

Università degli Studi di Genova  
Dottorato in Informatica  
(XIX Ciclo)

Thesis Progress Report

---

UML based AOSD and Join Point Model  
for Software Evolution

---

Candidate: Sonia Pini,  
Department of Computer and Information Sciences  
Università degli Studi di Genova

Advisors: Prof. Massimo Ancona,  
Department of Computer and Information Sciences  
Università degli Studi di Genova  
*ancona@disi.unige.it*

Dott. Walter Cazzola,  
Department of Informatics and Communication  
University of Milan.  
*cazzola@dico.unimi.it*

December 2005



# Contents

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Current State of Research</b>	<b>5</b>
2.1	Synchronizing design and implementation . . . . .	5
2.1.1	How to maintain consistency between design and implementation . . . . .	6
2.2	Joint Points Definition . . . . .	7
2.3	Pointcut mechanism . . . . .	11
2.3.1	Structural Pointcuts . . . . .	12
2.3.2	Behavioral Pointcuts . . . . .	12
<b>3</b>	<b>Work Plan</b>	<b>17</b>
<b>4</b>	<b>Structure of the Thesis</b>	<b>19</b>
<b>A</b>	<b>The Pointcut Language</b>	<b>21</b>
	<b>Bibliography</b>	<b>23</b>



# Introduction

---

An intrinsic property of a successful software application is its need to evolve. In order to keep an existing application up to date, we continuously need to adapt it. The law of software evolution [12] said:

”a program that is used in a real-world environment must change, or became progressively less useful in that environment”.

Usually, evolving an application requires it to be shut down, however, because updating it at runtime is generally not possible. In some cases, this is beyond the pale. The unavailability of critical systems, such as web services, telecommunication switches, banking systems, etc. could have unacceptable financial consequences for the companies and their position in the market. Since, maintenance and evolution of continuously running systems have become a major topic in today’s market of embedded systems, mobile devices, and service infrastructures.

A common solution to make highly important, mission-critical systems run non-stop during evolution, is to design them in a special way and run on a specially configured, redundant hardware. This solution is complex and expensive, and many users would a relatively easy-to-use and general-purpose technology supporting runtime changes to applications.

However, when a system evolves, it becomes more complex and automatic techniques to support these phenomena are fundamental to improve the management of unanticipated software evolution and the software efficiency.

To most people software is the code that is the end result of the software development process. When a company starts developing a new product, it typically uses a clean straightforward engineering scheme and goes through requirements analysis, high-level design, design and implementation phases. This development process changes when a first implementation is finished. From then on, the implementation receives more and more attention at the cost of the maintaining the higher-level artifacts.

The design process is very important to the usability and understandability of the system, for example functional requirements present a *complete* description of how the system will function from the user’s perspective, while non-functional requirements dictate properties and impose constraints on the project or system. At the beginning design view and implementation view are consistent since one

is derived and developed from the other, and we must preserve the correlation between the two views, and this correlation must exist for all the software life-cycle.

Often, during the evolution and maintenance phase a discrepancy between the two view can occur, because the initial stages of development are ignored once the code has been developed. The main problem is the fact that models are shown like only an intermediate step in the software development life-cycle, and this practice causes several problems when the system must evolve, because the evolution of only one view of the system causes a *gap* between them, that could create confusion, misunderstanding and mistakes.

The term *gap* has been stated by Rumbaugh [15]

”... too often, there is still a gap between concept and execution.”

as a problem area in this retrospective review of O-O methodology. In the past developers have tried different ways to link design to implementation, and to tackle this gap problem, but to date, no existing, general purpose methodology for this issue. When the change is not applied also to the design view, it is hard for manager, programmer and customer to have the opportunity to plan future directions, goals, schedule and the necessary budget, since the design view could not provide an immediate and understandable global view of the system consistent with the code. Moreover, it also hinders the integration of new functionality. When there is a gap between the views it becomes difficult or impossible to know all the necessary changes to apply, therefore the evolution cannot be planned *in-the-large*. To simplify the evolution process, it is necessary to have a *global view* of the system to apply all the evolutionary steps.

In our point of view, the global view may be well represented by the design information, because it is usually graphic, and more intuitive and understandable than the code. Moreover, the design information can be used to drive software evolution and consequently to preserve the consistence among design and code views.

We propose an infrastructure to dynamic adapt software system, called RAMSES [1]. Our project involves a reflective middleware whose aim consist of consistently evolving software systems, both design information and implementation code, against runtime changes. This middleware provides the ability to change the system at runtime without stopping it, by using its design information, and consequently preserving consistence between design and code.

