2
Background

This chapter defines the basic concepts used in this dissertation and describes the context of the work. The essential concepts of exception handling mechanisms and aspect-oriented programming (AOP) are presented as well as the AspectJ language and other languages that incorporate AO concepts in order to address limitations of object-oriented programming. The way the exception handling mechanisms relate to AO programming is also presented. Finally, a brief introduction about verification approaches for exception handling code is given.

2.1.
Aspect-Oriented Programming

Aspect Oriented Programming (AOP) (Kiczales, 1996; Kiczales et al., 1997) has been proposed as a paradigm for improving separation of concerns in software design and implementation. It proposes a new abstraction, called Aspect, to capture concerns that cannot be easily expressed by the elements of traditional decomposition approaches (e.g., classes, functions); such concerns are usually spread over several system modules and tangled with other concerns.

AOP have been increasingly used to modularize crosscutting concerns such as persistence (Rashid and Chitchyan, 2003; Soares et al., 2006), distribution (Soares et al., 2006), and design patterns (Hannemann and Kiczales, 2002; Garcia et al., 2005). It has been empirically observed that AOP decompositions promotes modularity (Garcia et al., 2005), design stability (Greenwood et al., 2007), and that aspects abstractions can be used to modularize the exception handling concerns in some situations (Filho et al., 2007).

2.1.1.
Aspect Weaving

Aspects has the ability of externally modifying the behavior of programs (Krishnamurthi et al., 2004; Aldrich, 2005). Such modifications can happen

statically – manipulating the structure of program source code – or *dynamically* – using the ability of reflecting on the state of program's execution to conditionally modify it. The composition process between aspects and the elements that composes primary decomposition of a software system (e.g., classes) is called *aspect weaving* (Kiczales et al., 1997), and involves making sure that aspects will affect the appropriate points in the code.

There are no strict rules about when the *aspect weaving* should be done. In current AO languages *aspect weaving* can happen at compile-time, load-time or run-time. The main advantages of compile-time weaving are that it avoids unnecessary runtime overload and that static checks performed during compilation can expose many composition errors. On the other hand it requires that all the affected code (and the code referenced by it) to be present during weaving. When performing load-time weaving, the class loader reads a configuration file that specifies the aspects to be woven when applications are loaded. Finally, the run-time weaving allows advices to be included and removed at runtime. It usually requires the definition of proxies for every element that will be affected by a new behavior, as in Spring AOP (Johnson et al., 2005). Both load-time and run-time weaving bypasses the static checks performed during compilation that could expose many composition errors. On the other hand the aspect weaving can happen on demand, which opens a new realm of possibilities for software composition.

2.1.2. Obliviousness and Quantification Properties

Filman and Friedman (2005) have identified two properties, *quantification* and *obliviousness*, which they believe are fundamental for aspect-oriented programming. The *quantification* property refers to the desire of programmers to write programming statements with the following form: "*In programs* P, *whenever condition* C *arises*, *perform action* A". The AspectJ programming language, for example, supports this property by means of the pointcut, join point and advice mechanisms described next (Section 2.1.4). *Obliviousness* establishes that programmers of the base code (i.e., the classes that will be affected by the aspects) do not need to be aware of the aspects which will affect their code. It

means that programmers do not need to prepare the base code to be affected by the aspects. The following sentence from the authors synthesizes both properties: "*AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers.*" (Filman and Friedman, 2005)

2.1.3. Crosscutting Interfaces (XPI)

Sullivan et al (2005) have compared a development methodology based on the *obliviousness* property with a new AO development approach based on *design rules* (Baldwin and Clark, 2000). In their approach, the authors propose the specification of interfaces between the base code and the aspects, which determine the anticipated exposition of join points from the base code before its implementation. These join points are used subsequently in the implementation of the aspects. The design rule based approach (Sullivan et al., 2005) addresses the decoupling of the base and aspect code by offering a clear specification of the interaction and contracts between them and by allowing their parallel development. In the study, the authors have also observed how their approach helps to reduce or eliminate several disadvantages of the obliviousness approach, such as, the codification of complex and fragile pointcut expressions and the tight coupling of the aspects to changeable and complex details from the base code.

