Computer Science

Ivar Bergman
Christian Göransson

An Introduction to Aspect-Oriented Programming

Bachelor's Project

2004:06

An Introduction to Aspect-Oriented Programming

Ivar Bergman
Christian Göransson

This report is submitted in partial fulfillment of the requirements for the Bachelor's degree in Computer Science. All material in this report which is not our own work has been identified and no material is included for which a degree has previously been conferred.

Ivar Bergman

Christian Göransson

Approved, 2004-01-15

Advisor: Mari Göransson

Examiner: Stefan Lindskog

Abstract

An overall goal in software development is to design a modular system where the modules are both easy to use, reuse and have well defined responsibilities. However, limitations in most programming language today can make this hard to achieve.

This thesis will present the theories behind Separation of Concerns (SoC) and Aspect Oriented Programming (AOP), and how these can be applied in software development, allowing for a more modular system design.

There are no programming languages today that have native AOP support, therefore we have chosen to focus on a Java extension, AspectJ.

We have concluded that the fundamental ideas from AOP and SoC are relevant in software development. AOP is a fairly new principle and it is impossible to predict how it will be accepted by the software community. However, AOP exposes problems in software today that will require a solution. Perhaps AOP will reveal itself to be a part of the solution.

Acknowledgements

We would like to thank our supervisor Mari Göransson for excellent supervision of this bachelor thesis and for providing relevant information on the subject.

Contents

List of Figures

List of Tables

1 Introduction

In this thesis we summarize how aspect oriented programming (AOP) is meant to improve software development in relation to object-oriented programming. We will also present some of the common difficulties designers face when using object-oriented design methods and how these can be avoided using AOP.

To introduce the subject, we will start this chapter with a discussion on program language evolution and what we can expect from new programming languages. We will also list our objectives from this article and the restrictions on them. The following three chapters will focus on AOP theories.

1.1 Background

A programming language should help the developer to focus on the problem at hand. Instead, is is often the case that the programmer has to take into account what environment the program will run in and that it should be executed on a specific machine. This demand for abstraction from the machine has driven the evolution of programming language.

Low level languages, like Assembler demands a great deal of machine knowledge of the program writer. Imperative languages, like C introduce new language constructs for flow control, memory management, procedures, etc. All these new constructs help the programmer to write more complex and more secure systems. However C is by far not a prefect programming language, it still requires in depth knowledge of computer architecture.

The next major paradigm shift, after the imperative languages, were the object-oriented programming (OOP) languages, whose primary goals are to increase the level of abstraction, modularity and the possibility of code reuse. To achieve these goals the developer has the concept of inheritance, polymorphism and abstract data types. Although object-oriented languages (OOL) have constructs to allow system to be designed in a modular way it is often not obvious how classes should be organized to achieve these goals. This problem has resulted in development of design principles and design patterns for OOP. In chapter 2 we present a design principle called Separation of Concerns (Soc). But as we will see, some ideas from SoC are hard to implement in classic OOL using inheritance and composition. A real life example is logging, that tends to spread across the entire system. AOP promise to provide a solution for this problem.

1.2 Objectives

Our objectives are to summarize the current state of the AOP research as well as present an aspect-oriented language. The AOP theories should be evaluated and explained by relevant example programs. The example programs should be simple enough to be useful to the reader when learning AOP. We should also present and discuss some of the most important open issues that AOP research groups have to deal with in the nearest future.

1.3 Limitations

Since AOP is a young field of computer science, most of the existing AOP languages are still only on a research level. We have limited our choice of AOP language and OO language to AspectJ and Java. We believe that AspectJ is the most mature aspect oriented language available today. In our objectives we stated that AOP should be evaluated by example programs. We will compare program size, crosscutting code and a free discussion on implementation complexity on two implementation of the same program, using OO and AOP languags.

1.4 Previous Work

Our thesis is a presentation and a summary of theories presented in other articles and papers. We will here list the most important reference article we have used. A full reference list can be found in the Reference appendix at the end of this thesis.

Aspect Oriented Programming [6] was the first article on AOP. It was published in 1997 by Gregor Kiczales and his co-writers and presents the foundation of AOP theories. It suggests a new decomposition model to achieve separation of concerns in software development. The article is based on a software engineering principle called *Separation of Concerns* [5] presented in an article with the same name. Since then, several new articles relating to the subject have been published. Most of the articles discuss different AOP languages and how they succeed in the goals stated by [6, 5].

Today there exists a number of languages that tries to implement support for AOP. We have focused on an extension of Java called AspectJ. Documentation for this language can be found on the AspectJ project web site [3].

1.5 Outline

Chapter 2 introduces the theories of Separation of Concerns, and the effects of crosscutting concern in the final program code. Chapter 3 explains new concepts and constructs in AOP. Chapter 4 presents an overview of the AOP programming language AspectJ, defining the language constructs and how they should be used. Chapter 5 illustrates how a persistence concern can be implemented in both a classic object-oriented language and an AOP language. Open issues and current state of the AOP research is outlined in chapter 6 together with some of our own comments and experience on the subject.

2 Separation of Concerns

The achieved complexity in software is tightly connected with the tools and languages used to create it. Today's high level object-oriented programming languages have enabled developers to create systems more complex than ever, but will it allow them to create the systems of tomorrow? Programming languages are constantly under development, with each new language trying to solve the problems of the previous. Object-oriented languages are no exception; they too have problems that need to be addressed if the systems of tomorrows are to see the day of light. Aspect-oriented programming has been presented as a solution to some of the problems, but before we start to look at the theories of AOP we need to create an understanding of the problems in contemporary OO languages. This chapter will present the "why" of AOP. The "how" will follow in subsequent chapters.

2.1 Background

In general, a software development process starts with a specific problem that should be modeled in a programming language. A phase of analysis and design begins where the problem is decomposed into a set of subproblems that are smaller and less complex compared to the original. A problem is decomposed into units of the programming language currently considered. The units might be functions in structural programming or objects in OOP. These units will, during the implementation phase, be composed into a working application using composition mechanisms in the language, these include inheritance and aggregation in OOL.

Often it is not enough to describe the problem by fundamental computational algorithms. Special purpose computing requirements such as concurrency, distribution, real-time constraints, location control, persistence, and failure recovery may be needed to fulfil special requirements of the application, or to manage and optimize the basic computational algorithms.

A programming language should be able to implement these requirements as separated and self-contained software units. OOP has introduced abstract data types, polymorphism and inheritance, but as we will see in the following example, there are requirements that OO languages fail to separate.

2.1.1 Synchronization example

Consider an application [10] that works asynchronous on shared data. In an OO language the data may be encapsulated in a `Data` class, and the synchronization requires a mechanism to handle locking. A common solution is to let the `Data` class inherit an abstract class that provides two methods, `lock` and `unlock`. When using this solution, every method that works on the data must handle the locking to make sure that no other object is accessing the data at the same time. This might seem like a good approach, but it does have some drawbacks:

- A lot of changes is required to `Data` if every method is to implement the code for locking and unlocking.
- There might be occasions when the `Data` class must inherit from other classes as well. In a single inheritance language, like Java, this is impossible.
- It will be harder to reuse the `Data` class in a context where synchronization is not required.

The main problem in the example is that the synchronizing requirement and that the data encapsulation can not be separated into two different software units. The synchronizing code is mixed into the code of `Data`. This is an example of an important and common problem in software development today. It is not a new phenomenon, and the will to solve and identify it has been around for a long time under the name Separation of Concerns (SoC). The next sections will describe the implications of successful and unsuccessful separation of the concerns.