*Our scope is to simplify the evolution/maintenance mechanism. That is, to render the changes required by the evolution immediately available both to the design models and to the implementation, all that we will have as direct consequence the maintenance of the consistency between the design and the code.*

How said into the Thesis Proposal [14], the evolution process can be divided into two different phases, that we call *planning phase* and *deployment phase*. The

---

planning of the evolution in out of the scope of this thesis, it will be faced in the thesis of Ahmed Ghoneim supervised by W. Cazzola. Instead, we will focus our efforts on the deployment phase.

To develop our thesis we have chosen to use:

- the UML to describe design information, because UML is *de facto* the standard (graphical) language used during the design process;
- the JQVQ programming language because, is becoming increasingly popular in many areas, including server-side application. Therefore, developing a mechanism that would support runtime changes of JQVQ applications is a quite natural step. This is greatly facilitated by the fact that in the classic case JQVQ applications are initially compiled into machine-independent bytecodes, which run on top of a Java Virtual Machine (JVM). Compared to a statically compiled application, any JVM already has much of the infrastructure that would be needed to inspect and change running applications. Bytecodes are also much easier to analyze and instrument than machine code; and
- the meta-data facility (JQVQ 1.5 annotation) and Aspect-Oriented (AO) technique like mechanisms for representing design information in JQVQ, because the combination of annotations, semantic reasoning, and AO approach seem to be the ideal solution to represent design information in object-oriented languages.

The main goal of my thesis is to define a clear and appropriate definition of the join point model (JPM) to realize our scope. Following the general definition given in [13] a JPM consists of three elements:

- the *join points*, which are the points of reference that aspect can use to refer to the computation of the program;
- a means to identifying join points, this mechanism is called *pointcut* mechanism,
- a means to specifying the new/or additional behavior that should run at join points, this mechanism is called *weaving* mechanism.

The JPM is a critical element in the design of any aspect-oriented language mechanism. This model provides the common frame of reference that makes it possible for execution of a programs aspect and non-aspect code to be properly coordinated.

In the rest of the document, after discussing the obtained results with respect to the thesis proposal [14], we describe what are the main research topic that we plan to investigate next year.

The document ends with a scheme of how we plan to structure the thesis.



## Current State of Research

---

In this section we discuss how the work described in the PhD thesis proposal [14] has been carried out in the past year.

First, we have analyzed and highlighted the role of the design information in software evolution, and the results of this previous work has been presented at 2nd ECOOP Workshop of Reflection, AOP and Meta-Data for Software Evolution, with the paper [2]. Last, we have faced two of the three elements that compose the JPM, i.e. the *join points* and the *pointcut* mechanism.

In order to better explain our progresses in the past year, we can divide our recent results in three groups.

- The necessity of *co-evolution*, and in particular the need of synchronizing design and the implementation during all software life-cycle.
- The introduction of a new mechanism which allows programs or programs executions to be abstracted to *points of interests*, i.e. points where information is needed from points where new behavior has to be inserted, to realize the evolution.
- Pointcut mechanism, where pointcuts are not simple predicates on join points, but they can be "sample" UML diagrams.

### **2.1** Synchronizing design and implementation

Currently, the idea that software is not just source code is well accepted. Software is multidimensional, the requirements, the analysis, the design, and the implementation are all part of the software system. Software development and evolution are strongly influenced by the relation of all the different aspects of the software.

The main problem is that there are not tools for managing the various dimensions of software, and consequently the result is that different software artifacts representing different dimensions tend to evolve at different rate and in different ways. To support a correct software evolution we need a mechanism to *synchronize* changes between dimensions. In particular, in our project we focus only the *design* and *implementation*.

When implementation and design evolve in different ways and directions because they are not explicitly related, it is introduced the problem known as *drift/erosion* [11].

For complex systems, the need to carefully manage system evolution is a critical task, adapting such software can be very difficult, because of the software size and complexity, moreover, when there is a gap between the views it becomes difficult or impossible to know all the necessary changes to satisfy the evolving requirements.

Our idea consists of rendering systems' evolution more simple and feasible using design information to have a global view of the system, to specify the evolution and to drive the changes into the code, maintaining the system dimensions (design and implementation) synchronized.

The design information we consider can be divided into two categories: system structure and behavior.

- Structural Design Information is an explicit description of the structure of the system.
- Behavioral Design Information describes the computations and the communications carried out by the application objects, e.g. we consider object behavior, collaboration between objects, state of the objects, and so on.

Structure and behavior of the system are modeled by class/object diagrams, sequence, collaboration diagrams and state diagrams. These diagrams can equally well describe the system evolution, helping to plan and integrate the required changes within the existing code. The original design information (in our case UML diagrams) can determine the code to be adapted, the evolved diagrams specify how the code has to be adapted, and moreover the difference between such diagrams before and after the evolution represents the evolution itself.

Our approach to maintain consistency between design model and code consists of a framework where the evolution can be described through UML diagrams and automatically planned and mapped into the code without stopping the system.