Griswold et al (2006) have recently shown how the interfaces between the base code and the aspects, called crosscutting interfaces (XPIs) and previously proposed by the design rules based approach (Sullivan et al., 2005), can be partially implemented in AspectJ. The XPIs are used to abstract a crosscutting behavior existing in the base code. The implementation of XPIs in AspectJ is composed of: (i) *a syntactic part* – which allows to expose specific join points by specifying pointcuts in aspects; and (ii) *a semantic part* – which details the meaning of the exposed join points and it can also define constraints (such as, pre- and post-conditions) that must be satisfied when extending those join points (Griswold et al., 2006). This semantic part can be partially implemented with enforcement aspects (implemented with `declare error` and `declare warning` AspectJ constructs) (Colyer, 2004) or by defining contract aspects which

guarantee specific constraints that are satisfied before and after the advices execution.

2.1.4. Aspect J

AspectJ (Kiczales et al., 2001a; Kiczales et al., 2001b; Colyer, 2004) is the most used aspect-oriented programming language. It is a general-purpose extension to Java, which allows developers to structure a program using both classes and *aspects* to represent crosscutting concerns. The *aspect* abstraction in AspectJ is composed of: *pointcuts*, *advices*, *inter-type* declarations, and internal methods and attributes. *Join points* are well defined locations in the execution of a program (e.g., method call, method execution). *Pointcuts* are represented as expressions that identify (match) one or more join points in code. Advices are method-like constructions that contain the additional behavior that should be included on the *join point* matched by a *pointcut*. *Inter-type* declarations specify new attributes or methods to be introduced into specific types on the *base code* (the code affected by aspects).

Therefore, *pointcuts* enable a developer to specify when an advice should be executed. One example is the rule "*the performance tracking code should be executed after each call of a void public method*." This rule can be represented by an aspect in AspectJ language, as illustrated in the code snippet below.

```
1. aspect PerformanceTracking {
2.
3.   public pointcut anyMethodCall() :
4.       call(public void *(..));
5.
6.   void around() : anyMethodCall(){
7.     long start = System.currentTimeMillis();
8.      proceed();
9.      long elapsed = System.currentTimeMillis()-start;
10.     System.out.println("Elapsed time:" + elapsed);
11.  }
12. }
```

Pointcut: Identify points in the code that should be intercepted.

Advice: define the behavior to be added on such points

Listing.1. Code snippet an aspect in AspectJ.

The code above illustrates a simple aspect for tracking the performance of every public method call of a program. This aspect, basically, consists of a pointcut that matches every public method call (lines 3-4) and an advice (lines 6-8) that tracks the performance of every method matched by the *pointcut*. Next sections present the main AspectJ constructs in more detail.

2.1.4.1. AspectJ Main Constructs

AspectJ constructs make it possible: (i) to define additional implementation to run at certain well defined points in the *execution of a program,* as illustrated previously in Listing 1 (also known as *dynamic crosscutting* capabilities); and (ii) to change the static structure of a software system by adding class members, changing the classes' hierarchy, or replacing checked by unchecked exceptions (also known as *static crosscutting* capabilities).

The join point model specifies the set of points in the *execution of a program* which can be intercepted by aspects, in other words, the join points that can be referred on *pointcut* expressions (Kiczales et al., 2001b). Some of the well defined points that comprise the join point model in AspectJ are: a call to a method (see lines 3-4 in Listing 1 above), an access to an attribute, an object initialization, and an exception handler.