2.2 Separation of Concerns

“A concern can be any cognitive element that can be considered while building a program (e.g., a protocol, a feature, a requirement, etc.)” [9]. Given this definition we see that a concern could be any property of a software system. Separation of Concerns [5] focuses on identifying, distinguishing and separating concerns from each other. The goal is to keep the concerns as separated, isolated and independent from each other as possible.

In the synchronization example above, we can identify two concerns: the storage concern and the synchronization concern. These two are not separated in the implementation and the previously described drawbacks clearly illustrate what happens when SoC fail.

There are many benefits from achieving a good separation of concerns:

- **Simplified development:** A clear separation of concerns allow the developers to focus on one concern at a time, not worrying about all the concerns in the system.
- **Code reuse:** Objects that are specialized to operate in a particular environment will be hard to reuse in another context. The synchronization example defined an object `Data`, handling a synchronization concern. The synchronization code was mixed with the code to handle the data. Reusing this object without the synchronozation code is thus impossible.
- **Simplified maintenance:** Program code is always subject to changes. Usually this is not done by the person who initially wrote it. This means that the code first has to be understood if a change is to be made. To understand a component implementing several different concerns, the concerns must first be separated from each other, allowing the developer to make a change in the right place.

2.2.1 Separating Concerns

Concerns are separated into basic concerns and special concerns (sometimes called system-level concerns). Basic concerns define the real purpose of the system, without being dependent on requirements like security and performance. The basic concerns specify what is really important for an application program [5].

The special concerns include optimization, security and synchronization and are used to manage the basic concerns. Special concerns provide support for the basic concerns and therefore play an auxiliary role [5].

In a system to manage banking accounts, the basic concern could be to withdraw and deposit money. This is the purpose of the system; what it is expected to do. As always when dealing with money, security is a priority and the system would never be taken into production without it. But security is not at the essence of the system, it merely provides support to the basic concern, allowing it to function as intended.

2.2.2 Separation at Two Levels

During the software development process, we distinguish between SoC at two different levels:

- **Conceptual level:** The different concerns are identified during the design phase as abstract properties of the system. It is important to clearly identify each concern and separate them from each other, making sure that a concern is not a composition of several others.
- **Implementation level:** The goal at the implementation level is to get a clear mapping from the concerns identified at the conceptual level to units of the programming language. How well this can be achieved depends on the language of choice. Ideally, the blocks of code addressing the different concerns should be clearly separated, and be loosely coupled.

Its important to separate the two levels, and the separation achived at them. The example in figure 2.1 shows that the conceptual level has a good SoC, where the concerns are clearly separated from each other. However, this clear separation is not always available

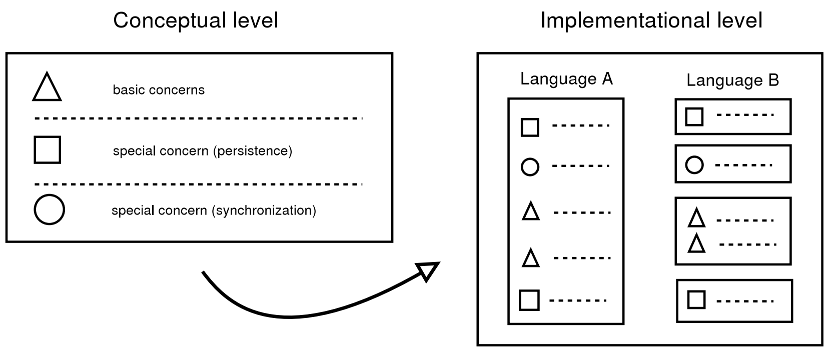

Figure 2.1: Separation of concerns at both conceptual and implementation level.

at the implementation level. The concerns have been implemented using two different languages A and B. In language A, a tight coupling has been introduced between the concerns, not keeping any of the separation achieved at the previous level. The implementation in laguage B is a more direct mapping, keeping the desired separation.

The attained SoC at the implementation level is thus dependent on the programming language of choice, and not a property of the concerns themselves.

2.2.3 Crosscutting Concerns

As we saw in the previous section, the level of independency achieved in the implementation is dependent on the programming language. The goal is always to have the concerns implemented as separated units, but what happends when this is impossible? Concerns impossible to separate during implementation are called crosscutting concerns and cut across multiple concerns, modules or class hierarchies. A simple example of a crosscutting concern is logging, which affects all other parts of a software system, making it impossible to isolate.

As we saw in the synchronization example, certain properties of the problem were not clearly separated. The `Data` object was responsible for managing the locking method calls. This inability is a symptom of a problem with the programming language, since it does not allow the problem to be composed in an independent manner. The problem with contemporary programming languages is that they only support one composition mechanism (one-dimensional composition). This is called the “tyranny of the dominant decomposition” [7]. If a clear separation is to be attained, a language supporting multi-dimensional composition is required.

2.3 When separation of concerns is not achieved

Code tangling and scattering are implications of poor separation of concerns. These are important issues that developers must be aware of to develop high quality software. It will not only compilcate the development, but will make the maintenance much harder.

Code Tangling - One unit concerned with several concerns

Code tangling occurs when a module is required to handle several different concerns simultaneously. The concerns often include persistence, logging, synchronization and security. Code for the basic concern becomes tangled with the code from the special concerns. The tangling makes the concerns hard to separate and leads to problems when changes are required to the module. Problems arise from the fact that you cannot address a problem directly, ignoring the tangled code. This is concluded in [6], where it is stated that “tangled code is extremely difficult to maintain, since small changes to the functionality require mentally untangling and then re-tangling it”. Consider the following example:

```
makeWithdraw(Account acc) {
   if (owner(acc)) {
      prepareAccount(acc);
      withdraw(acc);
      logWithdraw();
      releaseAccount(acc);
   }
}
```

It shows an example of tangled code. The purpose of this function is to withdraw money from an account. It is a simple request that is handled on line 4. However, other concerns like security, synchronization and logging is tangled with the code to withdraw, making it hard to read and understand.

Code Scattering - Several units concerned with one concern

Since crosscutting concerns, by definition, spread over many modules, related implementations also spread over all those modules. For example, a logging concern in a system may affect all modules in the system.

Problems related to code tangling and code scattering:

- **Impact of change:** Simple changes spread throughout the system.
- **Lower productivity:** Focus on the main concern is lost when a developer is simultaneously implementing multiple concerns.
- **Less code reuse:** Other systems requiring similar functionality may not be able to reuse a module, because it implements multiple concerns.
- **Poor code quality:** Code tangling produces code with hidden problems. Moreover, by targeting too many concerns at once, one or more of those concerns will not receive enough attention.

2.4 Classifying tangled code

This section is a summary of [4], in which an attempt has been made to classify tangled code. By examples, it has been shown that there is a difference between the tangled code itself and how this code crosscuts existing structures. Two examples will now be presented that is used to clarify the concepts in this section.

2.4.1 Examples of tangled code

Example one

```
public class A {
  int i, j;
  ArrayList observers = new ArrayList();

  public void setI(int i) {
    this.i = i;
    informObservers();
  }

  public void setJ(int J) {
    this.j = j;
    informObservers();
  }

  public void attachObservers(Observer observer) {...}

  /* Additional code for managing observers */
}

public class B  {
  ArrayList observers = new ArrayList();

  public void attachObservers(..) {..}

  /* Additional code for managing observers */
}
```

Example one shows two classes implementing the observer pattern and contain exactly the same code. This can be solved by creating a new class Observable which A and B can extend. However, since many languages (including Java) only support single inheritance, this is not always practical.

