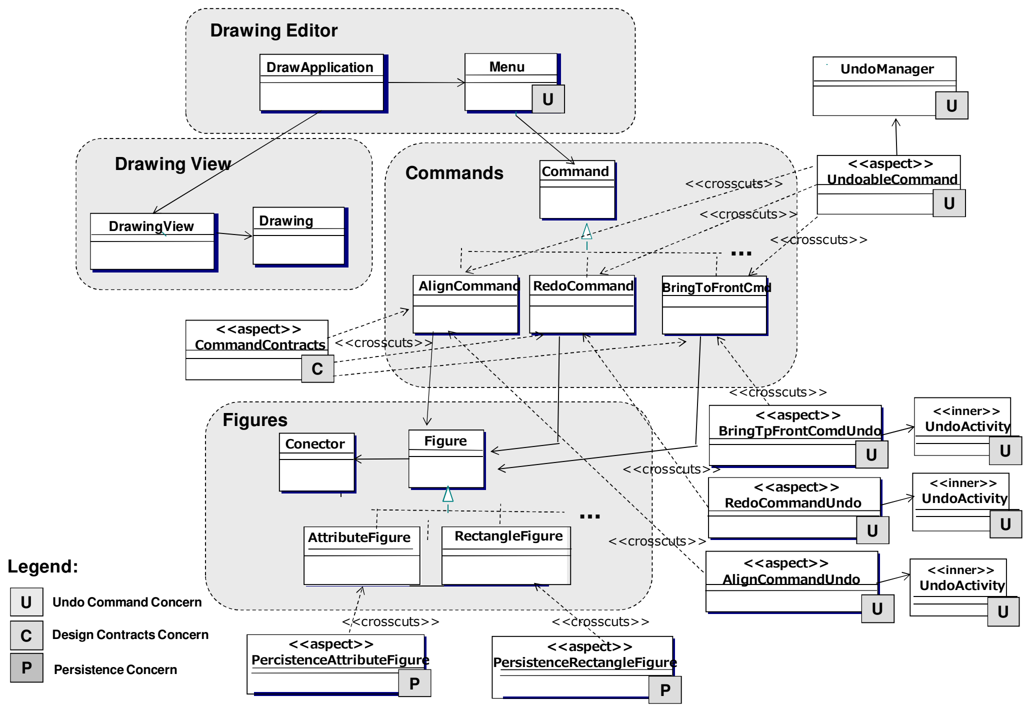

**Figure 8.** The AO design of AJHotDraw.

The AO version of the JHotDraw (AJHotDraw) system was built through a well defined set of *refactoring* steps (Deursen et al., 2005; Marin et al., 2007) whose goal was to modularize crosscutting concerns. Most of the aspects derived from these *refactoring* steps are composed by intertype declarations. Some *refactoring* steps moved specific methods from classes to aspects, such as the methods related to persistence and undo concerns (see Figure 8). Since, the exception handling concern was not *aspectized* in JHotdraw, the handlers defined for the exceptions thrown by the refactored methods (moved to aspects) remained on the same places on the base code. In the AO version some concerns were *refactored* to aspects: the persistence concern, the undo command concern and some design contracts related to command execution.

### 3.1.1.4. Characteristics of the Target Systems

As mentioned before, in this study, more than one version was evaluated for some target systems. Table 2 summarizes the crosscutting concerns that were implemented as aspects in the AO versions of each target system.

| System | Versions and Respective Crosscutting Concerns |
|---|---|
| Health Watcher (HW) | **Version 1**: concurrency control, persistence (partially) and exception handling (partially). |
| | **Version 9:** concurrency control, transaction management, design patterns (Observer, Factory and Command), persistence (partially) and exception handling (partially). |
| Mobile Photo (MP) | **Version 4**: exception handling and some functional requirements comprising photo manipulation, such as to sort a list of photos, to choose the favorites, and to copy photo. |
| | **Version 6**: exception handling and some functional requirements comprising the manipulation of different kinds of media (i.e., photos and audio files), such as: to sort a list of medias, to choose the favorites, and to copy a media and sending SMS). |
| AJHotDraw (HD) | **Version 1:** persistence concern, design policies contract enforcement and undo command. |

**Table 2.** Crosscutting concerns per target system version.

The target systems were also selected because they met a number of relevant additional criteria for our intended evaluation (Section 3.2). First, they are non-trivial software projects and particularly rich in the ways exception handling is related to other crosscutting and non-crosscutting concerns. Second, the behavior of exception handlers also significantly varied in terms of their purpose, ranging from error logging to application-specific recovery actions (e.g., rollback). Third, each of these systems contains a considerable amount of code dedicated to exception handling, within both aspects and classes, as detailed in Table 3.