Besides allowing the developer to specify which methods or fields should be intercepted, AspectJ also provides some pointcut designators specifically for *scoping purposes*. These enable us to extend the pointcut expression with a scope designator so that we can match only a subset of interest. For instance, we can extend the pointcut presented in Listing 1 above in the following way:

```
public pointcut anyMethodCall() :
     call(public void *(..)) && within(MyClass);
```

This pointcut expression will match only the call to public methods within a specific a *class* or interface, called `MyClass` in the code snippet above. Moreover, AspectJ also offers control-flow pointcut designators that enable us to match join points based on what is happening at runtime. The two pointcut

designators that match join points occurring within a given control flow during the runtime execution of a program are: `cflow` and `cflowbellow`.

Table 1 presents the main elements that compose the join point model of AspectJ language, and their corresponding pointcut descriptors. In Table 1 <Signature> is a method or constructor signature, <Expression> is a boolean expression, and <StaticScope> can specifies reference types (i.e., arrays, classes or interfaces) or packages names. A *pointcut expression* is, therefore, defined by composing *pointcut* designators through the use of || (or), && (and), and ! (not) operators.

Pointcut Descriptor	Description
call(<Signature>)	Matches method or constructor call.
execution(<Signature>)	Matches method or constructor execution by Signature
get(<Signature>)	Matches field access by Signature
set(<Signature>)	Matches field assignment by Signature
handler(<Type>)	Matches the handler execution of a specific exception type.
adviceexecution	Matches the execution of every advice
if(<Expression>)	Matches executing code when the Expression evaluates to true
within(<StaticScope>)	Static scope designator that matches a join point arising from the execution of logic defined in specific classes, interfaces or packages defined by the <StaticScope>.
withincode(<Signature>)	Static scope designator that matches executing code defined in the method or constructor with a given Signature
cflow(<Pointcut>)	Dynamic scope designator that matches executing code in the control flow derived from a specific join point (specified by Pointcut)
cflowbelow(<Pointcut>)	Dynamic scope designator that matches executing code in the control flow derived below a specific join point (specified by Pointcut)

Table 1. Main pointcut designators of AspectJ language.

The AspectJ constructs that enable such *static crosscutting* are called *inter-type* declarations. Examples of *inter-type* declarations are the `declare parents` and `declare soft` constructs. The `declare parents` construct declares that a type defined on the base code should implement (or extend) types specified by this construct. For instance, we can add a `declare parents` on the code of the `PerformanceTracking` aspect presented in Listing 1. This construct declares that

`MyClass` implements the interface `MonitoredClass` (line 2), which can be used to restrict the scope of the pointcut (line 4).

```
1. aspect PerformanceTracking {
2.   declare parents:  MyClass implements MonitoredClass;
3.   public pointcut anyMethodCall() :
4.       call(public void *(..)) && within(MonitoredClass);
5. ...
```

The `declare soft` construct converts (wraps) a given exception into a specialized runtime exception, named `SoftException`, in a specific scope. The syntax is as following: `declare soft : <someException> : <scope>`. The `scope` is specified by a *pointcut* which matches a subset of *join points* in which the `someException` will be wrapped. Additional information about static and dynamic crosscutting can be found in the AspectJ Programming Guide (2007).

2.1.5. Other AO Languages

Aspect-oriented programming (AOP) is becoming increasingly popular in the Java environment. Therefore, in addition to AspectJ other languages have emerged. This section presents the characteristics of other mature AO languages, based in Java environment: CaesarJ (Mezini and Ostermann, 2003), JBoss AOP (Burke and Brock, 2003), and Spring AOP (Johnson et al., 2005; Johnson, 2007).

CaesarJ was proposed by Mezini and Ostermann (2003). It is based on the concept called *virtual class*. A *virtual class*, like a virtual method, can have different meanings, which depends on its context of use. Although the *virtual class* concept improves the composition flexibility, the crosscutting capabilities of the aspects defined in CaesarJ are very similar to the ones developed in AspectJ – since it reuses the *join point* model and *pointcut* designators of AspectJ language.