Example two

```
public class SingletonA {
  static SingletonA instance = null;

  private SingletonA() {..}

  public static SingletonA getInstace() {
    if (instance == null)
      instance = new SingletonA();
    return instance;
  }
}

public class SingletonB {
  static SingletonB instance = null;

  private SingletonB() {..}

  public static SingletonB getInstace() {
    if (instance == null)
      instance = new SingletonB();
    return instance;
  }
}
```

This example shows two classes SingletonA and SingletonB, implementing the singleton design pattern. The code introduced in the two classes are identical in structure, but differ in variable names.

2.4.2 Crosscutting Code

This section is a brief explanation of three different kinds of tangled code, focusing on the way how this code crosscuts existing structures.

Unit Dependency. If the code depends on the unit which is affected by the crosscutting, or where the crosscutting is located, we say that the code is unit dependent. Otherwise it is unit independent. Two kinds of unit dependency are identified:

- Object dependency. Object dependent code requires that an object contains a specific method or field. The call to `informObservers()` in A.setI is object dependent, since it requires that the object contains the method `informObservers()`.

- Method dependency. Code dependent on method related information, like parameters and local variables in a method, is method dependent. `notifyObservers(bar)` in the following code-snippet is method dependent, since it depends on the local variable `bar` in the method `foo`.

  ```
  public void foo() {
      int bar = 5;
      notifyObservers(bar);
  }
  ```

  The call to `informObservers()` in example one does not depend on any information on the methods and is therefore method independent.

Constant vs. Variable Crosscutting Code. Example one contains constant crosscutting code. The implementation for an observer does not change between the classes and could easily be moved to a new class.

The code in example two is an example of variable crosscutting code; it changes between the two classes. The code to implement a singleton in SingletonA and SingletonB is almost identical, but is dependent on the name of the class.

Transforming Crosscutting Code. Transforming code changes the declaration of the class it is applied to. Extending class A from Observer, instead of implementing it in class A, would be class transforming crosscutting code. Example one contains both class and non-class transforming crosscutting code. The call to `informObservers()` does not change the interface of A, but the introduction of the method `informObservers()` does.

2.4.3 Crosscutting

Crosscutting describes how code crosscutts a given position.

Constant vs. Variable Crosscutting. As an example of variable crosscutting we look at the singleton example. The code to implement a singleton is applicable to any other class wishing these certain properties. Thus the variable part of the crosscutting are the classes to which the implementation is applied to.

In a persistent Java Bean (see chapter 5), which stores its properties in a database each time they change, we have class-specific crosscutting code for each query. This code would make little sense in another context, since it is so tightly connected to a definite class, and is consequently an example of constant crosscutting code.

Crosscutting Location. We distinguish between crosscutting location and crosscutting affected units. Crosscutting location describes that kind of crosscutting which is restricted to a particular unit, while crosscutting affected units describe those units which are affected by the crosscutting. The unit we will look at is a class.

The calls to `informObserver()` in example one are spread across a single class, making them class located. The observer implementation is spread across several classes, but occurs exactly once in each of the classes, which makes it class affecting.

Static vs. Dynamic Crosscutting. A crosscut is said to be static if the crosscutting code exists only for so long as the affected unit exists. The methods for managing observers in example one and `getInstance()` from the singleton example, are examples of static crosscutting; they only exist while the affected classes exist.

3 AOP Theory

In this chapter we will look at AOP terminology and the goals of the AOP principles. We have chosen to discuss AOP in relation to object-oriented programming only, but the theories are applicable to imperative languages as well. Again, we must emphasize that AOP is a way to realize the theories from the SoC principle (see chapter 2), and especially the concept of concerns are of importance. In this chapter we will not use code examples to illustrate the theories of AOP, they are postponed to chapter 4 where the AOP language AspectJ™will be introduced.

AOP was first introduced in an article written in 1997 by Gregor Kiczales et al at Xerox Palo Alto Research Center. The article was simply called *Aspect-Oriented Programming* [6] and presented an analysis of "why certain design decisions are hard to clearly capture in actual code" and also the means of how these design decisions may be decomposed.

3.1 Aspects and Components

By definition, an *aspect* is a cross-cutting concern, i.e. concerns that are hard to capture in an encapsulated software unit and that is cross-cutting the system's basic functionality.

SoC divide concerns into two groups: basic and special concerns[1], AOP makes the same division but use the terms *aspects* and *components*. Consider a system and its implementation using any object-oriented language, then a concern is:

- A component, if it can be cleanly encapsulated. Cleanly means well-localized and easily accessed. Just as the basic concern of SoC, a component tends to be a unit of the system's decomposition.

- An aspect, if it can not be cleanly encapsulated. Like the special concerns of SoC, an

[1]In some AOP related articles[6] the term *property of a system* is used instead of *concern*. We belive that the terms are equivalent, and we choose to use the term concern throughout this paper to emphasize the connection to the SoC principle.

aspect is a concern that affects the requirements of an application or that manages and optimizes the basic concerns, i.e. components.

3.2 Join point, point cut and advice

Assume that we have decomposed a problem into components and aspects, how can we compose them to build the desired system? AOP has identified three language concepts (join point, point cut and advice) that provide a new composition mechanism for aspects and components.

Join points are well-defined points in the execution of a program, like: method calls, field access, conditional checks, loop beginnings, assignments and object constructions[8]. Join points are not necessarily explicit constructs in the component language. Rather like nodes in the dataflow graph and runtime method invocations they are clear, but perhaps implicit, elements of the component program's semantic[6]. It is important to get a clear understanding of the join point concept, since the whole idea of AOP is to execute some aspect specific code at these join points.

The program must be able to select a subset of all join points in the program flow. This is done by a language specific construct called *pointcuts*, or pointcut designators. Any AOP language must provide some syntax to allow the programmer to define point cuts. Most AOP languages use some sort of pattern matching mechanism to idenify joinpoints. This is a language specific syntax. We will see an example of this in the next chapter where the AspectJ pointcut syntax is presented.

After the point cuts have been defined, each of them should be associated with the aspect code. This construct is called an *advice*. However there are more options for the execution of the aspect code, e.g. should it execute before, after or perhaps around the specific join point? An AOP language must provide some constructs to allow the programmer to specify the time of the aspect code execution.

All three concepts (join point, point cut and advice) are grouped together into a software unit that is called an *aspect.* Depending on the AOP language the syntax of an aspect will of course differ, nevertheless any aspect will contain them.

3.3 Weaver

To get the final executable program the aspect must be woven across the components of the system. This can be done either statically or dynamically. The static weaver modifies the source code by inserting the advices as inline code at the defined join points. A static weaver is a pre-compiler that automates the generation of the source code. One problem with static weaving is that debugging is hard since the executable program has no way to determine if a specific code routine belongs to an aspect or a component. A example of a static pre-compile weaver is *AspectJ.* We have dedicated the next chapter to the study of how the AspectJ language realizes AOP.

Dynamic weaving means that aspects are woven across the components at run-time, i.e. the aspects can be adapted and replaced dynamically during run-time. Usually this is managed by an AOP framework and the program must be executed in the run-time environment provided by the framework. The flexibility gained from dynamic weaving will on the other hand reduce performance compared to static weaving. This paper will not focus on dynamic weavers since they are less mature compared to static weaver languages.

Aspect weavers must be able to process both the component language and the aspect languages, and composing them properly to produce the desired total system operation. Essential to the function of the aspect weaver is the join points, which are those elements of the component language sematics that the aspect programs coordinate with [6].

3.4 Aspect and component languages

As we saw in the section 3.2, an AOP language must have the possibility to define point cuts and advices. As of today there are no object-oriented programming languages that support these constructs, and therefore most AOP languages are language extensions of existing programming languages. There are of course alternative to language extensions, an AOP framework (AspectWerkz) use reflection techniques to weave the aspects into the program execution flow.

