

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

Thesis Proposal

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: Massimo Ancona,
Department of Computer and Information Sciences
University of Genoa.

Walter Cazzola,
Department of Informatics and Communication
University of Milan.

December 2004

Contents

1	Introduction	1
2	Thesis Outline	3
3	State of the Art	9
3.1	Software Evolution	9
3.2	Aspect Oriented Software Developmet	11
3.2.1	Fred	12
3.2.2	DemeterJ and DJ	12
3.2.3	AspectJ	12
3.2.4	Hyper/J.	12
3.2.5	AspectWerkz	13
3.3	Current Join Point Model	13
4	Preliminary Results	15
5	Future Work	23
5.1	Join Point	23
5.2	Pointcut Mechanism	26
5.3	Weaving Mechanism.	26
	Bibliography	29

Introduction

Nowadays a topical issue in the software engineering research area is the software evolution, in particular to render a system adaptable to environmental changes by adding new and/or modifying existing functionality. A particularly difficult test case for the software evolution is the evolution of nonstopping systems, we called such kind of evolution *dynamic software evolution*. For this type of application, there are several critical aspects linked with the runtime evolution, in first way it is hard to individuate a context for reasoning about, specifying and implementing changes, moreover it is difficult to select when and how to evolve the system without stopping the service supplying or without rendering the system execution inconsistent. Therefore, these applications must be able to dynamically plan and adapt themselves to sudden and unexpected changes to their environment because they are aware of their behavior and structure. The dynamic software evolution requires a mechanism that permits of applying the changes on the running system, which should be able of picking out the application code interested by the changes and then injecting the new code in the located code (code instrumentation).

Several work [] has been done in the last few years to render a system self aware, in our opinion one of the most substantiated approach is based on meta-data management. In particular the decision making phase could be driven by the system design information. Each system is (or should be) designed by using UML (or similar techniques). The design provides all the data necessary to the system to plan its evolution and a good evolutionary plan can be directly designed on these diagrams producing a new set of design data.

Unfortunately, UML diagrams as well as any kind of design information are typically not available at run-time. Moreover does not exist a direct connection between the design data and the application code, and finally the UML diagrams provide a mechanism clear, intuitive and powerful for understanding the design of the system for a human but this is no so simple for a tool. Hence, the design data could represent the panacea for the software evolution but at the moment is not so simple to adopt and manage them as meta-data.

The main aim of this thesis consists of modelling a mechanism for code instrumentation based on design information. This mechanism will be used in the context of dynamic software evolution. Nowadays, a promising approach to code instru-

mentation is represented by the aspect-oriented development techniques (AOP/AOSD). Aspect-oriented programming is a new designing and programming technique, it allows to express different concerns of a software system in a natural and separate way, it permits to picking out and select the various part of the application code and then automatically insert the separately developed concerns in the right place. Most of the aspect-oriented approaches exploit behavioral and syntactic data to drive the code instrumentation neglecting the use of design information for that. We are extending current AOSD models to overcome these problems and to provide the right mechanisms to support dynamic software evolution based on design information. In our opinion it is possible to annotate, the application code by using the description provided by the UML diagrams – and therefore to pick out the join points interested by the evolution –. The evolved UML diagrams of the application should drive the selection of the previously annotated join points involved by the evolution whereas the difference between the original and the modified design information will provide the code that must be injected, i.e., the code evolution itself.

Thesis Outline

The topic of this thesis straddles over two research fields, the software evolution and the software development techniques research area.

Evolution is a term related to many different domains. For example, natural species, societies, ideas, concepts and so on, are said to evolve in time. In all these contexts the term refers to continuous and progressive changes. Usually, the evolution serves to adapt the entity that evolves to the evolution of its environment. In particular, we are interested in software evolution [2], which is a phase in the software lifecycle where the software product is modified to correct faults, to improve performance or to adapt the product to a modified environment. In particular we are interested in runtime software evolution, that is, to evolve an application that must be continuously available. In the case of non-stoppable applications, to shut down and restart the system for an update is a very critical operation. Therefore to support dynamic adaptation is a key issue for these systems.

The principal concerns of runtime software evolution are:

- to identify what must be changed;
- to inject the necessary changes in the application code; and
- to control that the changes preserve the system integrity.

Several approaches to runtime software evolution have been proposed in the literature [23, 18, 7].

The software evolution process is composed of the following steps: to detect the event that generates the evolution (*why?*), to plan the appropriate action to face the occurred event and to generate the appropriate code for the evolution (*what?*), to determine the right moment to apply the changes to the system without undermine (threaten) the consistency and the integrity of the system (*when?*), and to identify what parts of code are interested by the evolution (*where?*) and then inject the code generated for the evolution in these code locations (*how?*).

The first steps can be resumed in a single phase, that we call *planning phase*, while the last step that concerns the effectively carry out the planned evolution is called *deployment phase*.

Nowadays the main approaches to the planning phase are based on the idea that the changes can be anticipated at design time, and then can be built in by some

form of parametrization, or on the idea that the users of the system request for bug fixing and new functionality that in somehow can be foreseen.

The main techniques adopted in the planning and deployment phases are: code instrumentation, code injection, declarative programming and alternative code wrapping.

Often, software evolution of a generic systems is carried out by stopping the system and manually, or with the aid of specific tools, changing the system behavior according to the required evolution. Nowadays, a more dynamic approach is required . A feasible approach consists of encapsulating the system prone to be adapted in a monitoring system that waits for an event. When the event occurs, the monitoring system plans a countermove that will imply the automatic and dynamic evolution of the monitored system. The monitoring system also takes care to grant the safety and stability of the monitored system against its evolution. Some examples of this approach are [8] and [13].

All work made in the software evolution area is very useful but there is still an open issue (at least): all that work is based on designing all the feasible evolutions together with the system itself , therefore really unpredictable events cannot be managed. Recently, the researches are focusing their efforts on facing unpredictable events, many examples of these new direction can be seen in the proceeding of the conferences USE2002 and USE2003.

To manage unpredictable events, the system must have a complete representation of itself available at runtime, and it must be able to elaborate this information in the right way.

These representations could be obtained by the design information. Having several levels of design knowledge helps runtime software evolution, because information design is concerned with making complex information easier to understand and to use. Information design helps people and organizations to achieve understanding through the creation of relevant, and clear information. Design phase is the intentional process in which data-elements related to a specific domain are transformed to obtain an understandable representation of that domain. In particular, UML (Unified Modeling Language) [5] is the main methodology for software development, which describe the system's behavior, architecture and components. The main problem of using the design information, as system representation, consist of developing a mechanism to handle and use this information in the planning and execution phase of the software evolution.

The planning of the evolution is 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.

The work of this thesis concerns the deployment phase of the software evolution, and consists in supporting the evolution with code instrumentation, neglecting how the planning phase is developed.