### **2.1.1 How to maintain consistency between design and implementation**

Our goal, consists of transforming system models and code to implement required modifications, and propagating the transformation effect across the views, this can be faced by using a model-driven approach [7], i.e. to tackle the problem of evolving complex software systems by raising the level of abstraction from source code to models, and then maintain the consistency between model and code. Since software designers think about evolution at the design level, and since design information provides an immediate and understandable global view of the system, it is quite natural to exploit the UML and its unified meta-model for expressing system evolution too.

---

We could use, in first phase of the evolution (*planning phase*), horizontal transformation [6] on design models to describe the evolution of the system, and vertical transformation [6] in last phase of evolution (*deployment phase*) to propagate the evolution at the implementation in order to maintain consistency among the models. A vertical transformation results in a target model that is at a different level of abstraction, a typical example is transform higher-level, more abstract, specification (e.g. a UML design diagram) into a lower-level, more concrete, one (e.g. JAVA program). A horizontal transformation results in a target model that is at the same level of abstraction as the source model.

Graph transformation seems to be a suitable technology and associated formalism to specify and apply model transformations for the following reasons: graphs are a natural model representation; and the graphs transformation theory provides a formal foundation for the automatic application of model transformations. In particular, Abstract Syntax Trees describes the source code, while UML diagrams are represented as graph, conforming to the abstract syntax presented in the UML metamodel.

The evolution code can be automatically generated from UML diagrams analysis using vertical transformation.

## **2.2 Joint Points Definition**

Aspect orientation has been created with separation of crosscutting concerns in mind and thus would seem to be the ideal candidate for supporting the evolutions that is a typical crosscutting concerns, difficult to be achieved by framing.

Join points are the points in the computational flow used by aspect oriented programs to refer to the computation of the whole program.

Nowadays many programming languages support the aspect-oriented methodology; `AspectJ` [9] is the leading AOP approach and dictates most of the related terminology. Therefore, it can be considered as one of the most relevant frameworks supporting this methodology.

The join point represents the key concept in aspect orientation. Specifying a set of join points is a major task for aspect-oriented designers, and providing a suitable representation for join points is a primary task for an aspect-oriented language.

Notwithstanding the role played by the pointcut definition languages, they are not always adequate to the situation, there are situations in which the desired join points cannot be accurately described in the pointcut language or simply their description needs information not available at weaving time. Traditionally, in aspect-oriented languages it is possible to select join points only on a lexical base such as on using the explicit names of program elements. As stated by Kiczales in his keynote at AOSD 2003 [8], the pointcuts definition language has the most relevant role in the success of the aspect-oriented technology but most of them rely on a mechanism too tailored on the syntax of the program to manipulate.

The AspectJ pointcut language offers a set of *primitive pointcut designators*, such as `call`, `get`, and `set` specifying a method call and the access to an attribute. These primitive pointcut designators can be combined using logical operations forming more complex pointcuts.

Unfortunately, this mechanism introduces an high coupling between aspect and base code, reducing the re-usability of aspects, and the flexibility of the mechanism. This issue is known as the *problem of the fragile pointcuts* [10].

Neglecting the pointcuts' fragility aspect, they are still not usable in all possible contexts. There are some computational patterns that cannot be captured by a pointcut. The sequence of two or more calls in the most simple example of this problem.

In general, it is difficult to use complex patterns to recognize the join points but the difficulties increase when the pattern we are looking for is based on the properties of computational flow and does not rely in the syntax of the base code itself. Decoupling the pointcuts from the base code representation is essential to modularize a general and then reusable concern and to free the aspect modularization from the syntactic limits of the programming languages.

We perfectly agree with Kiczales [9] that next enhancement for the AOP methodology consists of extending the pointcut definition language to support join points selection on the basis of a (kind of) semantic query, i.e., pointcuts that does not only take into consideration the structure of the program but also its semantic.

Providing a more expressive and semantic-oriented selection mechanism means to use a query language based on a well-defined program representation that captures its behavior/properties abstracting from the syntactic details. We think *design information*, offers the most suitable representation of a program. The design information is the result of the designing phase and it describes behavior and the interrelations inside of the code in an intuitive often graphical, way. It can be complete and independent of the implementation details.

To have a connection between the code and its specification is fundamental for weaving the advice into the identified join points. The most grounded mechanism would consist of embedding the mapping between code and design directly into the code and retrieving it at weaving time through reflection. Given a mapping between the code and its specification it could be automatically embedded by decorating/annotating the code (or better the bytecode) with the mapping. Then a pointcut will be evaluated on the decorations and not directly on the code.

The join points selection incorporates references to the design information. We have to select program portions based on the annotated design information.

There are several ways about how design information be associated to program elements:

- manually attach them as custom annotations;
- automatically attach them by reasoning about the semantics of the program and the design information.