3.5 Goals of AOP

The general goal of AOP[6] is to support the programmer in cleanly separating components and aspects from each other, by providing mechanisms that make it possible to abstract and compose them to produce the overall system.

An interesting observation is that imperative and object-oriented languages only have the possibility to decompose the system into components. However the need of other decomposing mechanism has resulted in new language constructs, like the try-catch statements. Kiczales [6] argues that the popularity of this statement is a result from the fact that it is based on a different decomposition mechanism compared to the functional decomposition mechanism in structural languages.

4 AspectJ

This chapter will introduce the concepts and constructs of the AspectJ language and how it relates to Java. To get an understanding of the new semantic features of AspectJ there are several language constructs that the programmer must learn. Each new concept will be explained both by using a clear definition as well as a short code example of how the concept may be used. As reference for the entire chapter we have used the AspectJ reference documentation [3].

Since 1970, Palo Alto Research Center (PARC) has conducted research in several fields of computer science. Some projects have focused on methods to cleanly capture complex design structures in software implementations. This research has emerged into the development of AOP theories and AOP languages. The first AOP languages were special-purpose language that focused on a specific set of concerns. In 1997 PARC initialized a research project to develop a general purpose AOP language. The project is still active, as a open source project maintained by eclipse.org [1], and has developed AspectJ into a production ready AOP language.

4.1 AspectJ concepts and constructs

AspectJ is a language extension to Java. It adds AOP capabilities to the Java language by introducing new constructs. AspectJ uses static weaving, which means that aspects are woven into the Java classes before runtime.

One of the key concepts in AOP theory is the join point, which defines the points where aspects are allowed to be woven into the components. Note that the join point is not a language construct, i.e. it is not a keyword, but rather points in the Java program flow. AspectJ provides a language construct (point cuts) that identify join points.

Every join point has a unique signature that allows AspectJ access to a particular join point. Consider a class method, it is uniquely defined by the name, return type, parameter list and throws clause. An aspect can therefore define point cuts that pick out specific method calls.

The point cut uses a simple pattern matching syntax to identify join points, with the possibility to use wild cards. A pattern could for example have the following form: `public void *.foo(int)` that matches method signatures in both class `A` and class `B` listed below. The asterisk is a wild card that matches any class name.

The "`..`" wild card matches any list of parameters of any type. For example `public void A.foo(..)` would match the two methods is the class `A`.

```
public class A {
  public void foo(int arg1);
  public void foo(int arg1, int arg2);
}

public class B {
  public void foo(int arg);
}
```

Let us implement a simple system logger aspect, that writes a log message to the standard output stream whenever one of the `foo` methods are called. All details of the aspect will be explained later in this chapter.

```
public aspect LogAspect {
  pointcut fooLog(): call(public *.foo(..)) {
    System.err.println("Log Message: foo() has been called.");
  }
}
```

Table 4.1: LogAspect

Join points can not be identified uniquely by a pattern alone. A class method has join points on both method call and method execution. AspectJ has introduced a set of *primitive point cuts* that when used together with a pattern allow the programmer to identify all possible join points.

The following sections present the join points emerging from the Java programming language and simple examples of AspectJ point cuts that pick out the specific join point.

4.1.1 Method related join points

A method call and the execution of a method body are two join points that may be captured by the primitive point cut designators `call(`*`MethodPattern`*`)` and `execution(`*`MethodPattern`*`)`. The following example defines a point cut that captures all calls to public methods.

```
pointcut publicCall(): call(public *.*(..));
```

4.1.2 Field related join points

With respect to non-constant fields, there are two well defined join points: when a field is assigned a new value, and when a field is referenced. To capture these join points, AspectJ provides the `set(`*`FieldPattern`*`)` and `get(`*`FieldPattern`*`)` point cuts.

```
aspect VerifyPos {
    static final int MAX = 100;
    before(int newval):
        (set(int Position.x)|| set(int Position.y))
        && args(newval) {
        if (Math.abs(newval) >= MAX)
            throw new RuntimeException();
    }
}
```

The assign join point has one argument, namely the new value assigned to the field. The VerifyPosition aspect above is an example of how the assign argument could be used. The aspect prevents the absolute value of the field `x` or `y` of a class `Position` from exceeding a maximum value.

4.1.3 Object creation related join points

Several join points can be defined in the creation process of a Java object. AspectJ uses four different join points:

`call(`*`ConstructorPattern`*`)` is the point where the initial constructor is called, i.e. not when a constructor is called by super or this. At the constructor join point the object is considered to be an instance of the class.

`execution(`*`ConstructorPattern`*`)` is the body of an actual constructor. The object being constructed is the currently executing object and may therefore be accessed using `this`.

`initialization(`*`ConstructorPattern`*`)` is the point cut when a class is initialized. Before the initialization, defined class member variable will have the default value, for instance integer variables will have default value 0. However if the class is a subclass, the super class will have its own constructor executed before the initialization join point.

`staticinitialization(`*`ConstructorPattern`*`)` picks out the join point where the static initializer is being executed. However, if the class is a sub class, the super class will have its own static part executed before the static initialization join point.

4.1.4 Exception handler execution related join point

The execution of an exception handler provides a join point that may be used to capture exceptions by the point cut designator `handler(`*`TypePattern`*`)`. The following example will print a stack trace debugging message when a `MyException` is handled.

```
aspect DebugMyException {
    before(MyException e): handler(MyException) && args(e) {
        e.printStackTrace();
    }
}
```

4.1.5 State-based related join point

As defined above, a join point is a point in the program execution flow. By using the point cut designators in AspectJ we can access any joint point. Sometime we also need to determine from where a join point is being captured. AspectJ provides three primitive point cuts to capture join points when an object is being: executed, operated on or passed around.

`target(`*`Type or Id`*`)` picks out the join points where the object on which a method or a field is being used is of a specific type. If the target argument is an identifier then the target object must be bound in the points cut declaration.

`this(`*`Type or Id`*`)` will pick out the join points where the executed object is of a specific type. If the target argument is an identifier the target object must be bound in the points cut declaration.

`args(`*`Type or Id*`*`)` picks out the join points where the point cut declaration arguments are instances of specific types.

To illustrate the state-based join points, we can continue the previous example concerning the VerifyPosition aspect. If the Position class should be used for different board games (e.g. ChessBoard or ReversiBoard) we need different coordinate limits depending on what game using the Position object. To differentiate between the chess and reversi game classes we can use the primitive point cut designator `this()`.

```
aspect VerifyMarkerPosition {

    static final int REVERSI_MAX = 10;
    static final int REVERSI_MIN = 1;

    static final int CHESS_MAX = 8;
    static final int CHESS_MIN = 1;
    pointcut positionChange(int newval):
        (set(int Position.x)|| set(int Position.y)) &&
        args(newval);

    before(int newval): positionChange(newval) &&
        this(ReversiBoard)
    {
        if ( (REVERSI_MIN >= newval) && (newval <= REVERSI_MAX) )
            throw new RuntimeException();
    }

    before(int newval): positionChange(newval) &&
        this(ChessBoard)
    {
        if ( (CHESS_MIN >= newval) && (newval <= CHESS_MAX) )
            throw new RuntimeException();
    }
}
```

4.2 Point cut definition

A point cut is a program element that picks out join points and exposes data from the execution context of those join points. It is defined by composing primitive point cut designators and user defined point cuts using boolean operations. For a complete list of primitive point cuts in AspectJ, see appendix A.

Like class methods, a point cut has an access modifier: public, private, protected or default and they can be declared either in a class or an aspect. A point cut is treated as a member of that class or aspect. Point cuts that are not final may be declared abstract, however those must then be defined in an abstract aspects.