JBoss AOP[2] (Burke and Brock, 2003) was first released in 2003 as an addition to the JBoss application server framework (Fleury and Reverbel, 2003). It uses a XML-based aspect declaration style. In this style aspects are represented as Java classes and the advice bodies are implemented using plain Java methods. The

pointcut expressions, the aspects and the advice signatures are represented on XML files. JBoss AOP allows the run-time weaving described before.

Spring AOP (Johnson et al., 2005; Johnson, 2007) was first released in 2004 as an addition to the Spring framework. Similar to JBoss AOP, the advice implementation in Spring AOP is a plain Java method. The XML declares the special classes called beans which gives the Spring framework access to the application objects and specifies the matching pattern for the code to be intercepted (the *pointcut* expressions).

2.2. Exception Handling Mechanisms

A software system consists in a number of components that cooperates to deliver a set of services. A *failure* occurs when the service delivered by the system deviated from what was specified. An *error* is an abnormal computation state that can lead to a *failure*. A *fault* is the cause of an error - it can be a physical defect on hardware or a flaw on software. Developers of dependable systems often refer to *faults* as *exceptions,* as they are expected to manifest rarely during the system execution.

Modern applications have to cope with an increasing number abnormal computation states that arise as a consequence of *faults* in the application itself (e.g., access of null references), noisy user inputs or faults in underlying middleware or hardware. Currently, the exception handling mechanism (Goodenough, 1975; Garcia and Rubira, 2001) is one of the most used schemes for detecting and recovering such *exceptional conditions*. It enables the developer to structure an application in a way that the code that deals with the *exceptional conditions* will be defined separate from the code that deals with the normal execution flow of the program. The separation between normal and exceptional flow of a program, makes developers to systematically think beforehand of erroneous conditions that may happen during program execution, and then prepare the software to deal with them.

The use of exception handling in the construction of several real-world systems and its addition in many mainstream programming languages, such as

[2] http://www.jboss.org/jbossaop

Java, Ada, and C++, attest its importance (Garcia and Rubira, 2001). However, there are small differences in the way exceptions are represented and handled in each language. This thesis focus on the exception handling mechanism implemented in Java language - the one also adopted in AspectJ. Next sections briefly describes how the exception handling mechanism of Java language works and relate each element of such mechanism with the AO main concepts.

2.2.1. Exception Handling Mechanism in AspectJ Programs

In order to support the reasoning about the exception flows in aspect-oriented programs we present the main concepts of an exception handling mechanism implemented in Java language and correlate them with the constructs available in most AO languages (Coelho et al., 2008). An exception handling mechanism is comprised by four main concepts: the exception, the exception signaler, the exception handler, and the exception model which defines how signalers and handlers are bound.

Exception Signaler. An exception is raised by an element - method or method-like construct e.g., advice - when an abnormal computation state is detected. Whenever an exception is raised inside an element that cannot handle it, it is signaled to the element's caller. The exception signaler is the element that detects the abnormal state and raises the exception. In Figure 1, the advice a1 detects and abnormal condition and raises the exception EX. Since this advice intercepts the method mA, such exception will be included into method mA together with the additional behavior encapsulated on the advice.

Exception Handler. The exception handler is the code invoked in response to a raised exception. It can be attached to protected regions, e.g. methods, classes and blocks of code, or specific exceptions (Garcia and Rubira, 2001). The exception handlers are responsible for performing the recovery actions necessary to bring the software system back to a normal state and, whenever this is not possible, to log the exception and abort the system in an expected safe way. In AO programs, a handler can be defined in either a method or an advice. Specific types of advice (e.g. `around` and `after` advice (Colyer, 2004))have the ability to handle the exceptions thrown by the methods they advise.

Exception Model. In many languages, the search for the handler to deal with a raised exception occurs along the dynamic invocation chain. This is claimed to increase the software reusability, since the invoker of an operation can handle it in a wider context (Goodenough, 1975). In AO programs the handler of one exception can be present:

(i) in one of the methods in the dynamic call chain of the signaler; or

(ii) in an aspect that advises any of the methods in the signaler's call chain.