In particular we want to carry out system evolution by using the design information, based on the UML diagrams, of the system itself, independently of how the design knowledge is used in the planning phase.

The main problem is to develop a mechanism that picks out the part of code, both source and target, involved by the evolution and that permits the realization of the evolution injecting the appropriate code in the right code location (instrumentation)

Aspect Oriented Programming [20] provides mechanisms that allow of modifying the behavior and the structure of an application, also of a nonstopping application. The AOP mechanisms address functionality that crosscut the whole implementation of the application. Evolution is a typical functionality that crosscut the code of many objects in the system.

Aspect-Oriented (AO) paradigm is a new trend to assist the development (both design and implementation) of highly complex software systems, proposing specific language level constructs to develop a system as a set of separate concerns and then to compose these concerns.

Currently, when we think to the better way of programming a system the first words that jump in mind are 'separation of concerns'. Aspect Oriented Programming (AOP) allows to express the different concerns (aspects) that characterize a software system in a separate and natural form, and then to automatically combine these separate descriptions into a final executable [20]. These aspects crosscut the software, they are not captured as a single design entity that results from decomposition, but they can be looked at as some of the properties of the software.

It is possible that an object-oriented software system be developed by using aspect-oriented programming techniques that mainly consist of classes and aspects. Classes implement the primary functionality of an application, whereas, aspects capture non-functional concerns. Finally aspect code is injected into the code of the classes, in the AO parlance this process is called weaving.

Aspect-oriented software development provides means for identification, modularization and composition of aspects by using both source and target code instrumentation.

An aspect in the sense of AOP is a mean of specifying policies or strategies that are orthogonal to the software primary functionality. In our idea, evolution can be considered as a non functional concern and therefore it can be specified by using AOP.

There are several tools which support AOP, such as AspectJ [16], Hyper/J [27], AspectWerkz [33], Steamloom [3], and so on, but it is also possible to incorporate the AO concepts into an object-oriented system using the reflective methodology. The AOSD methodology identifying well defined points in a program's execution (called join points) and to pick out (at run-time) a set of these well defined points based on defined criteria (called pointcuts), all this in AO context is called *Join Point Model (JPM)*.

Current AO tools and their JPMs are mainly oriented to the syntax, that is, they permit to refer only a few code fragments, such as method invocation, execution of methods, field access, and so on, though their syntax and the name of the involved entities.

Although AOP seems to become useful in solving run-time software evolu-

tion, the situation is not perfect because current JPMs are too primitive and not expressive enough. In an ideal JPM any kind of pattern that can be identified by a programmer should be definable in AOSD implementation. Each of current AOP languages is based on a fixed set of JPMs. Many different JPMs have been proposed, and they are still evolving to modularize various crosscutting concerns.

Often, in current techniques for identifying join points, there is a high coupling between the aspects and the join point in the program at which these apply. Consider for example the crosscut language of AspectJ [21], which allows the use of patterns (and wild cards) in identifying join points. When we want to identify for example every method changing the internals of a class, also known as the setters of a class, we might use a pattern like: `execution(* set*(*)`). With this pattern we identify every method of a particular class with a name starting with 'set'. This works in most of the cases, but what if we have a method settings, which returns the settings of a class and does obviously not change internals. Then this join point would also be identified. Or what if we introduce a method which does change the internals of a class, but does not start with set.

The previous critiques and many others that we will expose in the next derive both by our preliminary analysis [10] of feasibility and by the work of other researchers [31, 30, 1, 17].

The previous critiques evidence, that the current approaches to JPM are too syntax oriented and thus not expressive enough to identify several problematics. Now, the main question is: *Exists a mechanism that allow to describe all the possible situations?* In our opinion the UML [5] methodology with its variety of diagrams fits the problem. The UML methodology forces the developer to describe each software system by using a set of diagrams. These diagrams take care of representing every aspect of the system, from its structure (e.g., class and object diagrams) up to its behavior (e.g., statecharts, sequence and activity diagrams). Moreover, these diagrams describe the behavior of the system independently of the syntax adopted in the code but rather in terms of the trace of its execution. Therefore, these diagrams should provide all the necessary data for the evolutionary purpose, they describe such data as a whole and abstracting from any syntactic description and, because of their pictorial nature, they are also characterized by a higher degree of intuitiveness. Then, in our opinion the UML diagrams could provide the appropriate mechanism to implement a new JPM to support the deployment phase of run-time software evolution. In the last few years many researchers like Sillito and Jacobson have begun to use the UML diagrams, in particular the use case, for the identification of join points and the definition of the pointcuts. In particular, according to Ivar Jacobson [19] Aspect Oriented Programming (AOP) is the *missing link* to allow you slice a system, use case by use case, over *all* life-cycle models.

To use design information such as behavior, that is easily identifiable from the UML diagrams, is a hard job with the current JPMs. Whereas the current weaving mechanisms are very useful, also in our context.

Resuming, our idea is to manage the run-time software evolution by using for the planning and the deployment phases the design information. Therefore, we have

to develop a JPM based on the UML, to support the run-time software evolution.

State of the Art

In this chapter, we analyze some research area related to our thesis, in particular Software Evolution (SE), and Aspect-Oriented Software Development (AOSD).

3.1 Software Evolution

In [2], *software evolution* is defined as a kind of software maintenance that takes place only when the initial development was successful. The goal consists of adapting the application to the ever-changing, and often in an unexpected way, user requirements and operating environment.

Software evolution, as well as software maintenance, is characterized by its huge cost and slow speed of implementation. Often, software evolution implies a re-design of the whole system, the development of new features and their integration in the existing and/or running systems (this last step often implies a complete rebuilding of the system).

Besides, software systems are often asked for promptly evolving to face critical situations such as to repair security bugs, to avoid the failure of critical devices and to patch the logic of a critical system.

It is fairly evident the necessity of improving the software adaptability and its promptness without impacting on the activity of the system itself. This statement brings forth the need for a system to manage itself to some extent, to dynamically inspect component interfaces, to augment its application-specific functionality with additional properties, and so on. Nonstopping applications with a long life span are typical applications that have to be able to dynamically adapt themselves to sudden and unexpected changes to their environment. Therefore, the support for run-time adaptive software evolution is a key aspect of these systems. Software evolution of a generic system is usually carried out by stopping the system and manually, or with the aid of specific tools, changing the system behavior according to the required evolution. A more dynamic approach consists of encapsulating the system prone to be adapted in a monitoring system that wait for an event. When the event occurs, it plans a countermove that will imply the automatic and dynamic evolution of the monitored system. The monitoring system also takes care to grant the safety and stability of the monitored system against its evolution. Some examples of this

approach are [8] and [13], whereas some example of a more standard approach are [23, 15] and [18]

Cazzola et al. [8] describe how design information can be used to evolve a software system and validate the consistency of such an evolution. This work is based on a reflective architecture which provides objects with the ability of dynamically changing their behavior by using their design information. The meta-level supervises the evolution of the system to be adapted that runs at the base-level system of the reflective architecture. The evolution takes places in two steps: first a meta-object, called *evolutionary meta-object*, plans a possible evolution against the detected event then another meta-object, called *consistency meta-object* validates the feasibility of the proposed plan before really evolving the system.