Often an aspect needs to access the context of the executing join point. A point cut is defined with an interface and may expose variables of the execution context, by providing a formal parameter list in the point cut definition. Any variables that the aspect code must access from the join point must be a part of the parameter list in the point cut definition. The aspect below, verifies a coordinate value in a Position class, illustrates how join point context can be exposed.

```
pointcut verify(int x):
  call(* Position.setX(int)) && args(x);
```

Any aspect using the above point cut has access to the `x` argument of the `setX()` method call. The point cut simply states that `x` is a local variable of the point cut `verify` and that the join point should have an argument named `x`.

4.3 Advice

To get the final bits and pieces together we must use the point cuts together with aspect code. This is done by a language construct called an *advice*, of the form:

```
AdviceSpec [ throws TypeList ] : Pointcut { Body }
```

where an *AdviceSpec* is either *before*, *after* or *around*, and defines where the Body (the aspect code) should be woven into the program component, either: before, after or around the join point. Observe that we have a language construct that we can use to define the cross cutting behavior of the program.

An advice can be defined with a single point cut, or an expression of point cuts using boolean operators. If join point context should be exposed to the advice code, the exposed variables must be declared as an argument parameter list in the advice specification.

Again, let us consider the aspect that verifies coordinates in a Position class. We need to implement an advice that defines a point cut that picks out the set-methods of the Position class, and the advice code should perform the verification.

```
    private static int MIN = 0;
    private static int MAX = 10;

    pointcut verify(int v):
        (call(* Position.setX(int)) || call(* Position.setY(int))) &&
        args(v);
```

```
before(int v): verify(v) {
    if ( (v < MIN) || (v > MAX)) {
        System.err.println("Invalid coordinate value " + v);
    }
}
```

4.4 Inter-type declaration

An inter-type declaration is an external member declaration in a class. This means that an aspect can add new fields, methods and constructors into other classes. For example if we need to create an adapter method for a third party class, we could simply add a new method by inter-type declaration, instead of using the design pattern Adapter, that is the common solution in object-oriented programming.

The syntax for inter-type declaration is like a standard declaration in Java except that we need to explicitly state the target class. For example to add a new counter variable in a class `A` we should write `private int A.counter;`.

Another consequence is that a class can be specialized with new methods even if it already extends a super class. This way to declare members is considerably useful when the member is cutting through the class hierarchy and is hard to encapsulate by only using inheritance.

AspectJ also have the possibility to add new super classes to a class. By using the `declare` syntax we can add new interface classes to any class, and also add a new super class (if not already declared).

4.5 Aspect

We now have the tools to declare point cuts and inter-types, and how to define advices. But we need to be able to localize these cross cutting behaviors into a software unit. It can not be done in the class construct since we need a way to separate core components from aspects. AspectJ has introduced a new language construct simply called, `aspect`. The declaration syntax for the aspect is much like the declaration of a class, but it can include point cuts, inter-type declaration and advice.

Just like in a Java class you may have normal class members in an aspect. In fact you may implement an aspect in the same way as a class since it is transformed into an ordinary Java class by the AspectJ compiler (`ajc`).

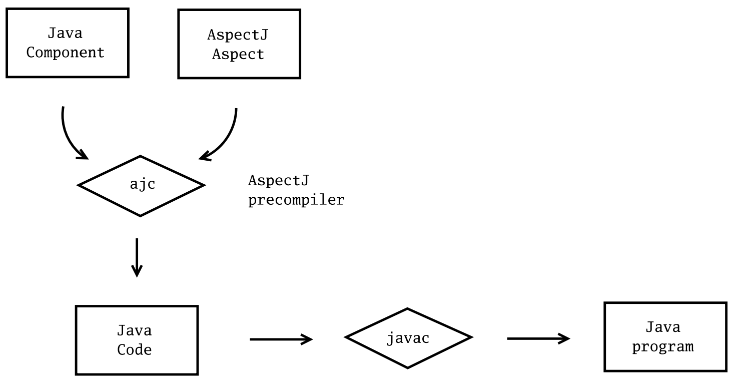

Figure 4.1: AspectJ weaver

An interesting question is how to interpret semantic meaning of the inheritance between an aspect and a class. In standard object-oriented meaning an inherited class is usually viewed as a specialization of the superclass. But the special properties of aspects are that they cross cut the inheritance structure, so we should not interpret the relation in terms of inheritance but rather just see it as a way of improving code reuse.

5 Persistent Java Bean Example

In this section we will present a specific example program and compare two different solution alternatives, one that uses an AspectJ implementation to separate the different concerns and the other is implemented using a common object-oriented design. The example itself is rather easy, the idea is that two Java Beans should be persistently managed in a relation database. The two beans are `UserBean` and `DepartmentBean`. Each bean has a set of properties that define the basic functionality, one of these properties acts as a unique identifier (primary key). The first example implementation will illustrate a common technique to solve the persistent problem, and the second example illustrates an AOP solution to the problem.

5.1 The Persistence Concern

The persistent managing of objects is a common problem in object oriented software development. There are a number of questions that need to be addressed to achieve a good system design: Who should be responsible for the persistent managing? Should it be the object or should it be the client using the object? Should the persistent handling be transparent? I.e. should the client be unaware of the fact that the object is stored. The persistence concerns must be regarded as a special concern, since it is applied to the JavaBean classes (the basic concern in the program).

Today there exist several techniques to handle persistence concern. The simplest way is to use plain old Java and let the object itself handle the calls to a database, using straight forward SQL (Structured Query Language) queries through JDBC (Java Database Connectivity). As we will see later in this example this technique will result in cross cutting code. Another alternative would be to use some design pattern and separate the responsibility into different classes or some reflection mechanism. This technique is used by Java Data Objects (JDO) that transparently let the programmer access the under lying storage medium without any database specific code[2].

5.2 Example Description

Appendix C contains the code for the two different solutions of the persistent JavaBean example. The background of the problem is to store a `UserBean` and a `DepartmentBean` in a database. Both beans have a unique identifier and may therefore be updated or stored from the database. Any changes of a bean should be reflected in the database.

```
+---------------+  +----------------+
| UserBean      |  | DepartmentBean |
+---------------+  +----------------+
| getUid()      |  | getDid()       |
| getFirstname()|  | getName()      |
| getLastname() |  | getAdmin()     |
| setUid()      |  | setDid()       |
| setFirstname()|  | setName()      |
| setLastname() |  | setAdmin()     |
+---------------+  +----------------+
```

A DepartmentBean has a field `admin` of type UserBean, all other bean properties are of type String. When the userid property of a UserBean object is set the other properties should be updated from the database if userid exists. Whenever a property is set or changed the values in the database should be updated.

The system has three different concerns; the first concern is classified as a component and represents the bean functionality. The second concern is an aspect that handles the connections to the database, i.e. creates a new connections and release existing connections. Finally, the third concern is related to the update and store operation for each bean to implement the persistent of the objects.

[2]JDO succeed in separating the persistance concern from the Java Bean code by using a framework solution. The difference between this solution and AOP is that AOP provide means to separate any concern in a system, but JDO provide a concern specific separation.

5.3 First solution - plain old java

In the first solution we have added two new methods directly to the bean classes, that handles `update` and `store` operations. When a bean property has changed the bean must handle the calls to the corresponding database access method.

```
public void setFirstname(String firstname) {
    this.firstname = firstname;
    storeUserBean();
}
```

This solution leads to tangled code, since all three concerns are mixed into the same classes. Also the code is scattered, since the persistent method `store` is called from more than one method. This implementation will reduce reusability of the bean class, since it will be impossible to use in an environment without persistence.

Usually the persistent methods `store` and `update` can be implemented in a subclass of the UserBean class. This will allow a better reuse of the Java Bean classes, but it will not reduce the scattering and tangling of the program.