| | Health Watcher V1 | | Health Watcher V9 | | Mobile Photo V4 | | Mobile Photo V6 | | HotDraw | |
|---|---|---|---|---|---|---|---|---|---|---|
| Number of: | OO | AO | OO | AO | OO | AO | OO | AO | OO | AO |
| Lines of code | 6080 | 5742 | 8825 | 7838 | 2540 | 3098 | 1571 | 1859 | 21027 | 21123 |
| Lines of code for exception handling | 1167 | 854 | 1889 | 1242 | 474 | 424 | 356 | 296 | 320 | 341 |
| Classes | 88 | 90 | 132 | 129 | 46 | 49 | 30 | 29 | 288 | 279 |
| Aspects | 0 | 11 | 0 | 24 | 0 | 14 | 0 | 10 | 0 | 31 |
| `try` blocks | 131 | 118 | 233 | 173 | 49 | 40 | 36 | 24 | 60 | 61 |
| `catch` blocks | 285 | 177 | 481 | 266 | 69 | 60 | 52 | 38 | 67 | 72 |
| `throw` clauses | 227 | 182 | 334 | 229 | 21 | 18 | 20 | 17 | 52 | 56 |
| `try` blocks inside classes | 131 | 108 | 233 | 161 | 49 | 21 | 36 | 9 | 60 | 61 |
| `catch` blocks inside classes | 285 | 164 | 481 | 252 | 69 | 28 | 52 | 16 | 67 | 72 |
| `throw` clauses inside classes | 227 | 176 | 334 | 219 | 21 | 4 | 20 | 4 | 52 | 51 |
| `try` blocks inside aspects | n/a | 10 | n/a | 12 | n/a | 19 | n/a | 15 | n/a | 0 |
| `catch` blocks inside aspects | n/a | 13 | n/a | 14 | n/a | 32 | n/a | 22 | n/a | 0 |
| `throw` clauses inside aspects | n/a | 6 | n/a | 10 | n/a | 14 | n/a | 13 | n/a | 5 |
| `after` advices | n/a | 4 | n/a | 22 | n/a | 30 | n/a | 15 | n/a | 15 |
| `around` advices | n/a | 5 | n/a | 6 | n/a | 21 | n/a | 17 | n/a | 18 |
| `before` advice | n/a | 3 | n/a | 4 | n/a | 5 | n/a | 2 | n/a | 15 |

**Table 3.** Code characteristics per system.

Finally, AOP was applied in different ways through the system releases: (i) aspects were used to extract non-exception-handling concerns in JHotDraw, and all exception handlers are defined in the base code, (ii) aspects were used to modularize various crosscutting concerns in the Mobile Photo product line, including exception handling apart from the original release, and (iii) aspects were used to partially implement error handling in Health Watcher, where other behaviors were also *aspectized*. AOP best practices were applied to structure such systems as stated in (Soares et al., 2006; Greenwood et al., 2007; Marin et al., 2007; Figueiredo et al., 2008). Similar to Java releases, all the AspectJ releases were implemented and changed by developers with around three years of experience in aspect-oriented design and programming. In fact, HW and MP systems have been used in the context of other empirical studies focusing on the assessment and comparison of their Java and AspectJ implementations in terms of modularity and stability (Greenwood et al., 2007; Figueiredo et al., 2008). Alignments of Java and AspectJ versions have been undertaken in order to guarantee both were implementing the same normal and exceptional functionalities.

## 3.1.2. Reasoning About the Exceptional Behavior

Reasoning about the exception flow of programs can easily become unfeasible if done manually (Robillard and Murphy, 1999). To discover the exceptions that can flow from a method, the developer needs to recursively analyze each method that can be called from such method - due to the use of unchecked exceptions. Moreover, when one method from a library is used, the developer must rely on library documentation, which most often is neither precise nor complete (Thomas, 2002; Sacramento et al., 2006).

Thus to support the reasoning about the flow of exceptions in AspectJ programs in our study we had to implement the *exception flow analysis tool* called SAFE briefly described on the next section and detailed in Chapter 5.

## 3.1.3. Automated Exception Flow Analysis

Current exception flow analysis tools (Robillard and Murphy, 2003; Fu et al., 2005) do not support AOP constructs. Even the tools which operate on Java bytecode level (Fu et al., 2005; Fu and Ryder, 2007) cannot be used in a straightforward fashion. They do not interpret the aspect-related code included on the *bytecode* after the weaving process of AspectJ. Hence, we developed a static analysis tool, called SAFE (*Static Analysis for the Flow of Exceptions*), to derive exception flow graphs on AspectJ programs and support our investigation on determining flaws associated with exception flows. This tool is based on the Soot[9] framework for bytecode analysis and transformation (Vallée-Rai et al., 1999) and is composed of two main modules: the Exception Path Finder and the Exception Path Miner. Both components are described next, and more detailed information can be found at Chapter 5.

- **Exception Path Finder**. This component uses SPARK, one of the call graph builders provided by Soot (Lhotak, 2002), also used by other static analysis tools (Fu et al., 2005). The Exception Path Finder generates the exception paths for all checked and unchecked exceptions, explicitly

[9] http://www.sable.mcgill.ca/

thrown by the application or implicitly thrown (e.g. via library method) by aspects and classes. It associates each exception path with information regarding its treatment. For instance, whether the exception was uncaught, caught by *subsumption* or caught by the same exception type. In this study we are assuming that only one exception is thrown at a time – the same assumption considered in (Fu et al., 2005; Fu and Ryder, 2007)[10]. Listing 2 below illustrates a simplified version of the tool output. The first element in the exception path is the exception signaler and the last element is its handler, followed by the handler type (e.g., subsumption, specific handler or uncaught).

```
(Signaler)<healthwatcher.persistence.TransactionManagementAspect: void afterReturning()>
(Intermediate Element)<healthwatcher.business.HealthWatcherFacade: void insert(…)>
(Handler)<healthwatcher.view.command.InsertHealthUnit: void executeCommand (…)>
(Action) Subsumption: org.aspectj.lang.SoftException captured by java.lang.Exception
```

**Listing.2.** Example output from the exception flow analysis tool.

- **Exception Path Miner**. This component classifies each exception path according to its signaler (i.e., class method, aspect advice, intertype or declare soft constructs) and handler. Such classification helps the developer to discover the new dependencies that arise between aspects and classes on exceptional scenarios. For instance, an exception can be thrown by an aspectual module and captured by a class or vice-versa. These different dependencies represent seeds to manual inspections whose goal is to evaluate the fault proneness of the abnormal code in AO systems. More details about the tool implementation are described in Chapter 5.