Dowling in [13] introduce the architecture meta-model, that realizes a dynamic software architecture. The basic idea is to reifies the software architecture as a typed, directed configuration graph, where interfaces are the vertices, components the type labels and connectors are directed edges.

Kramer and Magee, in [23] present a structural-based approach to run-time adaptation of the configuration of a distributed system. When a modification is required, nodes (that is, the components of the distributed system) that are directly affected by the change are processed, and nodes adjacent to them enter into a *quiescent* state. When a node is in the quiescent state, it can not initiate communication with peers, this grants that nodes affected by a change will not receive request for service during the adaptation. Changes, specified in a declarative language, are induced to the running system by the reconfiguration manager. The reconfiguration manager is responsible for making decisions regarding the change application policy and its scope. The reconfiguration managers must do so based on a limited model of the application consisting of the system's structural configuration and whether its nodes are in quiescent states or not . As a result, designers must consider the reconfiguration managers role in the run-time adaptation, and structure the system to attain desired effects.

Gorlick et al., in [15] present a data flow based approach to run-time adaptation-called Weaves. A weave is an arbitrary network of tool fragments connected together by transport services. A tool fragment is a small software component, on the order of a procedure, that performs a single, well-defined function and may retain state. Each tool fragment executes in its own thread of control. Transport services buffer and synchronize data communication between tool fragments. The Weave run-time system guarantees the atomicity of data transfer between tool fragments and queues; if any problem occurs during communication, the tool fragment initiating the communication is notified and may retry the operation at its discretion. This enables the run-time reconfiguration of a weave without disturbing the flow of objects. Designers use an interactive, graphical editor to visualize and directly reconfigure a weave during run-time. Weaves does not currently provide a mechanism to check the consistency of run-time changes and no explicit support is provided for representing change policies. The designer is solely responsible for change management.

Peterson et al., in [18] present an approach to module-level run-time adaptation based on Haskell, a higher-order, typed programming language. Their technique requires to the programmers to anticipate portions of the program likely to change at run-time, and to structure the program around functions that encapsulate such changes. Developers encode decisions regarding the change application policies and change scope in the application source code. This technique permits fine-grained control over run-time change since designers can implement change policies tailored to the application. However, because change policies are not isolated in the application source code, they can be difficult to alter independent of application behavior. As a result, managing change in large systems becomes complex. It is unrealistic to assume that the programmer is able to provide a countermove to any possible event during the development. Therefore these approaches, based on the prediction of the events, cannot face the dynamic evolution against (really) unexpected events, they lack in flexibility. Nowadays, software evolution, especially of large and critique systems, is more and more driven by unpredictable changes, so it is necessary to have a mechanism for software evolution that renders a system able to respond to unpredictable events and therefore that overcomes the drawbacks of the existing approaches.

3.2 Aspect Oriented Software Development

The term *aspect-oriented programming* is introduced by Gregor Kiczales et al. in [20]. The original goal of the AOP is to make it possible for programs to clearly capture all of the important aspects of a system's behavior, including not only its functionality. In other word aspect oriented programming allows to express the various concerns of a system in a natural and separate form and then automatically re-combine the separate parts into an executable. The main characteristic of aspects is that they are not captured by a single design unit, that is aspects crosscut the system.

AOP techniques are based on three main concepts [20]:

- *Join Points*, which are a well defined points in the execution of the program,
- *Pointcuts*, which are a way of referring to a collection of join points, and
- *Aspects*, which are used to add an extra behavior at the selected join points.

Join point represent the key concept in Aspect Orientation, to specify a set of join points is a major task for aspect oriented designers, and to provide a suitable representation for join point is primary task for an aspect oriented language.

AO tool uses a weaving mechanism for merging the extra behavior at the join points. There are two ways to apply aspect into base software: static or dynamic [4, 29].

The AOP infrastructure, that is join points, can be implemented by instrumenting all possible join points (that is code locations) with hooks, and at the compile

time hooks are called any time that may be a join point regardless of whether there is aspect code associated with or no. The hooks can be inserted and deleted also at the run-time.

Static weaving means to modify the source code of a class by inserting aspects at the join points, that is the aspect code is injected into source code. The main drawback of the static weaving is that it makes difficult to pick out the aspects in the woven code.

Dynamic Weaving allows the use of AOP in response to adapt application to environmental change. With this type of weaving aspects can be woven and unwoven at the run-time.

In the last months we have analyzed many AO tools.

3.2.1 Fred

Fred [26], developed by Doug Orleans, is a language that integrates aspects oriented programming and predicate dispatching and it is implemented in [14]. The main concepts in Fred are message, branch, and decision point. A message maybe thought of as a function. Then, a branch is analogous to a function call. The decision point is where a decision is made about which branch should be invoked.

3.2.2 DemeterJ and DJ

The main concepts of Demeter are class graph, strategy, visitor, and advice. The class graph is the schema of the data structures used within a program. A strategy is a direction on how to traverse the object graph. The visitor in Demeter is the one that traverse the traversal graph. It has advices that are invoked for particular types of node in the traversal graph. The previous are the Demeter concepts, the three main components class graph, strategy, and visitor are specified in different ways in DemeterJ and DJ.

3.2.3 AspectJ

AspectJ [21] extends the Java language to allow programmers to express crosscutting concerns. The main features of AspectJ are join point, pointcut, and advice. It support two kind of crosscutting implementation. The *dynamic crosscutting* makes it possible to define additional implementation to run at certain well defined points in the execution of the program, teh *static crosscutting* makes it possible to define new operation on existing types.

3.2.4 Hyper/J

Hyper/J [27] supports a new approach to constructing, integrating and evolving software, called multi-dimensional separation of concerns. Developers can decompose and organize code and other artifacts accord to multiple, arbitrary criteria (concerns) simultaneously and synthesize or integrate the pieces into large-scale

components and system. It operates on standard Java class files -without need or source - and produces new class files to be used for execution.

3.2.5 AspectWerkz

AspectWerkz [33] is a dynamic, lightweight aspect oriented programming for java. It offers both off-line and on-line aspect weaving mechanisms for real-world application platforms. The off-line mode allows aspects to be integrated ('woven') into application code before the application is deployed. The on-line mode weaves the aspects at class load time in a transparent way. Both implement 'dynamic AOP' concepts that allow developers and system operators to add, remove, and modify aspects during the execution of an application.