5.4 Second solution - aspects

The major benefits from the AOP implementation are that it separates all concerns and still let the programmer have full control of database access calls. In this specific example there is no significantly reduce of the number of lines in AOP implementation and the non-AOP variants, but this is expected since there is only one class affected concern namely the database connection allocation/deallocation. The other concern, JavaBean persistence, is a class located concern and is therefore not possible to reuse.

6 Current State of AOP

We will in this chapter summarize the open issues the AOP research groups must deal with in the nearest future, and also present two alternative techniques to AOP. In the last section of the chapter we give some personal comments on our experience of using AspectJ as a programming language.

6.1 Open Issues

AOP has over the last year gained much attention. Hundreds of articles have been written on different areas of AOP. But still there are questions and open issues to be addressed. Some of them are related to how applicable AOP is in larger project when there is a greater need to formalize the design of aspect. For this to work, a proper notation to describe how aspects work and how they interact with the system must be defined. Some research has suggested a UML (Unified Modeling Language) notation for aspects. But as far as we know, no AOP notation is planned by OMG (Object Management Group).

An interesting feature of AspectJ is that it has the possibility to modify declaration of methods and classes. How does this align with the goals of design by contracts? Can a developer rely on a contract if an aspect writer may have changed the class declaration?

An important factor for the acceptance of AOP is that if AOP concepts will be integrated natively in popular programming language like Java. We have not seen any discussion on public forums that indicates that this should be the case.

6.2 Alternatives to AOP

We believe that the SoC principle has formulated some very important properties of the software design process, but the question is if AOP is the right solution to the problem? There are other language and techniques [9] that aim to improve the possibility to separate concerns in a system:

- Subject Oriented Programming (SOP) or Multi-Dimensional Separation of Concerns main idea is a *hyperspace* that contains specification for dimension and concerns of importance. Contrarily to AOP, SOP does not make any different between special and basic concerns. This means that all concerns can be composed into the final system. HyperJ [2] is a tool which provides support for Hyperspaces in Java™.

- Adaptive Oriented programming or Demeter Method has focused on the fact that sometime it is easier to solve a general problem than solving a specific problem. It is regarded as a law of nature that software evolve due to specification changes, and therefore software developer do best in writing programs that can adapt to context changes.

All these programming models argue that the software community must allow more decomposition alternative than the ones provided by OOL. They do not mean that the OO paradigm is useless, they only mean that *some* concerns are better decomposed by using other techniques.

6.3 Experience of AOP

AOP advocates claims that the size of the source program will be reduced significantly [6]. We think that this depends on the type of concerns in the system. The appended JavaBean example, does not differ that much compared to the plain old Java solution. The reason for this is that most concerns in the example are class affected concerns (see chapter 2). However, the size of the AOP program is unlikely to grow bigger than the OO implementation.

AspectJ is described as "a simple extension to Java" [1]. We feel that this is slightly misleading since AspectJ introduces a large set of new language constructs and uses of several new concepts. Learning AspectJ is not easy, and we believe that AOP researchers must communicate design guidelines and educate the developer community in AOP techniques.

In the end, it is the developers that will settle if AOP is a success or not.

With AspectJ, all special concerns can be encapsulated as aspects, but we have found that these aspects are often hard to reuse in other context, due to the fact that they operate on a specific set of basic concerns (components).

7 Conclusions

In this thesis we have summarized the current state of the AOP research and the theories that AOP is based on. We have also introduced the java language extension, AspectJ that adds AOP capabilities to Java by introducing a set of new language constructs. AOP theories have been explained by program examples written both in Java and AspectJ, to allow a fair comparison between the two paradigms OO and AOP.

From our comparison we could not see major differences in program size, we belive that this depends on the type of concerns in the program. Some concerns may be implemented as reusable aspects (and hence reduce program size) while other concerns are specific for a component. The AOP implementation did successfully separate the concerns in the example program, i.e. no crosscutting code.

We believe that AOP suggest an interesting and general solution to the goals stated by SoC. Still, several problems must be solved before AOP can gain acceptance in the industry. To our knowledge, no larger software project uses AOP techniques today. Hence there is no real proof that the theories actually work. Only the future will tell if AOP is to be used widely.

References

[1] Eclipse aspectj project, http://eclipse.org/aspectj/. 2003.

[2] AlphaWorks at IBM. Hyperj, http://www.alphaworks.ibm.com/tech/hyperj. 2003.

[3] Eclipse. The aspectj programming guide, http://eclipse.org/aspectj/. 2003.

[4] Stefan Hanenberg and Rainer Unland. A proposal for classifying tangled code. 2002.

[5] Walter L. Hursch and Cristina Vidiera Lopes. Separation of concerns. 1995.

[6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. 1997.

[7] Stanley M. Sutton Peri Tarr Harold Ossher. William Harrison. N degrees of separation: Multi-dimensional separation of concerns. 1999.

[8] Niklas Påhlsson. Aspect-oriented programming.

[9] Martin P. Robillard. Separation of concerns and software components.

[10] Markus Voelter. Aspect-oriented programming in java http://www.voelter.de/data/articles/aop/aop.html. 2000.

A AspectJ primitive point cut designators

```
call(MethodPattern)
execution(MethodPattern)

get(FieldPattern)
set(FieldPattern)

call(ConstructorPattern)
execution(ConstructorPattern)
initialization(ConstructorPattern)
preinitialization(ConstructorPattern)

staticinitialization(TypePattern)

handler(TypePattern)

adviceexecution()

within(TypePattern)
withincode(MethodPattern)
withincode(ConstructorPattern)

cflow(Pointcut)
cflowbelow(Pointcut)

this(Type or Id)
target(Type or Id)
args(Type or Id, ...)

if(BooleanExpression)
```

B Terminology

This appendix contains definitions of some of the most important terms that relates to AOP.

Separation of Concern

Separation of Concerns is an important engineering principle. It refers to the ability to identify, encapsulate, and manipulate those part of software that are relevant to a particular concern (concept, goal, purpose, etc).[8]

Concern

A typical system consists of several concerns. In the simplest form there are the core concerns, i.e. the natural components of the software. Except for these core concerns, there are system level concerns, like security, logging, authentication, persistence and so on; concerns that tend to affect several other concerns.[8]

Code tangling

When concerns are tightly intermixed, code tangling occurs.[8]

Code scattering

When concerns are poorly localized this is called code scattering.[8]

Components

Components are properties of a system, for which the implementation can be cleanly encapsulated in a generalized procedure.

Aspect

An aspect, is by definition, modular units that cross-cuts the structure of other units. An aspect is similar to a class by having a type, it can extend classes and other aspects, it can be abstract or concrete and have fields, methods, and type as members. It can encapsulate behaviors that affect multiple classes into reusable modules.[8]

With respect to system development some decisions are difficult to cleanly capture in actual code. We call the issues these decisions address aspects.[6]

Join point

The joinpoint are well-defined points in the execution of a program like method calls, field access, conditional checks, loop beginnings, assignments and object constructions[8]

Join points are not necessarily explicit constructs in the component language. Rather like nodes in the dataflow graph and runtime method invocations they are clear, but perhaps implicit, elements of the component program's semantic.[6]

Point-cut

Pointcuts, or pointcut designators, are program constructs to designate joinpoints and collect specific context at those points. The criteria can be explicit function names or function names specified by wildcards.[3][8]

Advice

An advice is code that runs upon meeting certain conditions. In AspectJ there are three different advice; before advice, after advice and around advice.[8]

[3]AspectJ specific definition

Aspect Weaver

Aspect weavers must process the component and aspect languages, co-composing them properly to produce the desired total system operation. Essential to the the function of the aspect weaver is the concept of join points, which are those elements the component language semantics that the aspect programs coordinate with.[6]