---

We have chosen last option to associate design information to program element, because this mechanism must be transparent to the user.

To implement our proposal, we define the concept of *semantic annotation*, an annotation that allows us to assign a particular semantic and design information to a part of the code.

As already stated, we have chosen the JAVA programming language as our development language. Since JAVA 5 incorporates the concept of custom annotation, which consists basically of markers that identify metadata related to the code. Unfortunately, JAVA annotations can decorate exclusively the declarations, whereas we need a more flexible annotation mechanism with a finer granularity. In practice, the current granularity of the JAVA metadata facility hinders our work. In the past year we built an extension to JAVA, named @JAVA, to overcome this limitation in the hope of adopting it in the development of this thesis.

To arbitrary annotate program elements such as blocks, individual statement, and expressions it is necessary to extend the annotation syntax. @JAVA minimally extends JAVA to support a finer level of granularity for code annotation:

- the definition of new annotation types still remains the one provided by Java; this means that from a syntactic point of view there is not difference between our kind of annotation and the standard ones; the single difference is that our annotations do not extends the Annotation Class but CodeBlockAnnotation Class.
- @target supports a new kind of program element (BLOCK) as a target for an annotation type; all the other meta-annotations are unchanged;
- declarations are annotated as in JAVA 5.

The only real difference from the standard mechanism is related to the possibility of annotating a code block. In this case, the annotation must precede the first statement of the block to annotate and the whole block (that can also be composed of a single statement) must be grouped by braces to denote the scope of the annotation. Code block annotations can also be nested.

The following code well exemplifies the syntax and the scope of the code block annotation in @JAVA:

```
public class AnnotatedClass {
    ...
    public void test (boolean e) {
        if (e) ThenAnnotation { ... }
        else
            ElseAnnotation(value="else branch") {
                try {
                    SecondLevelAnnotation {
                        ...
                    }
                } catch(Exception e1) { ... };
            }
        ...
    }
}
```

← nested annotations

9

To work on the code block annotations @JAVA provides a simple reflective API that permits of checking the presence of an annotation and if any to reify it and access to its members. Form the point of view of the reflective API, code block annotations are associated to the method whose body contains the annotated code block and can be retrieved through its reification.

The real difference between an annotation type associated to a method declaration and a code block annotation type is the scope of the annotation type, in the former case it covers the whole body whereas in the latter just few statements are annotated. The CodeBlockAnnotation class implicitly provides the programmer with iterators on the annotated code block and information on its position.

```
public Iterator <Byte[]> bytecodeIterator ();  
public Iterator <String> codeIterator ();  
public int startBytecode ();  
public int endBytecode();  
public int start ();  
public int end ();
```

Code block annotations enables the code inspection at two different levels:

- at source code level, if this is available, and
- at bytecode level.

Intercession can be realized through external tools such as OpenJava [3] and BCEL [4] that directly work on the source code and on the bytecode, respectively. The annotated code block is also identified from its position inside the source code and inside the bytecode. This information is normally coded inside the class bytecode and can be retrieved invoking the corresponding API. The annotated code position is necessary in conjunction with intercession because enables code injection and extrusion at the given point.

To be able to run on the standard JAVA virtual machine and to use all the bytecode rewriters and instrumentors available, we have kept @JAVA compatible to the standard JAVA at bytecode level. Therefore, the @JAVA compiler transforms the code block annotations into standard annotations to get two achievements: i) to avoid to generate a specific and non-compatible bytecode, and, ii) to render the code efficient as the one without annotations.

The basic idea consists of extending the JAVA syntax to support code block annotations and to use a specific parser that translates this kind of annotations to standard ones. Therefore, a code block annotation is removed from its position and promoted to annotate the declaration of the method. To preserve the original scope of the code block annotation, we have used bytecode transformation to add two fields to the code block annotation (that, as we have said before, is also a standard annotation) to indicate the beginning and the ending of the annotated block. These two new fields are also used to preserve the binding between the blocks and the corresponding annotations. Of course, both translation and the enrichment with two

---

fields only regards the annotation classes that represent a code block annotations, i.e., that extend the `CodeBlockAnnotation` class.

The `@Java` compilers is just a source to bytecode translator that:

- substitutes the code block annotations with normal annotations;
- adds to the class of the removed code block annotations two fields (`begin()` and `end()`);
- modifies the annotation by adding the actual value for the added fields.

### **2.3 Pointcut mechanism**

Pointcut mechanism serves to select to which join points an aspect will be woved. More elaborated pointcut descriptions are needed to quickly and easy pick up all join points involved in the evolution process. Expressive pointcut primitives which identify join points based on higher level information, offer solutions to the poor-ness of expression of aspects languages by allowing programmers to write pointcuts that reflect their intention more straightforwardly. Such pointcut descriptions are called *Semantic Pointcuts*. Notwithstanding that, the semantic pointcuts have a drawback: they tend to be application-specific or to require heavy program analysis.