3.3 Current Join Point Model

A join point model is used to describe at which points an aspect crosscuts program. An important issue is how these points can be captured using the crosscutting language without introducing highly coupling between the aspect and the program. Now, we consider some approaches to capturing the join points which are used in most current AOP languages. The first approach we consider is pattern matching. A pattern would describe what is common to all the join points that should match the crosscut. For example, this approach is found in AspectJ [21].

Gybels et al., in [17] present a logic-based crosscut language. Their crosscutting language is in essence a logic programming language, based on Prolog, in which predicates are provided which allow the writing of crosscut expression. The crosscut model of this language is a dynamic one, based on the AspectJ crosscut model, in which the join points are related to key events in the execution of an object-oriented program. There are five types of join points: message receptions by an object, message sends by an object, the accessing and updating of an object's state and the execution of code block. For each type of join point there are different types of properties associated with it.

Sillito in [30] provide a new aspect language, called AspectU, which supports modularization of crosscutting concerns in the use-case model. AspectU provides a JPM based on elements which are use case, steps and extension. An AspectU aspect capture changes to a system's dynamic behavior as expressed in its use cases. These behavioral changes are expressed as steps and extensions added to or replacing elements of the existing behavior.

Douence et al., in [12] propose an approach to AOP which is based on the observation of execution events (Event-Based AOP), which allows to systematically treat relationship between pointcut, operator-based composition of aspects, the definition of aspects which are applied to other aspects and dynamic instantiation of aspects. Aspects are expressed by means of events emitted during execution of the so-called base-program. Aspect weaving is realized using an execution monitor, which enables event sequences to be detected.

Preliminary Results

In the last months, we have explored the aspect-oriented approach as a tool for supporting the software evolution. The aim of this analysis is to highlight the potentiality and the limits of the aspect-oriented development for software evolution. From our analysis follows that in general (and in particular for `AspectJ`) the approach to join points, pointcuts and advices definition are not enough intuitive, abstract and expressive to support all the requirements for carrying out the software evolution. We have also examined how a mechanism for specifying pointcuts and advices based on design information, in particular on the use of UML diagrams, can better support the software evolution through aspect oriented programming. Our analysis and proposal are presented through an example.

In particular we have analyzed the evolution of nonstopping applications, these applications must be able to dynamically adapt themselves to sudden and unexpected changes to their environment. In general an approach to the run-time software evolution requires a mechanism that permits of concreting the evolution on the running system. In particular this mechanism should be able to:

- picking the code interested by the evolution out of the whole system code,
- carrying out the patches required by the planned evolution on the located code.

Both these steps must occur without stopping the system.

From our analysis aspect-oriented programming (AOP) [20] provide mechanism (*join points*, *pointcut* and *aspect weaving*) that allow of modifying the behavior and the structure of an application, also of a nonstopping application (*dynamic weaving* [29]). The AOP mechanisms better address functionality that crosscut the whole implementation of the application. Evolution is a typical functionality that crosscut the code of many objects in the system. Moreover, the AOP mechanisms seem suitable to deal with the detection of the code to evolve and with the instrumentation of such code. In our view, AOP will play the role of the low-level tool used to render effective the planned evolution.

The main goal of AOP consists of providing methods for the identification, modularization, representation, and composition of crosscutting concerns. The captured aspects (both functional and non-functional) are separated in well-defined modules

that can be successively composed in the original or in a new application.

The basic mechanism to separate the crosscutting concerns in aspects and for weaving them together again are: *join points*, *pointcut*, and *advice*. *Join points* represent well-defined points in the execution of a program, such as method calls, object field accesses and so on. *Pointcut* is a construct that picks out a set of join points based on given criteria, such as method names and so on. Pointcuts serve to define which advice has to be applied. An advice defines additional code to be executed at join points picked out by the pointcuts. Finally, an aspect represents a crosscutting concern and is composed of pointcuts definition and advices to be weaved at the corresponding join points. The frame that renders possible the proper execution of the assembled program is called *join point model*.

AspectJ [21, 24] has been the pioneer of the aspect-oriented languages and it is still one of the most relevant frameworks supporting the AOP methodology. In AspectJ an aspect looks like:

```
aspect <aspect name> { /* pointcut definitions */
    pointcut <pointcut designator> :
        <join points description>; /* advice definitions */
        <advice type> : <pointcut designator> { <advice body> } }
```

As defined in [21], join points are basically described by composing explicit method signatures with predicates on the execution flow, i.e., the given kind of join points. An example of join point description is: **call**(* *.print(. .)); it describes all the join points at the invocation of the methods named **print**, does not matter which object receives the message, its return type and how many arguments it needs. Whereas, the advice type suggests the point with respect to the join point where the advice body will be weaved, some examples are **before**(), **after**() and **around**() whose behavior comes after their name.

However, in this last few years many other frameworks have been developed, some of them are: AspectWerkz [33], Josh [11] and JMangler [22]. Notwithstanding that this analysis takes AspectJ as referring AOP framework, many of the considerations we do can be also applied to other frameworks.

From AOP characteristics, it is fairly evident that AOP has the potential for providing the necessary tools for instrumenting the code of a nonstopping system, especially when aspects can be plugged and unplugged at run-time. Pointcuts should be used to pick out a region of the code involved by the evolution, whereas the advices should be used to define how the code — identified by the pointcut — should evolve. Weaving such an aspect on the running system should either inject new code or manipulate the existing code, allowing the system dynamic evolution.

Unfortunately, to define pointcuts that point out the code interested by the evolution is a hard task because such modifications can be scattered and spread all around the code and not confined to a well-defined area that can be taken back to a method call. Moreover the changes could entail only part of a statement, e.g., the exit condition of a loop or an expression, and not the entire statement. To highlight the entity of the problem we can consider the implementation of a simple bounded

buffer with `get()` and `put()` operations with the usual semantics.

```
public class BoundedBuffer {
    private int first = 0;
    private final int MAX = 20;
    private int buffer[] = new int[MAX];

    public void put(int n) throws FullBufferException {
        if (first < MAX) buffer[first++] = n;
        else throw new FullBufferException();
    }
    public int get() throws EmptyBufferException {
        if (first > 0) return buffer[--first];
        else throw new EmptyBufferException();
    }
}
```

Listing 4.1: Bounded Buffer in JAVA

The JAVA code, reported in listing 4.1, despite of its simplicity, is enough complex to be used to explain the hardness of determining all the code involved by an attempt of evolution. At this point, we consider a change to the system requirements that forces a change to the applicability rule of the method `get()`. After that, the method `get()` can be invoked only after one or more invocations of the method `put()` and not immediately after another invocation of the method `get()`. In this case, the evolution simply involves the applicability rules of a method but such a simple change, as we show, implies many changes all around the application code, also in some cases unexpected changes. At a first glance could seem that the changes are confined in the body of the method `get()` since it is the only method whose requirements change, of course, this impression is not true.

```
public class BoundedBuffer {
    private int first = 0;
    private boolean lastIsPut = false;
    private final int MAX = 20;
    private int buffer[] = new int[MAX];

    public void put(int n) throws FullBufferException {
        if (first < MAX) {
            buffer[first++] = n;
            lastIsPut = true;
        } else throw new FullBufferException();
    }
    public int get() throws NotEnoughPutsException {
        if (lastIsPut) {
            buffer[first++] = n;
            lastIsPut = false;
        } else throw new NotEnoughPutsException();
    }
}
```

```
}
```

Listing 4.2: Evolved Bounded Buffer

Rather, the whole class code is affected by the required evolution:

- a new boolean attribute (`lastIsPut`) has been introduced in the class, if it holds true the method `put()` is the last invoked, it will hold false otherwise;
- each time the method `put()` is invoked the new attribute `lastIsPut` must be set to **true**;
- the precondition to the call of the method `get()` changes to satisfy the new constraint¹, moreover, the flag `lastIsPut` must be set to **false**.