### 3.1.4. Inspection of Exception Handlers

To discover the action taken inside each exception handler, we performed a complementary manual inspection. It consisted of examining the code of each

[10] In parallel applications, more than one component executing in parallel may detect an exceptional condition and signal an exception. These scenarios, however, are not tackled by the approach defined in this work.

handler associated with exception paths found by the exception flow analysis tool (Section 3.1.3). Such manual inspections were also targeted at: discovering the causes for uncaught exceptions and exception *subsumptions* and removing *spurious* paths reported by the tool. The manual inspection enabled us to systematically discover *bug hazards* associated with Java and AspectJ modules on the exception handling code. A bug hazard (Binder, 1999) is a circumstance that increases the chance of a bug. For instance, type coercion in C++ is a bug hazard because it depends on complex rules and declarations that may not be visible when working with a class.

| Category | Description |
|---|---|
| *Error Reporting* | |
| **exception message** | The message attribute defined on the exception object (exception. `getMessage()`) is presented to the user. |
| **customized message** | A user-defined message generally describing the failure is presented to the user. |
| **incorrect user message** | A message that is not related to the failure that happened is presented to the user. |
| *Error  Propagation* | |
| **Uncaught** | No handler catches the exception. |
| **new exception** | The handler catches an exception, (i) creates a new exception and (ii) throws it. |
| **Wrap** | This category is a specialization of the previously described category (i.e., **new exception**). In this case, the handler catches the exception, and stores the original exception in a new exception which is thrown. |
| **convert to soft** | This category is a specialization of **wrap** category. In this case, the exception is wrapped into a SoftException. This action is specific to AspectJ programs, and occurs when the `declare soft` construct is used. |
| Error Local Handling | |
| **Swallowing** | The handler is empty. |
| **Logging** | Some information related to the exception is logged, and no other action is taken. |
| **framework default action** | To avoid uncaught exceptions some application frameworks such as java.swing, define catch classes that handle any exception that was not caught by the application and performs a default action (e.g. kill the thread which threw the exception.). |
| **application specific action** | A specific action is performed (e.g., rollback). |

**Table 4.** Categories of handler actions and corresponding descriptions

Each handler action was classified according to one of the categories presented in Table 4. We can observe that each handler action category in Table 4 can of one of the following types: (i) *Error Reporting* when the action taken inside the handler is concerned in presenting an error message to the user; (ii) *Error Propagation* when the handler propagates the error to higher levels; and (iii) the *Error Local Handling* that comprises the handler actions locally handles the exception (without reporting or propagating).

## 3.1.5. Study Operation

This study began in March 2007 and was concluded in November 2007. During this period target systems were selected, the static analysis tool was executed for each target system, and the exception handling code of each system was manually inspected. Figure 9 illustrated the main study steps conducted per target system.

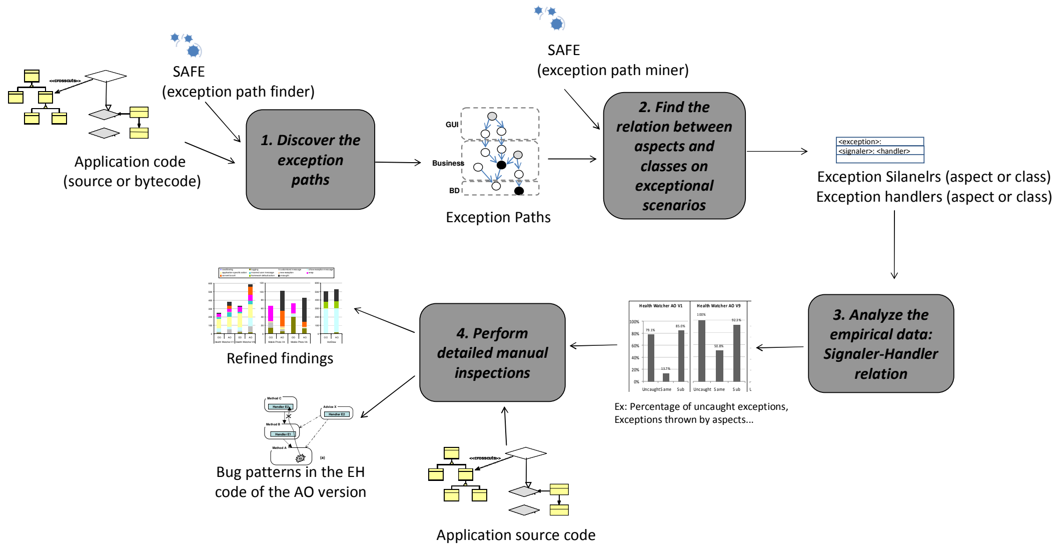
SAFE
(exception path finder)
Application code
(source or bytecode)
1. Discover the exception paths
GUI
Business
BD
Exception Paths
SAFE
(exception path miner)
2. Find the relation between aspects and classes on exceptional scenarios
<exception>:
<signaler>: <handler>
Exception Silanelrs (aspect or class)
Exception handlers (aspect or class)
3. Analyze the empirical data: Signaler-Handler relation
Ex: Percentage of uncaught exceptions,
Exceptions thrown by aspects...
4. Perform detailed manual inspections
Refined findings
Bug patterns in the EH code of the AO version
Application source code

**Figure 9.** Each step conducted in the study operation.

The *Exception Path Finder* was used to generate the exception flow graph (i.e., set of all possible exception paths) for every exception occurrence (Step 1). Then the *Exception Path Miner* classified each exception path according to its signaler and handler (Step 2) (see Table 5). We have discarded a few unchecked exceptions[11] that can be thrown by JVM in almost every program statement execution and are not normally handled inside the system. The same filter was adopted by Cabral and Marques (2007), who empirically investigated the exception handling code in object-oriented systems - otherwise too many exceptions would be reported and would impair the study analysis (Jo et al., 2004). This filtering was performed on the static analysis tool. Next, the tool output was analyzed in detail (Step 3), and we manually inspected each one of the 2.901 exception paths presented in Table 5 (Step 4) – this set of exception paths comprises the exception paths calculated to every target system. The goal of this inspection was twofold: (i) discover what caused uncaught exceptions and exception *subsumptions*, (ii) identify the handler action (i.e., the action taken inside the `catch` clauses) of each exception path, and (iii) determine the *bug patterns* associated with exception handling code in AO systems versions.