C JavaBean example

```
package ex.pb;

import java.io.Serializable;

/**
 * Simple UserBean class. A user is defined by the uiser id (uid),
 * firstname and lastname. Each bean property has the standard
 * associated set and get methods.
 */
public class UserBean  implements Serializable {

    protected String uid;
    protected String firstname;
    protected String lastname;

    /**
     * Empty default constructor, according to the Java bean
     * specification.
     */
    public UserBean()  {

    }

    /**
     * Set/change the uid property.
     */
    public void setUid(String uid) {
        this.uid = uid;
    }

    /**
     * Returns the value of the uid property.
     */
    public String getUid() {
        return uid;
    }

    /**
     * Set/change the lastname property.
     */
    public void setLastname(String lastname) {
        this.lastname = lastname;
    }

    /**
     * Returns the value of the lastname property.
     */
    public String getLastname() {
        return lastname;
    }

    /**
     * Set/change the firstname property.
     */
    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    /**
     * Returns the value of the firstname property.
     */
    public String getFirstname() {
        return firstname;
    }


    /**
     * Returns a string representation of this bean.
     */
    public String toString() {
        return "UserBean: " + uid + " - " + firstname + " " + lastname;
    }
}
```

```
package ex.pb;

import java.io.Serializable;

/**
 * Simple DepartmentBean class. A department is defined by the
 * department id (did), name and admin. Each bean property has the
 * standard associated set and get methods.
 */
public class DepartmentBean  implements Serializable {

    protected String did;
    protected UserBean admin;
    protected String name;

    /**
     * Empty default constructor, according to the Java bean
     * specification.
     */
    public DepartmentBean()  {

    }

    /**
     * Set/change the did property.
     */
    public void setDid(String did) {
        this.did = did;
    }

    /**
     * Set/change the admin property.
     */
    public void setAdmin(UserBean admin) {
        this.admin = admin;
    }

    /**
     * Set/change the name property.
     */
    public void setName(String name) {
        this.name = name;
    }

    /**
     * Returns the value of the did property.
     */
    public String getDid() {
        return did;
    }

    /**
     * Returns the value of the admin property.
     */
    public UserBean getAdmin() {
        return admin;
    }

    /**
     * Returns the value of the name property.
     */
    public String getName() {
        return name;
    }

    /**
     * Returns a string representation of this bean.
     */
    public String toString() {
        return "DepartmentBean: "+ name+" "+did+" Admin: "+admin;
    }
}
```

```
package ex.pb;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.DriverManager;

/**
 * A simple database connection manager class, used to get and release
 * db connection objects. This implementation creates a new connection
 * object for each getConnection call.
 */
public class DB {

    private static String host = "jdbc:mysql://enterprise.cse.kau.se/";
    private static String db = "lab3_5";
    private static String user = "worf";
    private static String pwd = "medM768ge";


    /**
     * Returns a new db connection object.
     */
    public static Connection getConnection() throws SQLException {

        Connection singleCon = null;

        try {

            Class.forName("com.mysql.jdbc.Driver").newInstance();
            singleCon = DriverManager.getConnection(host+db, user, pwd);

        } catch (Exception e) {
            e.printStackTrace();
        }

        return singleCon;
    }


    /**
     * Cleans up the resource held by the argument connection
     * object. Every connection object fetched from the getConnection
     * method sould be release by this method.
     */
    public static void release(Connection con) throws SQLException {

        try {

            if (con != null && !con.isClosed()) {
                con.close();
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
package ex.napb;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class UserBeanDB extends UserBean {

    //SQL query to update the bean.
    private String updateQuery =
        "select * from UserBean where uid = ?";

    //SQL query to store the bean.
    private String storeQuery =
        "update UserBean set firstname = ? , lastname = ? where uid = ?";

    /**
     * Overrides the method in UserBean, and updates the bean from db.
     */
    public void setUid(String uid) {
        super.setUid(uid);
        updateUserBean();
    }

    /**
     * Overrides the method in UserBean.
     */
    public String getUid() {
        return super.getUid();
    }

    /**
     * Overrides the method in UserBean, and stores the bean from db.
     */
    public void setLastname(String lastname) {
        super.setLastname(lastname);
        storeUserBean();
    }

    /**
     * Overrides the method in UserBean.
     */
    public String getLastname() {
        return super.getLastname();
    }

    /**
     * Overrides the method in UserBean, and stores the bean from db.
```

```
 */
public void setFirstname(String firstname) {
    super.setFirstname(firstname);
    storeUserBean();
}

/**
 * Overrides the method in UserBean.
 */
public String getFirstname() {
    return super.getFirstname();
}

/**
 * Helper method that updates a user bean from the database.
 */
private void updateUserBean() {

    Connection con = null;

    try {
        con = DB.getConnection();
        PreparedStatement stmt = con.prepareStatement(updateQuery);
        stmt.setString(1, uid);
        ResultSet rs = stmt.executeQuery();

        if (rs.next()) {
            firstname = rs.getString("firstname");
            lastname = rs.getString("lastname");
        }

        if (stmt != null) {
            stmt.close();
        }

        if (rs != null) {
            rs.close();
        }

    } catch (SQLException sqle) {
        sqle.printStackTrace();
    } finally {
        if (con != null) {
            try {
                DB.release(con);
            } catch (SQLException sqle) {
                sqle.printStackTrace();
            }
        }
        }
    }

    /**
     * Helper method that stores a user bean to the database.
     */
    private void storeUserBean() {

        Connection con = null;

        try {
            con = DB.getConnection();

            PreparedStatement stmt = con.prepareStatement(updateQuery);
            stmt.setString(1, firstname);
            stmt.setString(2, lastname);
            stmt.setString(3, uid);

            stmt.execute();

            if (stmt != null) {
                stmt.close();

            }

        } catch (SQLException sqle) {
            sqle.printStackTrace();
        } finally {
            if (con != null) {
                try {
                    DB.release(con);
                } catch (SQLException sqle) {
                    sqle.printStackTrace();
                }
            }
        }
    }
}
```

```
package ex.napb;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class DepartmentBeanDB extends DepartmentBean {

    //SQL query to update the bean.
    private String updateQuery =
        "select * from DepartmentBean where did = ?";

    //SQL query to store the bean.
    private String storeQuery =
        "update DepartmentBean set admin = ? , name = ? where did = ?";

    /**
     * Overrides the method in DepartmentBean, and updates the bean from db.
     */
    public void setDid(String did) {
        super.setDid(did);
        updateDepartmentBean();
    }

    /**
     * Overrides the method in DepartmentBean.
     */
    public String getDid() {
        return super.getDid();
    }

    /**
     * Overrides the method in DepartmentBean, and stores the bean from db.
     */
    public void setName(String name) {
        super.setName(name);
        storeDepartmentBean();
    }

    /**
     * Overrides the method in DepartmentBean.
     */
    public String getName() {
        return super.getName();
    }

    /**
     * Overrides the method in DepartmentBean, and stores the bean from db.
     */
    public void setAdmin(UserBean admin) {
        super.setAdmin(admin);
        storeDepartmentBean();
    }

    /**
     * Overrides the method in DepartmentBean.
     */
    public UserBean getAdmin() {
        return super.getAdmin();
    }

    /**
     * Helper method that updates a department bean from the database.
     */
    private void updateDepartmentBean() {

        Connection con = null;

        try {
            con = DB.getConnection();

            PreparedStatement stmt = con.prepareStatement(updateQuery);
            stmt.setString(1, did);
            ResultSet rs = stmt.executeQuery();