Figure 1 depicts one exceptional scenario in which one advice (a1) is responsible for signaling the EX exception, and other advice (a2) is responsible for handling EX, i.e. a2 intercepts one of the methods in the dynamic call chain and handles this exception.

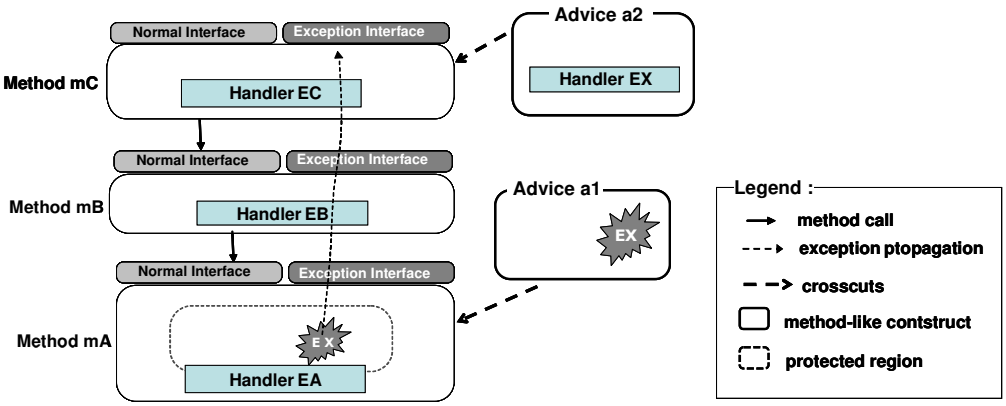

Figure 1. Exception-aware method call chain in AO programs.

Along this thesis we call *exception path* a path in a program call graph that links the signaler and the handler of an exception. Notice that if there is not a handler for a specific exception, the exception path starts from the signaler and finishes on program entrance point. In Figure 1, the exception path of EX is <a1→mA→mB→mC→a2>. Therefore, the *exception flow* comprises three main moments: the exception signaling, the exception flow through the elements of a program, and the moment in which the exception is handled or leaves the bounds of the software system without being handled (becoming an uncaught exception).

Besides these three main concepts that compose an exception handling mechanism, some other concepts related to the Java exception handling mechanism - that will be used along this work - are described bellow:

Exception Interfaces (Miller and Tripathi, 1997). The caller of a method needs to know which exceptions may cross the boundary of the called one. In this way, the caller will be able to prepare the code beforehand for the exceptional conditions that may happen during system execution. For this reason, some languages provide constructs to associate to a method's signature a list of exceptions that this method may throw. Besides providing information for the callers of such method, this information can be checked at compile time to verify whether handlers were defined for each specified exception. This list of exceptions is defined by Miller and Tripathi (1997) as the *exception specification or exception interface* of a method. Ideally, the *exception interface* should provide complete and precise information for the method user. However, some languages, such as Java and AspectJ, allow the developer to bypass this mechanism. In such languages exceptions can be of two kinds: *checked* exception – that needs to be declared on the method's signature that throws it – and *unchecked exception* – that does not need to be declared on the signaler method's signature[3]. As a consequence, the client of a method cannot know which *unchecked* exceptions may be thrown by it, unless s/he recursively inspects each method called from it. For convenience, in this thesis we split this concept of exception interface in two categories:

(i) the explicit exception interface that is part of the module (method or method like construct) signature and explicitly declares the exceptions; and

(ii) the complete (de facto) exception interface which captures all the exceptions signaled by a module, including the implicit ones not specified in the module signature.