The listing 4.2 shows the evolved bounded buffer class, to give more emphasis to the changes, they are written in gray. Therefore, it is fairly evident that our simple test has spawned several noisy and punctual changes that are difficult to deal with (both for maintenance, flexibility and clarity).

As said before, the AOP technology could be the right approach to deal with the evolution concern but scenarios similar to the one described by our example are difficult to administrate with the current aspect-oriented frameworks. The main issues that obstacle the use of the current AOP approaches are: the *granularity* of the requested manipulation and the *locality* of the code to manipulate. The requested granularity for the pointcut is too fine, traditional join point models refer to method invocation in several way whereas we want to be able to manipulate a single statements or a group of statements in somehow related. This means that we can manipulate the method execution at its beginning and at its ending but we can not alter its computational flow requiring the execution of another statement between two statements of its body.

In a limited way, we could work around the problem by extruding each group of statements interested by the evolution to the body of a method and replacing their occurrence with an invocation of such a method. Moreover we should (separately) define a specific pointcut (and related advices of course) for each join point that as to be manipulated. This solution is not always practicable because (neglecting the fact that it forces a manual refactoring of the original system and it is a little bit tricky):

- it is too fragmentary and therefore error prone when the spectrum of the evolution grows in size (how we could be sure that everything has been taken in consideration?);
- it is tailored on a specific case and does not permit to describe general pointcuts, for example, it can not be associated to a trace of the program execution;

¹Note that, if you can invoke `get()` only after a `put()` has occurred, it is impossible to invoke the method `get()` on an empty buffer.

- it is not applicable when the code interested by the evolution can not be promoted to a method, e.g., two interleaved statements or just part of a structured statement or expression;
- it strongly depends on the syntax of the program rather than on its semantics, that means that we can not use a single pointcut to describe the join points associated to two methods with the same behavior but with a different name;
- the removal of a statement is not so immediate and simple.

These problems are due to the poor expressiveness of the pointcut definition languages and of the related join point models provided by most of the actual AOP frameworks. Nowadays AOP languages provide very basic pointcut definition languages that heavily rely on the structure and syntax of the software application neglecting its semantics. The developer has to identify and to specify in the correct way the pointcut by using, what we call the *linguistic pattern matching* mechanism; it permits of locating where an advice should be applied by describing the join points as a mix of references to linguistic constructs (e.g., method calls) and of generic references to their position, e.g., before or after it occurs. Therefore, it is difficult to define generic, reusable and comprehensible pointcuts that are not tailored on a specific application. Moreover, current join point model is too abstract. Join points are associated to a method call whereas a finer model should be adopted to permit of altering each single statement.

Similar issues have been raised from several researchers that are providing their own pointcut language or join point model, some examples are [32, 17]. More or less each proposal addresses part of the problem but often the solution is not so intuitive as it is necessary to be usable.

In all their works they propose to use a more expressive pointcut definition language mainly based on logic deduction and pattern matching. Notwithstanding the powerfulness of their proposals, they do not provide a straightforward and easy approach to pointcut definition. Moreover the degree of abstraction and the relative granularity seems inadequate for the software evolution.

The question is: *does it exist a tool for describing the pointcuts that is independent of the program syntax, intuitive and easy to use?* In our opinion, the UML [5] methodology with its variety of diagrams fits the problem. The UML methodology forces the developer to describe each software system by using a set of diagrams. These diagrams take care of representing every aspect of the system, from its structure (e.g., class and object diagrams) up to its behavior (e.g., statecharts, sequence and activity diagrams). Moreover, these diagrams describe the behavior of the system independently of the syntax adopted in the code but rather in terms of the trace of its execution. Therefore, these diagrams should provide all the necessary data for the evolutionary purpose, they describe such data as a whole and abstracting from any syntactic description and, because of their pictorial nature, they are also characterized by an higher degree of intuitiveness.

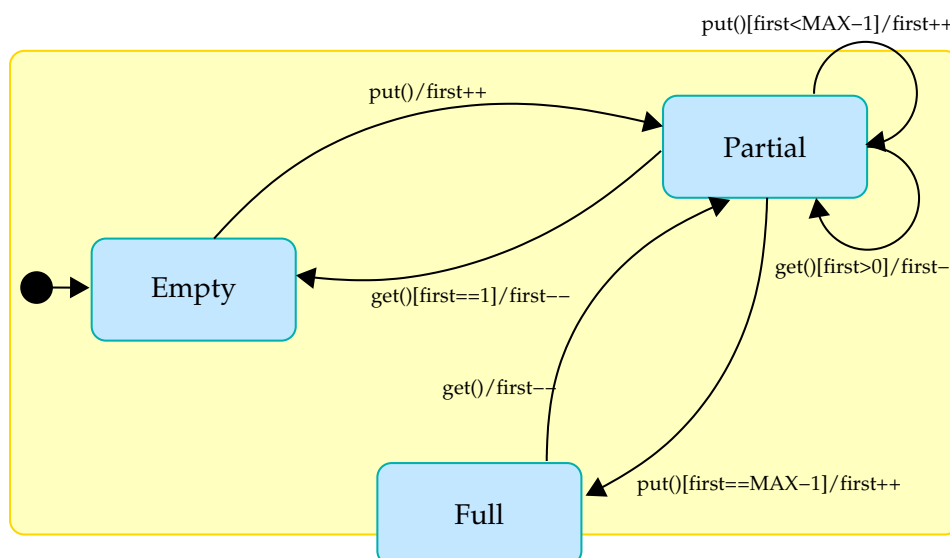


Figure 4.1: The Statechart of the Bounded Buffer

Coming back to the *bounded buffer* example, the approach provided by ASPECTJ, and by other AOP frameworks, does not provide the necessary granularity and degree of abstraction to permit its evolution in a straightforward manner. In figure 4.1, it is shown the statechart of the bounded buffer. The statechart describes the behavior of the bounded buffer in terms of its states and of the operations that force a change of state. The transition arrows express these changes and they are labeled by a triplet $\langle \text{event}, \text{condition}, \text{action} \rangle$. These triplets will identify the portion of code that provokes the change of state. The couple $\langle \text{event}, \text{condition} \rangle$ is used to express when the corresponding action can be performed, i.e., to define which constraints limit the applicability of the action. Of course, the action represents the code whose execution effectively provokes the change of state. The *event* it is used to identify the triggering event but it could be also used (as shown in the reported diagrams) to locate the action's code by letting coincide the label with the name of the method embodying that action. Moreover, a statechart can describe the behavior of all the instances of a class or of a specific instance as well. From these considerations, it is evident that the statechart describes the behavior of the bounded buffer in a compact and intuitive way providing also several levels of granularity (e.g., method calls are differentiated according to the state of the invoking object). Besides, the other diagrams (e.g., the sequence diagrams) permit, getting similar results, to deal with a trace of the program execution pointing out, for example, a particular sequence of method calls otherwise not manageable with the current pointcut definition language.