Currently, in the AO systems the access to non syntactical or structural information is limited. Our proposal defines more types of pointcuts, so that opening up the language to allow more expression.

To realize a semantic mechanism we base the pointcut language on design models . Then, in our proposal, pointcuts are UML diagrams, and consequently join points are points in these diagrams (expressed by using the `«joinpoint»` stereotype ), and the aspects weaving fall into the weaving of models (i.e. UML diagrams).

With a simple UML diagram it is possible to indicate code fragments that should be refactored. Moreover, to automatically suggest (and possibly execute) refactorings based on a set of detected code smells [5], a UML-based approach should present a meta-model that is adequate for describing such code smells.

For example, by using an activity diagram we provide a pattern that could be found in the computational flow of the program, and we can identify a set of join points in this diagram. Every time a certain pattern is recognized, the related join points are picked up to apply the correspondent advice.

We consider two different kinds of pointcuts: structural and behavioral.

- *Structural Pointcut* where interface, methods, and fields can be added to existing classes. We have the advice model and we need to specify how this model will be merged into the application.

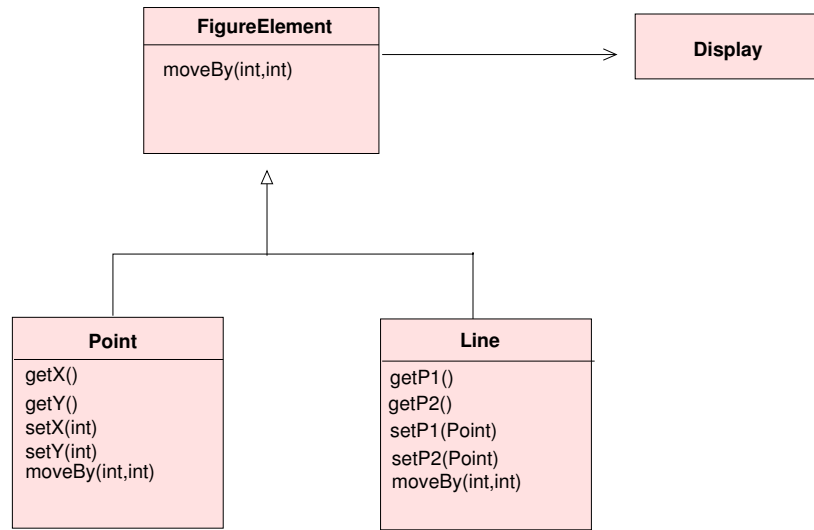


Figure 2.1: Figure Showing Example: Class Diagram

- *Behavioral Pointcut* where we identify computational flows (as a pattern) using UML behavioral diagrams for modeling them, and using the stereotype `<<joinpoint>>` to identify the join points inside these diagrams.

### 2.3.1 Structural Pointcuts

Currently, this kind of pointcuts have not been developed, first in order to simplify the presentation, last because these pointcuts are little interest since they are not involved into the dynamic bytecode acknowledgment.

### 2.3.2 Behavioral Pointcuts

Our pointcut language is based on UML diagrams, but we are developing a textual version of the language, essentially based on the `JAVAC` syntax to unify both behavioral and structural diagrams. Actually, we have studied only 3 kinds of UML behavioral diagrams to identify join points: the activity diagrams, the sequence diagrams and the statecharts. Mainly we are concentrating our efforts on activity diagrams. We have started with activity diagrams because they express the computation flow at more low level than other kinds of diagrams and moreover they are near to the code and therefore they can be directly and easily mapped on the code of the application. For better explain our pointcut language we introduce the implementation of a simple graphical figure element example shown in Fig 2.1, and in Listing 2.1 and later we will see the definition of some pointcuts on such example. We have chosen this example because it is a typical AO example used by Kiczales et al. in a lot of AO paper. Each shape has its own display state, and when that state changes, the display must be notified of that so it can refresh itself.

Listing 2.1: Example in JAVA

```
interface FigureElement {
}
class Line implements FigureElement{
    private Point _p1, _p2;

    Line (Point p1, Point p2){
        _p1=p1,
        _p2=p2;
    }
    Point getP1() { return _p1; }
    Point getP2() { return _p2; }

    void setP1(Point p1) { _p1 = p1; }
    void setP2(Point p2) { _p2 = p2; }
    void moveBy (int dx, int dy) {
        _p1.moveBy(dx,dy);
        _p2.moveBy(dx,dy);
        Display.update;
    }
}

class Point implements FigureElement {
    private int _x = 0, _y = 0;

    Point (int x, int y) {_x=x; _y =y;}

    int getX() { return _x; }
    int getY() { return _y; }

    void setX(int x) {
        _x = x;
        Display.update ();
    }
    void setY(int y) {
        _y = y;
        Display.update ();
    }
    void moveBy (int dx, int dy) {
        _x += dx;
        _y += dy;
        Display.update ();
    }
}
}
```

Let's start by describing the basic notation that we have used in the simple pointcuts shown in figure 2.2.