## 3.2. Data Analysis and Interpretation

This section presents the results for each of the study stages. First, it presents evaluation of the data collected via the exception flow analysis tool. The following discussion focuses on the information collected during the manual inspections of each exception path.

Our goal in providing such a fine-grained data analysis is to enable a detailed understanding of how aspects typically affected the robustness of exception handling in each target system and its different releases. In this analysis we wanted to answer the following questions: *Were all the uncaught caused by flaws on aspectual code? Were all exceptions signaled by aspects becoming uncaught or caught by subsumption? The following sections describe the empirical data analyzed in this study and provide answer such questions.*

[11] The discarded exceptions were the exceptions thrown by JVM (NullPointerException, IllegalMonitorStateException, ArrayIndexOutOfBoundsException, ArrayStoreException, NegativeArray SizeException, ClassCastException, ArithmeticException) and exceptions specific to the AspectJ (NoAspectBoundException).Since such JVM exceptions may be thrown by almost every operation, including them could generate too much information which could compromise the usability of the exception analysis. The NullPointerException will be analyzed in a future study since it requires every expression to be analyzed in order to evaluate if it could lead or not to a NullPointerException.

### 3.2.1. Empirical Data

Table 5 presents the number of *exception paths* identified by the exception flow analysis tool (Section 3.1.3). It presents the tally of exception paths per target system structured according to a "*Signaler-Handler*" relation. The element responsible for signaling the exception can be either a class or an aspect. When the exception is signaled by an aspect, it is signaled by one of its internal operations: an advice, a method defined as intertype declaration, or a `declare soft` construct[12]. An exception occurrence can be caught in two basic ways. It can be caught by a *specialized handler* when the `catch` argument has the same type of the caught exception type. Alternatively, it can be caught by *subsumption* when the `catch` argument is a supertype of the exception being caught. It is also possible that the exception is not handled by the application and remains *uncaught*. This happens when there is no handler defined for the exception type in the exception flow.

| | Health Watcher V1 | | Health Watcher V9 | | Mobile Photo V4 | | Mobile Photo V6 | | HotDraw | |
|---|---|---|---|---|---|---|---|---|---|---|
| | OO | AO | OO | AO | OO | AO | OO | AO | OO | AO |
| **Signaler: Class** | | | | | | | | | | |
| Uncaught | 5 | 9 | 9 | 0 | 0 | 0 | 0 | 0 | 124 | 112 |
| **Handler on Class** | | | | | | | | | | |
| Specialized Handler | 196 | 132 | 277 | 119 | 53 | 26 | 63 | 13 | 64 | 5 |
| Subsumption | 43 | 26 | 47 | 21 | 13 | 0 | 9 | 0 | 316 | 143 |
| **Handler on Aspect** | | | | | | | | | | |
| Specialized Handler | n/a | 8 | n/a | 8 | n/a | 7 | n/a | 2 | n/a | 0 |
| Subsumption | n/a | 4 | n/a | 40 | n/a | 0 | n/a | 0 | n/a | 0 |
| **Signaler: Aspect** | | | | | | | | | | |
| **Construct: Advice** | | | | | | | | | | |
| Uncaught | n/a | 2 | n/a | 27 | n/a | 5 | n/a | 16 | n/a | 0 |
| **Handler on Class** | | | | | | | | | | |
| Specialized Handler | n/a | 0 | n/a | 0 | n/a | 2 | n/a | 0 | n/a | 0 |
| Subsumption | n/a | 3 | n/a | 2 | n/a | 1 | n/a | 3 | n/a | 84 |
| **Handler on Aspect** | | | | | | | | | | |
| Specialized Handler | n/a | 21 | n/a | 60 | n/a | 18 | n/a | 8 | n/a | 0 |
| Subsumption | n/a | 98 | n/a | 181 | n/a | 0 | n/a | 2 | n/a | 0 |
| **Construct: Declare Soft** | | | | | | | | | | |
| Uncaught | n/a | 32 | n/a | 1 | n/a | 42 | n/a | 40 | n/a | 0 |
| **Handler on Class** | | | | | | | | | | |
| Specialized Handler | n/a | 0 | n/a | 0 | n/a | 0 | n/a | 0 | n/a | 0 |
| Subsumption | n/a | 46 | n/a | 47 | n/a | 1 | n/a | 1 | n/a | 36 |
| **Handler on Aspect** | | | | | | | | | | |
| Specialized Handler | n/a | 0 | n/a | 63 | n/a | 0 | n/a | 0 | n/a | 0 |
| Subsumption | n/a | 0 | n/a | 20 | n/a | 0 | n/a | 0 | n/a | 0 |
| **Construct: Intertype** | | | | | | | | | | |
| Uncaught | n/a | 0 | n/a | 0 | n/a | 0 | n/a | 0 | n/a | 24 |
| **Handler on Class** | | | | | | | | | | |
| Specialized Handler | n/a | 0 | n/a | 0 | n/a | 0 | n/a | 0 | n/a | 0 |
| Subsumption | n/a | 0 | n/a | 0 | n/a | 0 | n/a | 0 | n/a | 121 |
| **Handler on Aspect** | | | | | | | | | | |
| Specialized Handler | n/a | 0 | n/a | 0 | n/a | 0 | n/a | 0 | n/a | 0 |
| Subsumption | n/a | 0 | n/a | 0 | n/a | 0 | n/a | 0 | n/a | 0 |

**Table 5.** Classification of exception paths per target system**.**

[12] `declare soft` is an AspectJ specific construct. It is associated to a pointcut and wraps any exception thrown on specific join points in a `SoftException`, and re-throws it.