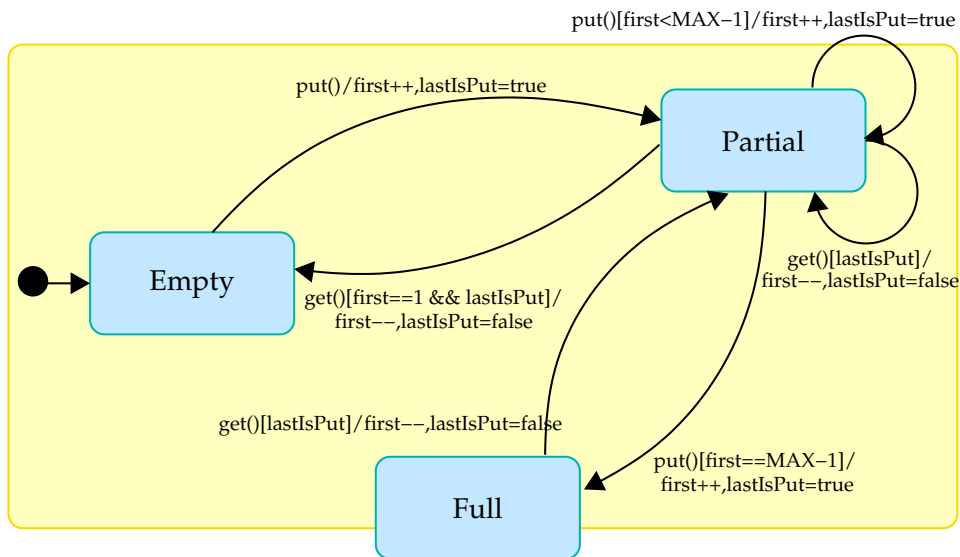


Figure 4.2: The Statechart of the Evolved Bounded Buffer

Hence, the UML diagrams provide a mechanism clear, intuitive and powerful for identifying portions of code associated with particular executions of the program but this is not enough to allow the evolution of a system: it is still missing a mechanism to determine how the system has to evolve, of course a mechanism that provides an high degree of intuitiveness, flexibility and granularity. In our opinion, the UML diagrams provide again the solution. The evolved system, as well as the original system, can be modeled by using the UML diagrams. The difference between such diagrams before and after the evolution represents the evolution itself. Whereas the original diagrams determine the code to be adapted, the evolved diagrams specify how the code has to be adapted, therefore the former contribute to the pointcuts/join points definition, the latter contribute to the advices definition. Figure 4.2 shows the statechart of the evolved bounded buffer. By comparing the diagrams in figure 4.1 with the one in figure 4.2 it is possible to understand how the new semantics of the method `get()` affects the whole behavior of the class and which code is involved. Each transition arrow stresses the portion of code subject to the evolution in a specific state and how it evolves. Just as an example, we are going to examine how the adaptation impact on the transition arrow from the *partial* state to the *partial* state (`get()` event). In the original design the arrow was labeled with:

$$\text{get() [first>0]/first--}$$

whereas, to respect the new requirements, the label changes in:

$$\text{get() [lastIsPut]/first--,lastIsPut=false.}$$

In this case, the triggering event is still the invocation of the method `get()` but both the condition and the action change. Originally, the method `get()` could be invoked when at least a value was contained in the buffer that it is expressed by the condition `first>0`. Then this constraint has been overwhelmed by the new semantics and it has been replaced by: the method `get()` can be invoked just after the invocation of the method `put()`; condition expressed by the boolean flag `lastIsPut`. Analogously, the action varies to deal with the boolean flag `lastIsPut`, that is, it is enriched by the statement: `lastIsPut=false`.

A criticism to the use of the UML diagrams to describe join points, pointcuts and advices could be moved to the fact that the diagrams have a pictorial nature and therefore to extract information from them is a difficult job. In the RAMSES project, Cazzola et al. [8,9] showed how to use the design information to evolve a system. They deal with the UML diagrams not in their pictorial form but encoded in the XMI language. The XMI is a standard variant of the XML language designed to render easy the extraction of features from the UML diagrams. Moreover, as shown in [9], the use of XMI consents to automatically determine the extent of the required evolution by comparing the diagrams: original and modified.

In this first phase of analysis, we have individuated all the problematic linked with the run-time software evolution, we have indicated a possible mechanism (AOSD) to support it, but there are still many problem. It lacks both a efficient mechanism to pick out the code involved by the evolution, and a technique to inject the new code in the application. Finally, it lacks a clear and appropriate definition of our JPM. In this preliminary work, we have individuated possible way to proceed.

Future Work

In the previous section we have presented our preliminary results, in particular we have analyzed aspect-oriented development techniques in relation with the software evolution problem. From our examination results that it is an hard job to determine the code that has to be evolved and to manage its evolution with the current join point models. The main problems are the granularity of the join point model and its dependence of the structure and syntax.

We have also showed that the UML methodology provides a more flexible, abstract and complete description model to use as a basis for a JPM more adequate for the software evolution issues.

Our future works will be to provide all that lacks, to allow an automatic run-time software evolution based on design information, in particular UML diagrams. To support our approach to run-time software evolution we will provide a JPM definition, a mechanism to determine the code affected by the evolution, and a technique to inject the evolved code in the right place.

The key issue of our thesis is the clear and appropriate definition of the join point model. Following the general definition given in [25] a join point model 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 and/or additional behavior that should run at join points, this mechanism is called *weaving* mechanism.

In the following section, we explain our idea to develop each element of the JPM.

5.1 Join Point

In current AO tools, such as in AspectJ, the join point is an action during program execution.

In our work, the join point has a little bit different extent, that is, it could be an

action during the program execution, single statements, block of statements, conditions, a particular behavior and events. In other word, a join point is all that it is possible to pick out by a UML diagram.