- **Context** indicates the object type(s) where to look for the computational trace, i.e., it specifies the type of which objects could be interested by this flow. In the pointcut1 the context is Point, while in the other one the context

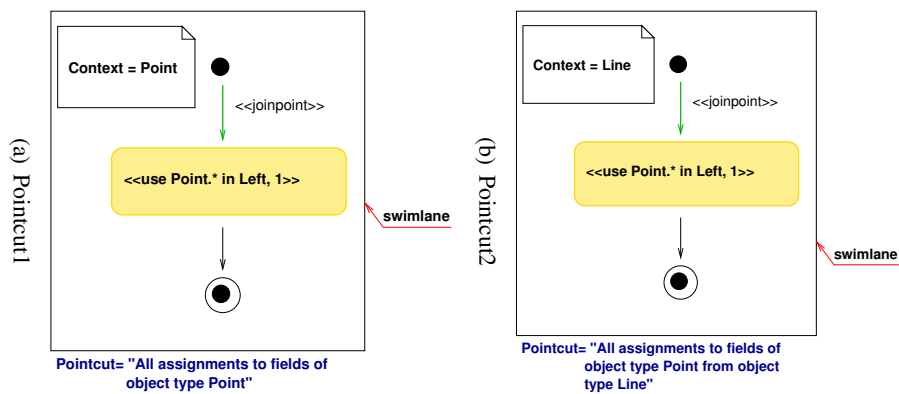


Figure 2.2: Example of Activity Pointcuts: a) All assignments to field of Point b) All assignments to fields of Point from Line.

is Line.

- The green arrow indicates the point that we want to locate.
- The action written inside the yellow rounded rectangle indicates that we are looking for a field of Point used as left side in an assignment.

The first pointcut represents the quest for of all assignments to fields of Point, while the second one represent the quest of all assignments to fields of Point from instances of Line. In the listings 2.2 and 2.3 are highlighted the join points that are interested by the first and second pointcut respectively. First pointcut locates four join points deriving by the 2 constructor executions, two join points deriving by the execution of the pt1.moveBy method, and finally four join points deriving by the execution of the ln1.moveBy. The last one locates only four join point deriving by the execution of the ln1.moveBy.

```
public class Main{
public static void main(String [] ar)
{
Point pt1= new Point (0,0); jp1,jp2
pt1 .moveBy(3,6); jp3,jp4
Point pt2 = new Point (4,4); jp5,jp6
Line ln1 = new Line(pt1, pt2);
ln1 .moveBy(3,6); jp7,jp8,jp9,jp10
}
}
```

Listing 2.2: join points match Pointcut1

```
public class Main
public static void main(String [] ar)
{
Point pt1= new Point (0,0);
pt1 .moveBy(3,6);
Point pt2 = new Point (4,4);
Line ln1 = new Line(pt1, pt2);
ln1 .moveBy(3,6); jp1,jp2,jp3,jp4
}
}
```

Listing 2.3: join points match Pointcut2

Now, we show a new pointcut example, see Fig 2.3, the pointcut of this example search: all points between two lines moving. In Listing 2.4 is shown an example of code, with this execution flow, our pointcut locates many join points, exactly 9 join

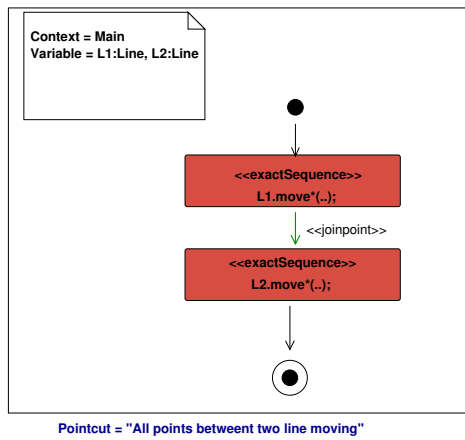


Figure 2.3: Another pointcut Example

Listing 2.4: Example of Code

```
public class Main{
public static void main (String [] args)
{
...
for (int i=0; i<10; i++){
... // lines insertion
}
int dx = 3;
int dy = 3;
// lines moving
for (Line ll : linee)
ll .moveBy(dx,dy);
}
}
```

points, one for every couple of lines moved. Since our pointcut is not syntax based the join points located by the pointcut are not visible into the code, to identify them it is necessary a loop unrolling,like shown in Listing 2.5.