The next subsections analyze the *exception paths* presented in Table 5 in detail. First, Section 3.2.1.1 contrasts the occurrence of *subsumptions* and *uncaught* exceptions in Java and AspectJ versions of each target system. Section 3.2.1.2 determines what the relation is between certain aspect elements (as exception signalers) and higher or lower incidences of *uncaught* exceptions and *subsumptions*. Section 3.2.1.3 focuses the analysis on how exceptions thrown by aspects are typically treated in the target systems.

### 3.2.1.1. The Impact of Aspects on How Exceptions are Handled

A recurring question to aspect-oriented software programmers is whether it is harmful to *aspectize* certain behaviors in existing object-oriented decompositions in the presence of exceptional conditions. Hence, our first analysis focused on observing how aspects affected the robustness of the original exception handling policies of the Java versions. Figure 10 illustrates the total number of exception paths on which exceptions (i) remained uncaught, (ii) were caught by *subsumption*, or (iii) were caught by specialized handlers, in each of the target systems.

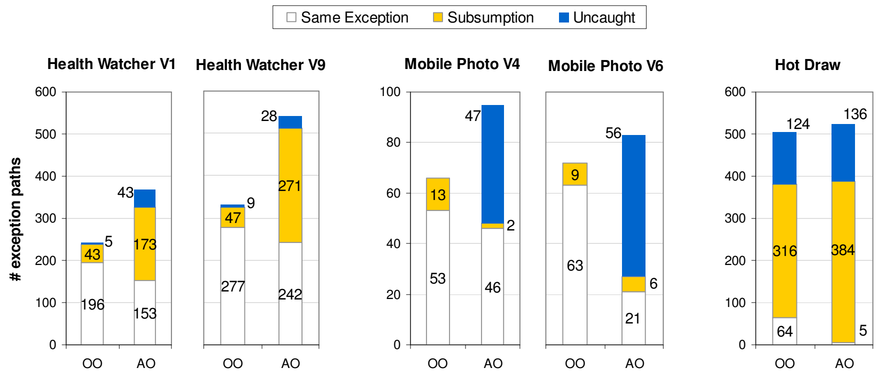

**Figure 10.** Uncaught exceptions, subsumptions, and specialized handlers per system.

Figure 10 shows a significant increase in the overall number of exception paths. Also significant is the increase of *uncaught exceptions* and *subsumptions* for the AO versions of all the three systems. This increase is a sign that the robustness of exception handling policies in AspectJ releases was affected and sometimes degraded when compared to their pure Java equivalents. Of course, the absolute number of exception paths is expected to vary due to design modifications, such as aspectual refactorings. However, the number of uncaught exceptions and *subsumptions* ideally should be equivalent between the Java and AspectJ implementations of a same system, since experimental procedures were undertaken to assure that both versions implemented the same functionalities.

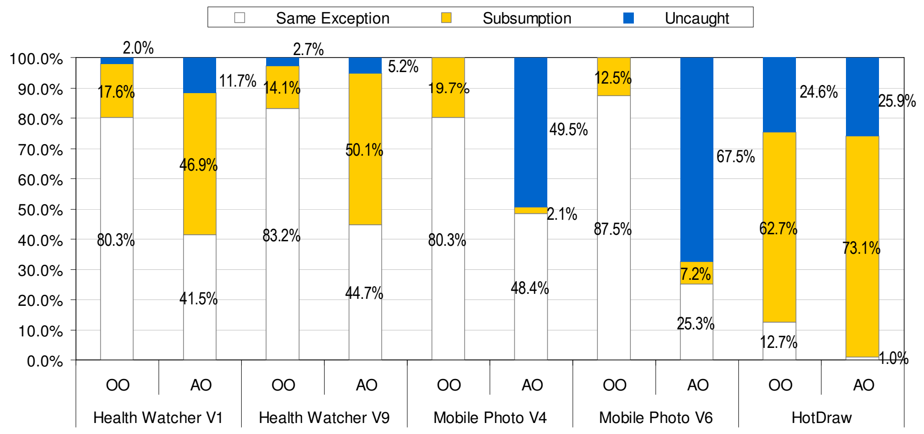

**Figure 11.** Percentage of uncaught exceptions, subsumptions, and specialized handlers.

Figure 11 shows the percentage of occurrence for each category of handler action. In the Mobile Photo V6, for example, the number of uncaught exceptions represents approximately 57% of the exceptions signaled on the system. In the Health Watcher V9, the percentage of exceptions caught by *subsumption* increased from 14.1% in OO version to 50% in the AO version. This significant increase raises the risk of unpredictable crashes in AspectJ systems, caused by either uncaught exceptions or inappropriate exception handling via *subsumptions*. Correspondingly, there was a decrease in the percentage of exceptions handled by specialized handlers in every AO implementation. When the handler knows exactly which exception is caught, it can take an appropriate recovery action or display a more precise message to the user. However, this was not the typical case in the AO implementations of the investigated systems.