In the rest of this thesis, unless it is explicitly mentioned, we use the expression "exception interface" to refer to a complete (de facto) exception

[3] In some situations, to list all the exceptions that may scape from a method in the throws clause may become unworkable. Some exceptions, for instance, cannot be adequately handled inside the program (e.g., out of memory exception). Forcing the developer to list all of them could lead to unnecessary work during development and maintanence tasks.

interface. Although both the normal interface (i.e. method signature) and the exception interface of a method can evolve along software life cycle, the impact of such change on the system varies significantly. When a method signature varies, it affects the system locally, i.e. only the method callers are directly affected. On the other hand, the removal or inclusion of new exceptions in an exception interface may impact the system as a whole, since the exception handlers can be anywhere in the code. As depicted in Figure 1, an aspect can add behavior to a method without changing the normal interface of that method. However, the additional behavior may raise new kinds of exceptions, hence impacting the exceptional interface of that method.

Exception Types and Exception Subsumption. Object-oriented languages usually support the classification of exceptions into exception-type hierarchies (Miller and Tripathi, 1997; Garcia and Rubira, 2001). In Java the exception interface is therefore composed by the *exception types* that can be thrown by a method. Each handler is associated with an *exception type*, which specifies its handling capabilities – which exceptions it can handle. The representation of exceptions in type hierarchies allows type *subsumption* (Miller and Tripathi, 1997; Robillard and Murphy, 1999) to occur: when an object of a subtype can be assigned to a variable declared to be of its supertype - the subtype is said to be subsumed in the supertype. When and exception is signaled, it can be subsumed into the type associated with a handler, if the exception type associated with the handler (i.e., *the hander type*) is a supertype of the exception type being caught.

Exception Handling Contexts. The *exception types* are always treated in the same way at specific regions in a program called Exception Handling Contexts (EHC) (Goodenough, 1975). Each EHC is associated with one or more handlers, which are responsible for handling exceptions from a given time. Therefore, when an exception is signaled inside an EHC a handler is chosen among the handlers associated to the EHC according to the type of the signaled exception.

2.2.2. Exception Handling Constructs in AspectJ

In AspectJ as in Java, `try` blocks define exception handling contexts, `catch` blocks define the exception handlers, and `finally` blocks define clean-up actions

- executed whether or not exceptions are raised (Gosling et al., 1996). Exceptions are represented in a hierarchical structure, as illustrated in Figure 2.

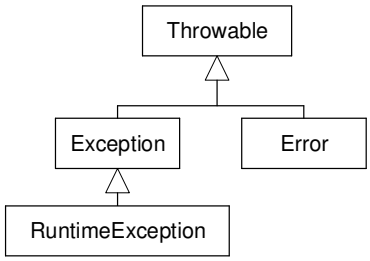

Figure 2. Exception Hierarchy in Java.

According to this structure every exception is an instance of the `Throwable` class. The user defined exceptions can be represented as a *checked* (extends `Exception`) or an *unchecked* exception (extends `RuntimeException`). By convention an `Error` represent an unrecoverable condition, they usually represent platform problems (Gosling et al., 1996; Robillard and Murphy, 2003).

Checked exceptions must be declared as part of the method signature that propagates it. The use of checked exceptions allows the compiler to statically check that appropriate handlers are provided within the system. The use of checked exceptions, however, is costly to maintain (Dooren and Steegmans, 2005) since every method on the call chain of a method that raises a new exception should be updated to declare this exception (on the `throws` clause defined on its signature) or handle it.

On the other hand, the unchecked exceptions do not need to be declared on the interface of their signalers, but as a consequence there is very little that can be checked at compile time. The client of a method cannot easily know which unchecked exceptions may be thrown by the method unless he/she carefully inspects the code of the method and the methods called from it – which can become a very time consuming or infeasible task. Moreover, the developer is not warned by the compiler if an unchecked exception is not handled inside the application. When libraries are used, the developer does not have access to their source code and thus needs to rely on the library documentation about the runtime exceptions that should be thrown – which, more often than not, are neither complete nor precise (Thomas, 2002; Sacramento et al., 2006). As a consequence,

unhandled unchecked exceptions can be seen as one of the major sources of bugs in current Java systems (Jo et al., 2004).