To indicate a join point we are evaluating to use the *annotation* mechanism. This means that our mechanism will look for the annotations referred by the pointcuts. The code or the byte-code will be annotated with the design information. This approach with respect to the more standard callback approach benefit that allows to treat a join point scattered in several part of the code as a single logical join point. Annotation languages are now integrated with programming languages, annotations provide a means to handle the object-oriented code and they can be readily compiled into run-time assertions.

Code instrumentation techniques and aspect oriented programming are two fields that are developed independently, but now the code instrumentation is a widespread implementation approach for the join point mechanism of aspect oriented programming. The idea is to instrument area in code that match the static part of pointcut, and then inserting dynamic checks for that part of matching that depends on run-time conditions, if needed.

We think that it is possible to develop an aspect language using existing code instrumentation techniques.

Annotations are one of the new language features in J2SE 5.0, and allow you to attach meta-data to any `JAVAC` construct. So, we think to adopt J2SE 5.0 to develop our project.

Unfortunately, `JAVAC` annotations can be only added to package declarations, type declarations, constructors, methods, fields, parameters, and variables, whereas we need to annotate more elements, such as single statement, block of statements, part of statement, and so on. In practice, for our work the current granularity of the `JAVAC` annotations hinders our work.

The same problem is faced in [6] for the programming language `C#`. In this paper Cazzola et al., present [a]`C#`, an extension of `C#` programming language supporting custom annotations on arbitrary code blocks or statement. The language extends the syntax of `C#` language to allow a more general form of annotation and provides a run-time library that extends the reflection support with operations for retrieving the information about annotations inside methods. They present an implementation of a compiler, which is a source to source compiler to reduce the extended model for custom annotations to the existing one with the helps of small modifications of the generated intermediate language.

To overcome this drawback also for the `JAVAC` language we are porting the ideas behind [a]`C#` to java with the help of a graduated student. The new default of `JAVAC` will be called `@JAVAC`

AOP makes it easier for you to encapsulate behavior that is usually harder or impossible to do with regular object-oriented techniques. Together, AOP and annotation make a new powerful combination that gives framework developers a more expressive way to work. For example you can use method annotation with AOP, and annotations allow you to define new keywords that you want to trigger your

special custom behavior and weave it into the execution of the method.

In our JPM, the join points are provided by the design information, in particular by the UML diagrams. The UML diagrams can represent several join points and a code location can be annotated by several join points, which can derive from several UML diagrams. Obviously, the UML diagrams cannot be used in their pictorial form, then our idea consists of using the XML code of the UML diagrams to annotate the code.

To pick out the right code from a UML diagram it is necessary to understand, in a more or less precise way, the code which implement such a diagram. In our opinion, on the basis of our preliminary analysis there are two possible techniques to face this problem, one is to use pattern matching, and another is to use a mechanism similar to Javassist.

Using a pattern matching technique you must figure out a possible portion of code generated by a UML diagram. In [28], Pintér et al. discuss the mechanism of the design pattern used for automatic generation of program code on the basis of high-level models, such as the UML statecharts. There are several common implementation of UML statecharts, the most common implementation approach uses two nested switch statements for partitioning the event handler function to segments reflecting the behavior of the object in specific states and sub-segments for each event handled in in the state The approach proposed by Pintér et al. is an extended variant of the Quantum Hierarchical state machine pattern. They use the Extended Hierarchical Automaton equivalent of a statechart for code generation. The code generated by these techniques will not completely reflect the one inserted in the application code, but it could give us a guideline.

Javassist is a class library for editing byte-codes in Java; it enables Java programs to define a new class at run-time and to modify a class file when the JVM loads it. Javassist provides two levels of API: source level and byte-code level. If the user uses the source-level API, they can edit a class file without knowledge of the specifications of the Java byte-code. The whole API is designed with only the vocabulary of the Java language. You can even specify inserted byte-code in the form of source text; Javassist compiles it on the fly. On the other hand, the bytecode-level API allows the users to directly edit a class file as other editors.

Javassist use the sample mechanism to adapt the component to inject in the system code. The same technique could be successfully used in our project. This technique could provide the code of the join points. The idea consists of generating samples of code for part and/or whole UML diagrams, and then to use these samples to retrieve in the code application similar portion of code. In practice, in our project the samples of code are used to pick out the portions of code that reflect the UML diagram analyzed, and then to insert the join points in the right places.

5.2 Pointcut Mechanism

The pointcut is the mechanism for identifying the join points. In the current AOSD approaches, a pointcut is a predicate on join points, which is used to recognize to which join points an advice must be applied to.

Since, in our model join points should be identified by annotations, the pointcut mechanism consists of a mean for identifying picking out these annotations.

In our idea a pointcut will not be a predicate, but a description of a behavioral trace of the system to be evolved. Such a description should be automatically driven by the evolution of the design information of the system itself.

The pointcut mechanism will be based on design information and the UML diagrams, the evolved UML diagrams of the application should automatically drive the selection of the join points involved by the evolution, whereas the difference between the original and the modified diagrams will provide the code that must be injected.

In the JAVQ 5.0 sun has introduced the support for annotations and the support mechanism to retrieve and manipulate these meta-data both at compile and at run-time.

At the moment, we are studying the feature of JAVQ 5.0, to use this annotation mechanism to identifying and analyzing the our points, both at compile and run-time, in the application code.

An annotation is a tag that they can insert into their source code. It does not alter the semantics of their code, but instead adds the ability for other application code to recognize and interpret the tag. As an example, you might want to be able to mark bits of code as `@joinpointA`, meaning that this part of code corresponds to a certain UML diagram or parts of this one. To support a `@joinpointA` tag, you should code an annotation type definition in `joinpointA.java`. Once you have compiled the annotation type, you can use it in your source code. Just like any Java type, the annotation type must be available on the classpath. You can then build a tool that uses reflection code to extract the annotation information from the code.

All elements that can be annotated, allow of extracting the annotation information by using the appropriate classes. For example, to determine whether or not an annotation has been applied to a method, you call `isAnnotationPresent` on the Method object and pass it the class object representing the annotation type.

5.3 Weaving Mechanism

In general, an AO program consists of a base program and some aspects that patch the base program. The main responsibility of an aspect weaver is to process aspects and base program, in order to produce the expected results. In our work the aspect corresponds to the evolution.

In current AO tools, the weaver parses aspect programs and collects information about the join points referenced by the program. Then, it locates these join points in the base program and finally weaves the code to implement what is specified in

the aspect code.

In our work the weaving mechanism should be different. The weaver should analyze the evolved UML diagram of the application, and should collect information about the join point referred by these diagrams. After, it locates these join points in the base program, finally on the basis of the difference between the original and the modified (evolved) diagrams will provide the code and the weaver inject the new code in the (join) points localized in the previous step.

Weaving aspects and classes involves program transformation techniques. This can be done using many technologies, but the most common approaches are: source code transformation and dynamic reflection.