### 3.2.1.2.
### The Blame for Uncaught Exceptions and Subsumptions

After discovering that the number of *uncaught* exceptions and *subsumptions* had significantly increased in the AO implementations (Section 3.2.1.1), we proceeded with our analysis, looking for the main causes of such discrepancies between AO and OO versions. Thus, at this stage of our study our hypotheses were the following: (i) the null hypothesis (H0) both classes and aspects were equally responsible for signaling the exceptions that became *uncaught* and were caught by *subsumption;* (ii) the alternative hypothesis (H1) was that most of the exceptions that became *uncaught* and were caught by *subsumption* were exceptions signaled by the aspects, in the three target systems. Figure 11 presents charts that confirm hypothesis H1; they show the participation of the exceptions signaled by aspects in the entire number of uncaught exceptions and *subsumptions* per system.

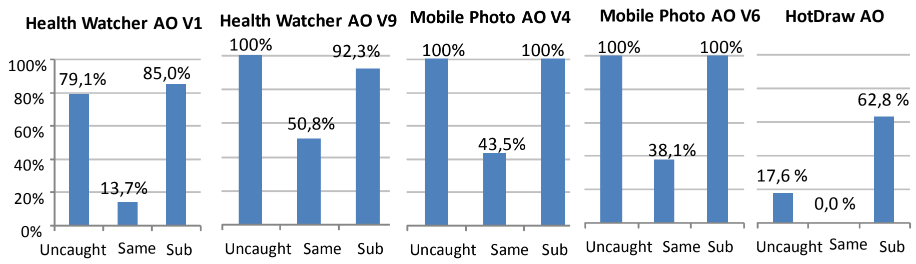

**Figure 11.** Participation of aspect-signalized exceptions on the whole number of subsumption, uncaught and specifically-handled exceptions.

In the AO implementations of the Health Watcher and Mobile Photo (in both versions), the aspects were responsible for signaling most of the uncaught exceptions and those ones caught by *subsumption*. In AO versions V4 and V6 of Mobile Photo, for example, aspects were responsible for 100% of the uncaught exceptions found in this system. This means that no base class in this system signaled an exception that became uncaught. In the AO version of JHotDraw, the aspects were responsible for signaling only 17.6% of the uncaught exceptions, and the aspects participation on the number of exceptions caught by subsumption was high (62.8%). This is explained by the fact that the exception policy of the JHotDraw OO was already based on exception subsumption, thus the exceptions signaled by aspects were handled in the same way.

### 3.2.1.3. Are All Exceptions Signaled by Aspects becoming Uncaught or Caught by Subsumption?

Figure 5 gives a more detailed view of what is happening with all exceptions signaled by aspects. We can observe that not all exceptions signaled from aspects become uncaught or are caught by *subsumption*. In version 9 of the AO implementation of Health Watcher, for example, only 7% of the exceptions signaled by aspects became uncaught, but they represented 100% of the uncaught exceptions reported to this system (see Figure 3). On the other hand, in the AO versions of the Mobile Photo, the percentage of exceptions signaled by aspects that became uncaught is high (68.1% and 80%). As discussed in the next section, this system was the one that *aspectized* the exception handling concern.

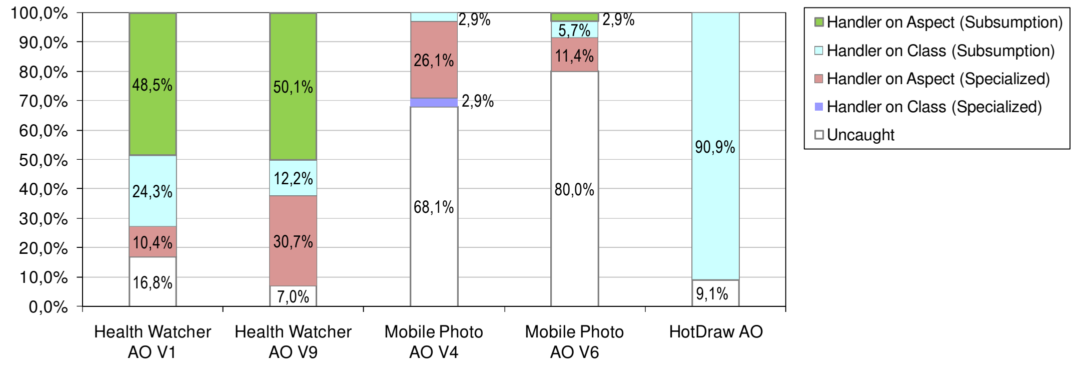

**Figure 12.** Handler type of exceptions thrown by aspects.

In Figure 12 the exceptions caught by *subsumption* on handlers codified inside classes characterize a *potential fault*. They may represent scenarios in which the exception signaled by an aspect is mistakenly handled by an existing handler in the base code. Another interesting thing to notice in Figure 12 is the increase in the percentage number of exceptions signaled by aspects and handled by specialized handlers from version V1 to version V9 of the AO implementation of Health Watcher. It illustrates that exceptions signaled by an aspect can be adequately handled.