            if (rs.next()) {
                admin = new UserBeanDB();
                admin.setUid(rs.getString("admin"));
                name = rs.getString("name");
            }

            if (stmt != null) {
                stmt.close();
            }

            if (rs != null) {
                rs.close();
            }

        } catch (SQLException sqle) {
            sqle.printStackTrace();
        } finally {
            if (con != null) {
                try {
                    DB.release(con);
                } catch (SQLException sqle) {
                    sqle.printStackTrace();
                }
            }
        }
    }

    /**
     * Helper method that stores a department bean to the database.
     */
    private void storeDepartmentBean() {

        Connection con = null;

        try {
            con = DB.getConnection();

            PreparedStatement stmt = con.prepareStatement(updateQuery);
            stmt.setString(1, admin.getUid());
            stmt.setString(2, name);
            stmt.setString(3, did);

            stmt.execute();

            if (stmt != null) {
                stmt.close();

            }

        } catch (SQLException sqle) {
            sqle.printStackTrace();
        } finally {
            if (con != null) {
                try {
                    DB.release(con);
                } catch (SQLException sqle) {
                    sqle.printStackTrace();
                }
            }
        }
    }
}
```

```
package ex.pb;

import java.sql.Connection;
import java.sql.SQLException;


/**
 * To handle the persistence of the JavaBeans, we first must have a
 * database connection. The allocation and deallocation of these
 * connection objects has been speparated into this abstract
 * aspect. Any other aspect that needs a connection should define the
 * dbAccess point cut.
 */
abstract aspect CMPAspect {

    //holds the current connection objecct, its validity is determined
    //by the dbAccess point cut.
    protected Connection con;

    /**
     * Abstract point cut that should be defined by sub-aspects.
     */
    pointcut dbAccess();

    /**
     * The advice that guarantees that the join point has access to a
     * existing jdbc connection.
     */
    void around():dbAccess() {

        try {

            //get a connection
            con = DB.getConnection();

            //continue processing the source method.
            proceed();

            //clean up
            DB.release(con);

        } catch (SQLException sqle) {
            sqle.printStackTrace();
        } finally {
            if (con != null) {
                try {
                    DB.release(con);
                } catch (SQLException sqle) {
                    sqle.printStackTrace();
                    }
                }
            }
        }
    }
```

```
package ex.pb;

import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.PreparedStatement;

/**
 * This aspects handles the persistence of a DepartmentBean object
 * (constant cruss cutting concern)x. A DepartmentBean is uniquely
 * defined by its did. Whenever a DepartmentBean sets its did propery
 * this aspect willl try to load the remaining bean properties from
 * the database. When the other properties change, this aspect will
 * write the new values to the database.
 */
privileged aspect CMPDepartmentBean extends CMPAspect {

    private String storeQuery =
        "update DepartmentBean set name = ? , admin = ? where did = ?";

    private String updateQuery =
        "select * from DepartmentBean as D where D.did = ?";


    // define method that should have database access.
    pointcut dbAccess():
        call(void DepartmentBean.set*(String));


    //update advice, picks up on each setDid() execution.
    after(DepartmentBean dept, String did) :
        target(dept) &&
        args(did) &&
        execution(public void DepartmentBean.setDid(String)) {

        updateDepartmentBean(dept);
    }

    //store advice, picks up on each setName() or setAdmin()
    //execution.
    after(DepartmentBean dept, String uid) :
        target(dept) &&
        args(uid) &&
        execution(public void DepartmentBean.setName(String)) ||
        execution(public void DepartmentBean.setAdmin(UserBean)) {

        storeDepartmentBean(dept);
    }
```

```
/**
 * Helper method that updates a department bean from the database.
 */
private void updateDepartmentBean(DepartmentBean dept) {

    try {

        PreparedStatement stmt = con.prepareStatement(updateQuery);
        //stmt.setString(1, dept.getName());
        //stmt.setString(2, dept.getAdmin().getUid());
        stmt.setString(1, dept.getDid());

        ResultSet rs = stmt.executeQuery();

        while (rs.next()) {
            dept.name = rs.getString("name");
            UserBean admin = new UserBean();
            admin.setUid(rs.getString("admin"));
            dept.admin = admin;
        }

        if (stmt != null) {
            stmt.close();
        }

        if (rs != null) {
            rs.close();
        }

    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
}

/**
 * Helper method that stores a department bean to the database.
 */
private void storeDepartmentBean(DepartmentBean dept) {

    try {

        PreparedStatement stmt = con.prepareStatement(storeQuery);
        stmt.setString(1, dept.getName());
        stmt.setString(2, dept.getAdmin().getUid());
        stmt.setString(3, dept.getDid());

        stmt.execute();
```

```
            if (stmt != null) {
                stmt.close();
            }

        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
    }

}
```

```
package ex.pb;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * This aspects handles the persistence of a UserBean object (constant
 * cruss cutting concern). A UserBean is uniquely defined by its
 * uid. Whenever a UserBean sets its uid propery this aspect will try
 * to load the remaining bean properties from the database. When the
 * other properties change, this aspect will write the new values to
 * the database.
 */
privileged aspect CMPUserBean extends CMPAspect {

    //SQL query to update the bean.
    private String updateQuery =
        "select * from UserBean where uid = ?";

    //SQL query to store the bean.
    private String storeQuery =
        "update UserBean set firstname = ? , lastname = ? where uid = ?";


    // define method that should have database access.
    pointcut dbAccess():
        call(void UserBean.set*(String));

    //update advice, picks up on each setUid() execution.
    after(UserBean user, String uid) :
        target(user) &&
        args(uid) &&
        execution(public void UserBean.setUid(String)) {

        updateUserBean(user);
    }

    //store advice, picks up on each setFirstname() or setLastname
    //execution.
    after(UserBean user, String uid) :
        target(user) &&
        args(uid) &&
        execution(public void UserBean.setFirstname(String)) ||
        execution(public void UserBean.setLastname(String)) {


        storeUserBean(user);
    }
```

```
/**
 * Helper method that updates a user bean from the database.
 */
private void updateUserBean(UserBean user) {

    try {


        PreparedStatement stmt = con.prepareStatement(updateQuery);
        stmt.setString(1, user.uid);
        ResultSet rs = stmt.executeQuery();

        if (rs.next()) {
            user.firstname = rs.getString("firstname");
            user.lastname = rs.getString("lastname");
        }

        if (stmt != null) {
            stmt.close();
        }

        if (rs != null) {
            rs.close();
        }

    } catch (SQLException sqle) {
        sqle.printStackTrace();
    }
}

/**
 * Helper method that stores a user bean to the database.
 */
private void storeUserBean(UserBean user) {

    try {

        PreparedStatement stmt = con.prepareStatement(updateQuery);
        stmt.setString(1, user.firstname);
        stmt.setString(2, user.lastname);
        stmt.setString(3, user.uid);

        stmt.execute();

        if (stmt != null) {
            stmt.close();
        }
        } catch (SQLException sqle) {
            sqle.printStackTrace();
        }
    }

}
```

```
package ex.napb;

public class Test {

    public static void main(String[] a) {

        UserBean user =  new UserBeanDB();
        user.setUid("foo");
        user.getUid();

        DepartmentBean dept = new DepartmentBeanDB();
        dept.setDid("HR");

        System.err.println("UserBean: " + user);
        System.err.println("DepartmentBean: " + dept);
    }

}
```

```
package ex.pb;

public class Test {

    public static void main(String[] a) {

        UserBean user = new UserBean();
        user.setUid("foo");
        user.getUid();

        DepartmentBean dept = new DepartmentBean();
        dept.setDid("HR");


        dept.setName("KAU");

        System.err.println("UserBean: " + user);
        System.err.println("Dept: " + dept);
    }

}
```