Listing 2.5: Loop Unrolling

```
public class Main{
public static void main (String [] args)
{
Line [] lines =null;
...
for (int i=0; i<10; i++){
... // lines insertion
}
int dx = 3;
int dy = 3;
// lines moving
lines [0]. moveBy(dx,dy);
lines [1]. moveBy(dx,dy);
lines [2]. moveBy(dx,dy);
lines [3]. moveBy(dx,dy);
lines [4]. moveBy(dx,dy);
lines [5]. moveBy(dx,dy);
lines [6]. moveBy(dx,dy);
lines [7]. moveBy(dx,dy);
lines [8]. moveBy(dx,dy);
lines [9]. moveBy(dx,dy);
}
}
```

<<joinpoint>>  
<<joinpoint>>  
<<joinpoint>>  
<<joinpoint>>  
<<joinpoint>>  
<<joinpoint>>  
<<joinpoint>>  
<<joinpoint>>  
<<joinpoint>>  
<<joinpoint>>

Note, that this kind of situation is not recognizable with the actual AO tools. In the Appendix A, there is a detailed description of all the pointcut language elements for the activity diagrams.



## Work Plan

---

Summarizing, what we have said so far, it follows that, it is necessary next year to refine the tasks already developed.

- To verify that `@JAVA` and the set of APIs already implemented is suitable for our purposes, i.e., the identification of the join points.  
At present the `@JAVA` APIs are a minimal set, and if this set won't be able to satisfy our need, we should be to extend our library.
- To finish the definition of the UML-based pointcut language. We should face the other diagrams, both behavioral ones such as state and sequence diagrams and structural ones such as class diagrams.
- To define a search algorithm which identifies the pointcuts express by our UML-base pointcut language into the system code, and then it marks into the code the join points identified by the pointcuts, using our kind of annotation supplied by `@JAVA`.

Furthermore, next year we will have to implement the final mechanism, that is the weaving one. The weaving merges together all single parts that we have developed so far, i.e., the UML-base pointcut language, the pointcut identification inside the code and the identification of the join points expressed inside the UML-base pointcuts, and finally the insertion of the evolution code inside the program (this part will be implemented next year).

Our proposal for weaving consists of to build a program representation, starting from the bytecode of the system (i.e., the `JAVA` class files), using BCEL [4] and in particular using `InstructionList` of BCEL. Also, all pointcuts will be mapped into the correspondent BCEL `InstructionLists`. Finally, the search algorithm compares the pointcuts representations and the program representation to search matches. The result of this phase will be the identification of a set of points inside the system code. These points are the join points identified by the pointcuts. The identification of the join points will be realized marking the code locations using the `@JAVA` annotations.

Since our main scope is not the development of a new AO language, but it is to build an application that realizes run-time software evolution using our AO language, we will present some case studies using our application, and consequently

our AO language. This experimentation will serve to highlight the utility of our language for the evolution process. At this time, the case study we think to use is the urban traffic control system (UTCS), since it perfectly shows how requirements can dynamically change and how the design of the system should adapt to such changes.

---

## Structure of the Thesis

---

A possible structure of the thesis is the following:

- ❶ *Introduction*
- ❷ *Motivation*
  - ❶ Problem Definition
  - ❷ Role of the Design Information
  - ❸ Role of our Work in RAMSES Project
- ❸ *The UML-based Join Point Model*
  - ❶ The Pointcut Language
  - ❷ Examples of Use of Our Pointcut Language
  - ❸ Comparison with AspectJ
- ❹ *Meta-Data Support to the UML-based Join Point Model*
  - ❶ Meta-data and Annotation
  - ❷ @Java as Support for Join Points
  - ❸ Join Point Shadows
  - ❹ The Dynamic Residue of the Join Points Matching
- ❺ *The Weaving Mechanism*
- ❻ *Related Work*
- ❼ *Conclusion*
- ❽ *Appendix*



---

## The Pointcut Language

---

Pointcut language is the query language to select a subset of the join points defined by the join point model.

The inadequateness of the current join point models is a common feeling and many attempts have been done to overcome it. Mostly they focus on: i) defining a novel pointcut definition language; or ii) extending the existing definition language with new features.

In this work we have explained our idea to define a new pointcut language to improve the power of JPM, using design information.

Design information represent the result of the designing phase, and they describe in an abstract and intuitive way the structure, the behavior and interrelations of the code. UML diagrams slide the program behavior in separate views. Therefore they represent both some entangled code in the program and a way to retrieve it.

The interrelations among the program components are immediately available through design information. Class and Object diagrams describe clientship and inheritance among the program elements. Activity and sequence diagrams, use case and statecharts describe how and when the clientship takes place. In practical, from UML diagrams we have at compile-time a global view of the static structure and of the dynamic behavior of the program.