### 3.2.2. Detailed Inspection

In order to obtain a more fine-grained view of how exceptions were handled in AO and OO versions of the same system we manually inspected each one of the

2,901 exception paths presented in Table 5. Each path was then classified according to the action taken in its handler – following the classification presented in Table 4 (in Section 3.1.4) illustrates the data collected during this manual inspection. Table 6 illustrates the data collected during this manual inspection.

| | Health Watcher V1 | | | Health Watcher V9 | | | Mobile Photo V4 | | | Mobile Photo V6 | | | HotDraw | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Handler Action** | **OO** | **AO** | **Ratio, %** | **OO** | **AO** | **Ratio, %** | **OO** | **AO** | **Ratio, %** | **OO** | **AO** | **Ratio, %** | **OO** | **AO** | **Ratio, %** |
| swallowing | 5 | 7 | 140.0 | 5 | 7 | 140.0 | 0 | 0 | -- | 0 | 0 | -- | 3 | 3 | 100.0 |
| logging | 7 | 1 | 14.3 | 12 | 10 | 83.3 | 14 | 6 | 42.9 | 41 | 13 | 31.7 | 4 | 11 | 275.0 |
| customised message | 12 | 43 | 358.3 | 20 | 73 | 365.0 | 13 | 4 | 30.8 | 0 | 0 | -- | 0 | 0 | -- |
| show exception message | 43 | 32 | 74.4 | 39 | 100 | 256.4 | 0 | 0 | -- | 7 | 1 | 14.3 | 291 | 285 | 97.9 |
| application specific action | 115 | 121 | 105.2 | 169 | 160 | 94.7 | 3 | 5 | 166.7 | 0 | 0 | -- | 8 | 0 | 0.0 |
| incorrect user message | 17 | 53 | 311.8 | 16 | 43 | 268.8 | 0 | 0 | -- | 0 | 0 | -- | 0 | 0 | -- |
| new exception | 3 | 3 | 100.0 | 3 | 3 | 100.0 | 0 | 3 | -- | 1 | 2 | 200.0 | 0 | 0 | -- |
| wrap | 37 | 38 | 102.7 | 60 | 65 | 108.3 | 36 | 0 | -- | 23 | 0 | 0.0 | 0 | 0 | -- |
| convert to soft | 0 | 40 | -- | 0 | 100 | -- | 0 | 37 | -- | 0 | 13 | -- | 0 | 8 | -- |
| framework default action | 0 | 0 | -- | 0 | 0 | -- | 0 | 0 | -- | 0 | 0 | -- | 74 | 82 | 110.8 |
| uncaught | 5 | 43 | 860.0 | 9 | 28 | 311.1 | 0 | 47 | -- | 0 | 56 | -- | 124 | 136 | 109.7 |
| **TOTAL** | 244 | 381 | 156.1 | 333 | 589 | 176.9 | 66 | 102 | 154.5 | 72 | 85 | 118.1 | 504 | 525 | 104.2 |

**Table 6.** Classification of exception paths according to handler actions.

As mentioned before, the total number of exception paths increased in most of the AO versions. During the manual inspections we discovered there were two causes for such increases: (i) if one exception is not caught inside a specific method (e.g., due to a fault on an aspect that acts as handler) this exception will continue to flow in the call chain, generating new exception paths; and (ii) specific design modifications bring new elements to the call graph and consequently lead to more exception paths. Figure 13 illustrates the handler actions per target system. Overall, it confirms the findings of previous sections based on the tool outputs: the aspects used to implement the crosscutting functionalities in the AO version tend to violate the exception policies previously adopted in each system. Subsequent subsections elaborate further on the data in Figure 13 and explain the causes behind AspectJ inferiority.

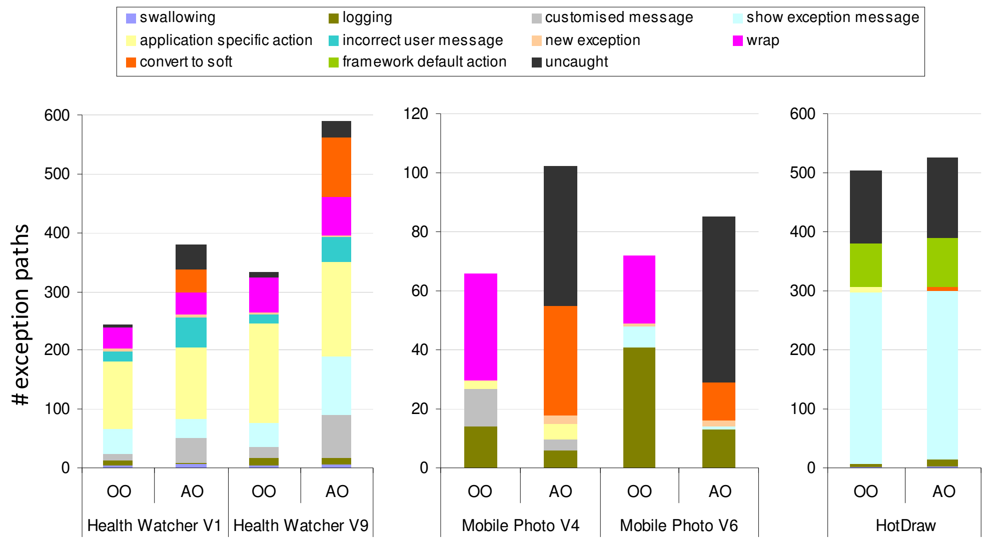

**Figure 13.** The handler action per target system.

### 3.2.2.1. Health Watcher

In the AspectJ versions V1 and V9 of Health Watcher, there was an increase in the number of exception paths classified as `incorrect user message` (see Table 4), in relation to the corresponding OO versions. It means that there were exception paths in these systems in which a message not related to the exception that really happened was presented to the user. This characterizes the problem known as *Unintended Handler Action,* when an exception is handled by mistake by an existing handler. The causes of such failures were diverse: (i) mistakes on the pointcut expressions of exception handling aspects in versions V1 and V9; (ii) in version V9, an aspect defined to handle exceptions intercepted a point in the code in which the exception was already caught; (iii) aspects signaled exceptions and no handler was defined for such exceptions in both versions; and (iv) the wrong use of the declare soft statement. Each of these causes entails a bug pattern in AspectJ that will be discussed in Chapter 4. In the version V1 of HW all softened exceptions became uncaught (categories `concert to soft` and `uncaught` respectively), because the `declare soft` statement was not used correctly (see *Handler Mismatch* in Chapter 4). In version 9 of the AO HW system the misuse of the `declare soft` statement was corrected but some exceptions remained uncaught or unintended, handled by a `catch` block on the base code that presented an `incorrect user message`.