AspectJ reuses the same Java constructs to raise (throw statement), handle exceptions (`try-catch-finally`) and specify exception in the method signature (`throws` clause). In AspectJ the *exception interfaces* of advice must be based on exception interfaces of the advised methods. It should follow a rule similar to the "Exception Conformance Rule" (Matsuoka and Yonezawa, 1993; Miller and Tripathi, 1997) applied during inheritance, when methods are overridden. As a result an advice can only throw a checked exception if it is thrown by "every" intercepted method. To overcome this limitation most of the advices throw *unchecked* exceptions[4] which do not need to be specified by every advised method. In other AO languages such as Spring AOP and JBoss AOP aspect advices are represented as regular Java, methods, which can throw any exception (*checked* or *unchecked*).

Some of the AspectJ constructs can be used to handle exceptions, as presented below:

- *Handler Pointcut Designator*. It provides a pointcut designator that allows an aspect to advise the places where specific exceptions are handled, and associate a specific handling task such as logging the exception being handled.
- *Advice After and After Throwing*. These kinds of advice allow aspects to be invoked when an exception is thrown by a method. They allow extra code to be executed when an exception is signaled. Such code may for instance wrap the original exception in a new one – however the code needed to perform such wrapping can become complex in some scenarios.
- *Around Advice*. This advice wraps the body of the advised method, and is able to include additional behavior before and/or after it, or even replace the method body. This advice can be used to handle exceptions thrown by the method and return the program to its normal control flow.

[4] Sometimes and advice throws a checked exception that is wrapped by an AspectJ specific unchecked exception (i.e., `SoftException`) through the use of the `declare soft` construct.

2.3. Checking the Reliability of Exception Handling Code

Despite its importance, several studies have shown that exception handling code is often the least well understood (Sinha and Harrold, 1998; Robillard and Murphy, 2000; Robillard and Murphy, 2003), documented (Cabral and Marques, 2007) and tested (Sinha and Harrold, 1999; Fu and Ryder, 2005) part of the system. The reasons are twofold. Firstly, since the exception handling code is not the *primary concern* to be implemented, it does not receive much attention during system design, implementation, and testing – usually during testing almost all the attention is paid to the "normal flow" of the program. Secondly, testing exceptional code is inherently difficult due to: (i) the difficulty to *simulate* the causes of exceptional conditions during tests, and (ii) the huge number of exceptional conditions that can happen in a system – which may lead to the *test-case explosion* problem (Myers, 2004; Bruntink et al., 2006). The following sections briefly describe possible verification approaches for the exception handling code.

2.3.1. Checking the Reliability through Testing

One way of checking the reliability of the exception handling code is *simulating* the exception occurrences during tests by *fault injection* strategies (Fu et al., 2004 ; Fu et al., 2005). In such strategies extra code, that throws exceptions, is injected at specific points in the code at compilation time. The test cases are therefore responsible for checking whether the exception handling code is executed as expected. Alternatively, the developer can simulate exception conditions through the use of *mock objects*. A *mock object* is a test pattern (Binder, 1999) proposed by Mackinnon et al. (2001) which replaces domain code with a dummy implementation that emulates real code. *Mock objects* can be used to *easily* simulate real objects behaviors that are hard to trigger, such as: a network error, faults derived from the interaction between the system under test and specific hardware or middleware (Mackinnon et al., 2001). They have been largely used in the development of automatic unit tests in OO systems.

The testing approaches require many faults to be simulated, and every exception path to be verified. The number of *test-cases* created per signaled exception, may make this technique prohibitive when testing the exception handling code of medium-sized systems - since the number of exceptions that can be possibly thrown inside such systems is usually large.