The first techniques could be implemented using a compiler or a pre-processor, that usually provide an interface to add and edit nodes generated in the parsing tree. In this case, the woven code is reorganized at compile time, statically. Whereas, the run-time reflection needs the explicit existence of mechanisms to interfering into running computations in the base level to modify the entities behavior at run-time. For our project the first one is not useful, whereas the second one should be applied to our idea of weaving mechanism.

To support our project, we need a support for dynamic weaving.

The weaving mechanism works by adding new classes, new methods or modifying existing program components. This work must be done in several ways such as using the inheritance mechanism to implement the behavior described in the aspects, in this way the source code of the affected classes is not modified and the aspect code is inserted in the generated sub-classes.

Another possible approach to the weaving is the reflection. With this approach a meta-class is created by the weaver for each class referenced by an aspect, providing message interception mechanisms, allowing the addition of sentences related to aspects in the parsing tree, using information about the base level (affected classes)

We are interested to use a reflective mechanism to develop our weaver, and mechanism like javassist and design pattern to automatically create the advice code to inject in the base program.

Future Work

Bibliography

- [1] Jordi Alvarez. Parametric Aspects: A Proposal. 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] Keith H. Bennett and Václav T. Rajlich. Software Maintenance and Evolution: a Roadmap. In Anthony Finkelstein, editor, *The Future of Software Engineering*, pages 75–87. ACM Press, 2000.
- [3] Christoph Bockisch, Michael Eichberg, Mira Mezini, and Klaus Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proceedings of the 3rd Int'l Conf. on Aspect-Oriented Software Development (AOSD'04)*, Lancaster, UK, March 2004. 83-92, ACM Press.
- [4] Kai Böllert. On weaving aspects. In *ECOOP Workshops*, pages 301–302, 1999.
- [5] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, third edition, February 1999.
- [6] Walter Cazzola, Antonio Cisternino, and Diego Colombo. [a]C#: C# with a Customizable Code Annotation Mechanism. In *Proceedings of the 10th Annual ACM Symposium on Applied Computing (SAC'05)*, Santa Fe, New Mexico, USA, on 13th-17th of March 2005. ACM Press.
- [7] 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.
- [8] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. Software Evolution through Dynamic Adaptation of Its OO Design. In Hans-Dieter Ehrlich, John-Jules Meyer, and Mark D. Ryan, editors, *Objects, Agents and Features: Structuring Mechanisms for Contemporary Software*, Lecture Notes in Computer Science 2975, pages 69–84. Springer-Verlag, Heidelberg, Germany, July 2004.

- [9] Walter Cazzola, Ahmed Ghoneim, and Gunter Saake. System Evolution through Design Information Evolution: a Case Study. In Walter Dosch and Narayan Debnath, editors, *Proceedings of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE 2004)*, pages 145–150, Nice, France, on 1st-3rd of July 2004. ISCA.
- [10] Walter Cazzola, Sonia Pini, and Massimo Ancona. Evolving Pointcut Definition to Get 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 83–88, Oslo, Norway, on 15th June 2004.
- [11] Shigeru Chiba and Kiyoshi Nakagawa. Josh: An Open AspectJ-like Language. In *Proceedings of the 3rd Int'l Conf. on Aspect-Oriented Software Development (AOSD'04)*, pages 102–112, Lancaster, UK, March 2004.
- [12] Rémi Douence and Mario Südt. A Model and a Tool for Event-Based Aspect-Oriented Programming (EAOP). Technical Report TR 02/11/INFO, École des Mines de Nantes, November 2002.
- [13] Jim Dowling and Vinny Cahill. The K-Component Architecture Meta-Model for Self-Adaptive Software. In Akinori Yonezawa and Satoshi Matsuoka, editors, *Proceedings of 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'2001)*, LNCS 2192, pages 81–88, Kyoto, Japan, September 2001. Springer-Verlag.
- [14] M. Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.
- [15] M. M. Gorlick and R. R. Razouk. Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering*. IEEE, IEEE Computer Society Press, May 1991.
- [16] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley and Sons, March 2003.
- [17] Kris Gybels and Johan Brichau. Arranging Language Features for More Robust Pattern-Based Crosscuts. In *Proceedings of the 2nd Int'l Conf. on Aspect-Oriented Software Development (AOSD'03)*, pages 60–69, Boston, Massachusetts, April 2003.
- [18] G. S. Ling J. Peterson, P. Hudak. Principled dynamic code improvement. Research Report YALEU/DCS/RR-1135, Department of Computer Science, Yale University, July 1997.
- [19] Ivar Jacobson. The Case for Aspects. *Software Development*, pages 32–37, September 2003.

-
- [20] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *Proceedings of ECOOP97*, LNCS 1241, 1997.
- [21] 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.
- [22] Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler - A Powerful Back-End for Aspect-Oriented Programming. In Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-oriented Software Development*, chapter 9. Prentice Hall, 2004.
- [23] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11), November 1990.
- [24] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Company, 2003.
- [25] 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.
- [26] Doug Orleans. Incremental programming with extensible decisions. In *AOSD*, pages 56–64, 2002.
- [27] Harold Ossher and Peri Tarr. Hyper/J: multi-dimensional separation of concerns for JAVa. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*, pages 729–730, Toronto, Ontario, Canada, 2001. IEEE Computer Society.
- [28] Gergely Pintér and István Majzik. Program Code Generation Based on UML Statechart Models. *Periodica Polytechnica Electrical Engineering*, 47(3-4):187–204, 2003.
- [29] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic Weaving for Aspect Oriented Programming. In *Proceedings of the 1st Int'l Conf. on Aspect-Oriented Software Development (AOSD'02)*, pages 141–147, Enschede, The Netherlands, April 2002.
- [30] Jonathan Sillito, Christopher Dutchyn, Andrew D. Eisenberg, and Kris De Volder. Use Case Level Pointcuts. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'04)*, Oslo, Norway, June 2004.

- [31] Tom Tourwé, Kris Gybels, and Johan Brichau. On the Existence of the AOSD-Evolution Paradox. In *Proceedings of the Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT'03)*, Boston, Massachusetts, April 2003.
- [32] Tom Tourwé, Andy Kellens, Wim Vanderperren, and Frederik Vannieuwenhuysse. Inductively Generated Pointcuts to Support Refactoring to Aspects. In *Proceedings of Software engineering Properties of Languages for Aspect Technologies (SPLAT'04)*, Lancaster, UK, March 2004.
- [33] Alexandre Vasseur. Dynamic AOP and Runtime Weaving for JQVC- How Does AspectWerkz Address It? In Robert E. Filman, Michael Haupt, Katharina Mehner, and Mira Mezini, editors, *Proceedings of the 2004 Dynamic Aspect Workshop (DAW'04)*, pages 135–145, Lancaster, England, March 2004.