#### 3.2.2.2. Mobile Photo

In all AO versions of Mobile Photo there was a significant increase in the number of uncaught exceptions. This application defined many exception handler aspects. Due to mistakes on pointcut expressions and a limitation on the use of `declare soft` many exceptions became uncaught. Different than the exception handling policy defined in the HW system - that defined “catch all” clauses on elements of the View layer - according to the exception policy defined in Mobile Photo, if the exception was not handled, it became *uncaught*.

#### 3.2.2.3. JHotDraw

The target system that presented the lesser impact on the exception policy was the JHotDraw system. The reason is twofold. First, the exception policy in OO version was poorly defined, which is visible thanks to the expressive number of uncaught exceptions and *subsumptions* (Figure 11). Second, the AO version of the JHotDraw (AJHotDraw) system was built through a well defined set of refactoring steps (Deursen et al., 2005; Marin et al., 2007), and most of the aspects of AJHotDraw are composed by intertype declarations. These refactorings moved specific methods from classes to aspects, such as the methods related to persistence and undo concerns. The `catch` statements for exceptions thrown by the refactored methods were not affected in the AO version, i.e. they remain in the same places on the base code. This explains why most of the exceptions signaled by aspects were caught by base code classes (Figure 12). However, even this system presented potential faults in the exception handling code (see Chapter 4).

### 3.2.3. Study Constraints

The main benefit of an exploratory study such as this one is that it allows the effect of a new method to be assessed in realistic situations (Wohlin et al., 2000). Somebody could argue that evaluating the AO and OO versions in a set of 10 releases for three different systems is a limiting factor. Indeed it is not a representative set, but it contains systems that implement significantly varied policies and *aspectization* processes for exception handling (Section 3.1) and can give us good insights about the impact of aspects on the exception flow of programs. Another factor that might influence the study results *against* aspectual decompositions could be the developers' expertise on AOP and AspectJ. However, as mentioned before (Section 3.1) all the target systems developers had a significant experience in AOP and AspectJ constructs. Moreover, the fact that the AO version of each target system was developed after the OO version, could also impact in the study results, acting *in favor* or *against* AO solutions. However, most AO systems developed so far are derived from an OO version, to which AO *refactorings* are typically applied. Therefore, the threats to validity in this study are not much different than the ones imposed on the other empirical studies with similar goals (Kulesza et al., 2006; Greenwood et al., 2007; Figueiredo et al., 2008).

## 3.3. Summary

An empirical study was conducted to evaluate the impact of aspects on the exception flow of AO programs. In this study we evaluated the AO and OO versions of 3 different systems: Health Watcher (Soares, 2004; Greenwood et al., 2007), Mobile Photo (Figueiredo et al., 2008) and JHotDraw (Marin et al., 2007). For Health Watcher and Mobile Photo two releases were evaluated [13].

These systems represent non-trivial software projects, particularly rich in the ways exception handling concern is implemented: (i) in Health Watcher the exception handling concern related to the crosscutting concerns represented as aspects were also *aspectized*; (ii) in Mobile Photo the exception handling concern related to crosscutting and non-crosscutting concerns was *aspectized* according catalog of best practice defined in (Filho et al., 2007) to guide error handling code modularization; and finally (iii) in JHotDraw the exception handling concern was not *aspectized* - the exceptions thrown by aspects were handled by elements in the base code.

A static analysis tool, called SAFE was developed to support the empirical study (for detailed information concerning the tool implementation see Chapter 5).

The exception flow graphs of each target system were calculated using the SAFE tool, and each exception path was classified to its *Signaler-Handler* relation. After analyzing the tool output we could observe a significant increase in the number of *uncaught* exceptions and exceptions caught by *subsumption* in the AO versions of almost all target systems. In the 6th version of Mobile Photo, for instance, there was no uncaught exception in the OO version, and in the AO version the number of uncaught exceptions represents approximately 57% of the exceptions signaled on the system. In the 9th version of Health Watcher, the percentage of exceptions caught by *subsumption* increased 36% in the AO version compared to the same number in the OO version. Such an increase was least significant in JHotdraw since there were few changes in the way exceptions were caught in OO and AO versions (most aspects contains intertype declarations and all handlers to the exceptions signaled by the methods include by intertype declarations remained in the base code).

The code related to each *exception path* was manually inspected to refine the findings and discover the causes of such discrepancies between the way exceptions were handled in AO and OO versions. We observed that specific characteristics of AO programs (e.g., use of aspects to handle exceptions, use of `declare soft` construct) caused such discrepancies, leading to a more error-prone exception handling code when compared to OO programs. As a consequence, more effort (i.e., use of verification approaches and tools) needs to be expended to assure the robustness of an exception-aware AO system.

Next chapter summarizes the findings gathered during the manual inspections, which includes: a set of bug patterns that were responsible for *uncaught* exceptions and exceptions caught by *subsumption* in the AO versions, and lessons learned and further discussions concerning the development of exception aware AO systems.

[13] The source code of every target system used in this study can be downloaded from: http://www.inf.puc-rio.br/~roberta/aop_exceptions.