2.3.2. Checking the Reliability through Static Analysis

The basis for static analysis is the desire to offer static compile-time techniques to predict behaviors arising dynamically at run-time when executing a program. The static analysis techniques have often been used in compiler design and have recently started to be used in the validation of software (Louridas, 2006).

Recently, approaches based on static analysis have been proposed to improve the reasoning about exception flow (Robillard and Murphy, 2003; Fu and Ryder, 2005) and assure the reliability of exception-related code (Bruntink et al., 2006). Bruntinik et al (2006) implemented a static checker called SMELL capable of statically detecting violations on exception handling policies of idiom-based systems developed in C. SMELL performs an intra-procedural static analysis in each function in order to find faults on exception handling code.

Other works (Robillard and Murphy, 2003; Fu and Ryder, 2005) propose algorithms and supporting tools for the inter-procedural analysis of exception flows. The exception-flow analysis is a dataflow analysis, resembling the analysis for finding *def-use* pairs (Myers, 2004), but instead of running on the control-flow graphs, it runs on the program call graph. Continuing the parallel between *exception-flow analysis* and def-use algorithms, the place in which an exception in thrown corresponds to the variable definition, and the place where it is handled (by an enclosing `try-catch` block) corresponds to the variable use.

When an exception is thrown it propagates backwards on the program call graph until a handler is found. Such tools discover if no handler is defined for an exception - if the exception reaches the program entrance point it is classified as an *uncaught exception*. During the exception propagation analysis, each method (a node in the call graph) in which the exception propagates is recorded as part of the

exception path and such an exception will be one of the exceptions that compose the *exception interface* of the method.

It is cheaper to check exception handling rules conformance statically, than creating test-cases to exercise every exception. However, the solutions proposed so far to statically analyze the exception paths are both *unsound* and *incomplete* (Chang et al., 2001). A complete checker would find all exception paths (and possibly the bugs related to them) and if it was sound it would report only real paths. Since such approaches are based on conservative analysis (Rountev et al., 2004) they can generate false positives. A false positive occurs for example when a tool reports an infeasible exception path (i.e., a path that will never be executed in runtime – common in most conservative static analysis).

However, such properties do not necessarily harm the usefulness of such tools given that the tools still allow the developers to analyze large number of exception paths (Robillard and Murphy, 2003; Fu et al., 2005) and detect the bugs on it (Bruntink et al., 2006) which would be very costly to detect manually. Experimental results have shown that the precision of such tools are currently within acceptable margins (Robillard and Murphy, 2003; Fu et al., 2005).

2.4.Summary

Aspect oriented programming (AOP) was proposed as a way to modularize (Kiczales, 1996; Kiczales et al., 1997) crosscutting concerns, aiming at increasing software maintainability, extensibility and reuse. AspectJ (Kiczales et al., 2001a; Kiczales et al., 2001b) is the most used aspect-oriented programming language developed on top of Java. Besides AspectJ other AO languages have been proposed, most of them are very similar and follow a *join point model* very similar to the one proposed by AspectJ.

Empirical studies have shown that AOP can indeed promote modularity (Garcia et al., 2005), design stability (Greenwood et al., 2007), and that aspects´ abstractions can be used to modularize the exception handling concerns in some situations (Filho et al., 2007). However such empirical studies do not account for the exceptions that may flow from aspects, or the ones that are caught by aspects, and what consequences they may bear.

An *uncaught* (Jo et al., 2004) exception thrown during advice execution impairs all system functionality. So, any advice capable of throwing an exception is potentially error-prone. Aspect developers may do their best to avoid exceptions from flowing through aspects (ex: avoiding explicit throw statements inside advices). However, any library called by an advice may throw an unchecked exception, and also an aspect may use a huge data structure that may cause a memory overflow, which may result in `OutOfMemoryError` - whereas no such error would be thrown without the aspect. The absence of empirical studies to evaluate the consequences of exceptions thrown by aspects, and the lack of approaches to help developers when assuring the quality of exception-related code in AO systems compose the motivation of this work described in next chapters.