Pointcuts can be expressed as *sample* UML diagrams, we look for a match between the sample diagram and the the program code to identify the related join points.

The following table A.1 summarize the main elements of our pointcut language, in particular these elements are used by activity diagram pointcut.

## Appendix A: The Pointcut Language




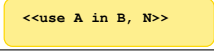
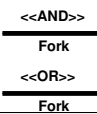

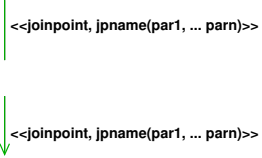

Pointcut Language: elements description	
Element	Description
context	indicates the type(s) of the objects where to try the computational flow contained in the swimlane.
Variable	indicates names and types of the fields used into the pointcut computational flow.
Method	indicates nouns and parameters of the method used into the pointcut computational flow.
	indicates that we want to find from N to M generic JQVQ instructions into the pointcut computational flow. If N is omitted we consider from 1 to M.
	indicates that in the pointcut computation flow we search exactly this type of object transaction.
	contains a block of JQVQ statements which must be exactly match a sequence of JQVQ instructions into the code of the classes indicate in Context. The names used in this sequence must be unified with the real names used into the class code. The names not useful to the pointcut definition are not inserted into Variable, but they are indicated directly inside the instruction as (i) with i=1 . .n. These names must be however unified with the real names.
	indicates the quest of a JQVQ instruction of type B containing the names indicate in A, inside the next N instructions. B can be: BooleanCondition, Left, Right, Index, and Statement.
	indicates the search of alternative (OR) or more (AND) flows. It is a pointcut language element and it doesn't indicate a parallel execution.
	indicates the end of the search of alternative (OR) or more (AND) flows, after a Fork. It is a pointcut language element, and it doesn't indicate the end of a parallel execution.
	indicates the point that we are lookinf for inside the specified flow. jpname is the name of the join point, it is optional. par1, . . ,parn are the eventual parameters. Parameters are necessary if the matching process between pointcuts and joinpoints might not be completely statically resolvable. If the «joinpoint» is located on an transaction between two objects, it must be put either near the caller or near the callee, since it is necessary to locate the exact point tried in the flow. joinPoint put near the caller means that we are trying the call of the method that has the following computational flow, «joinpoint» near the callee means that we want locate one point in the execution flow of a callee's method. «joinpoint» can also have one o more parameters, they will be indicate with functional notation on their names.
	indicates the quest of the iteration of the flow enclosed into the rectangle.

Table A.1: Pointcut Language: Elements Description

## Bibliography

---

- [1] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. RAMSES: a Reflective Middleware for Software Evolution. In *Proceedings of the 1st ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'04)*, in 18th European Conference on Object-Oriented Programming (ECOOP'04), pages 21–26, Oslo, Norway, on 15th June 2004.
- [2] Walter Cazzola, Sonia Pini, and Massimo Ancona. The Role of Design Information in Software Evolution. In *Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'05)*, in 19th European Conference on Object-Oriented Programming (ECOOP'04), Glasgow, Scotland, on 25th July 2005.
- [3] Shigeru Chiba. Load-Time Structural Reflection in JOVCL. In Elisa Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, LNCS 1850, pages 313–336, Cannes, France, June 2000. Springer-Verlag.
- [4] Markus Dahm. Byte code engineering. In *Java-Informationen-Tage*, pages 267–277, 1999.
- [5] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. FOW m 01:1 1.Ex.
- [6] Robert B. France and James M. Bieman. Multi-view software evolution: A UML-based framework for evolving object-oriented software. In *ICSM*, pages 386–, 2001.
- [7] Stuart Kent. Model driven engineering. In *IFM*, pages 286–298, 2002.
- [8] G. Kiczales. Aspect-oriented programming - the fun has just begun. In *Workshop on New Visions for Software Design and Productivity: Research and Applications*, December 2001.
- [9] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeff Palm, and Bill Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 327–353, Budapest, Hungary, June 2001. ACM Press.

## Bibliography

---

- [10] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. In *Proceedings of the European Interactive Workshop on Aspect in Software (EIWAS'04)*, Berlin, Germany, September 2004.
- [11] K. Kowalczykiewicz and D. Weiss. Traceability: Taming uncontrolled change in software development, 2002.
- [12] M. M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, December 1996.
- [13] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation Semantics of Aspect-Oriented Programs. In Gary T. Leavens and Ron Cytron, editors, *Proceedings of the Workshop on Foundations of Aspect-Oriented Languages (FOAL'02)*, pages 17–25. Iowa State University, April 2002.
- [14] Sonia Pini. Thesis proposal: Uml based aosd and join point model for software evolution, 2004.
- [15] James E. Rumbaugh. Modeling through the years. *JOOP*, 10(4):16–19, 1997.