Cork Institute of Technology

Department of Computing, Cork, Ireland

Design patterns in Aspect Oriented Programming

Patterns of enterprise application architecture in Aspect Oriented Programming

by

Bc. Michal Bali

Research project, M.Sc. in Software Development

Supervisor: Dr. John Creagh Date: August 2006

Abstract

Author: Bc. Michal Bali

Title: Patterns of enterprise application architecture in Aspect Oriented Programming

Design patterns offer flexible solutions to common software problems encountered in the design and construction of software systems. Their ordinary use, composition often leads to code duplication and unnecessary complexity. It is difficult to track, modularize and reuse these elements as they tend to vanish in the code.

Aspect oriented implementations of the patterns show modularity and other improvements. In this work we have analysed some of design patterns for enterprise application architecture, identified by Martin Fowler. We experiment aspect oriented programming as a modular technology to give new insights on the expression of design patterns. These implementations have properties like better code locality, reusability, transparent composability and (un)pluggability. The degree of improvement varies, with the greatest improvement coming when the pattern solution structure involves crosscutting of some form.

Finally, we also provide recommendations for applying this process to other patterns. This can be helpful to improve the implementation for other object-oriented patterns.

Contents

Acknowledgements

v

The author wishes to express sincere appreciation to Dr. John Creagh for his assistance in the preparation of this manuscript. Thanks to the members of the school council for their valuable input.

Vocabulary

Advice	A piece of advice is code that is executed when a join point is reached.
AOP	Stands for Aspect-Oriented Programming, a term coined by Gregor Kiczales at PARC in 1996.
Aspect	Aspects are the unit of modularity for crosscutting concerns. They behave somewhat like Java classes, but may also include pointcuts, advice and inter-type declarations and other constructs.
Crosscutting Concerns	Aspects of a program that crosscut the program design, i.e. applying a crosscutting concern to a program decreases the program modularity by either scattered or tangled code.
Inter-Type Declaration	Inter-Type declarations allow the programmer to modify a program's static structure, namely, the members of its classes and the relationship between classes.
Join Point	A well-defined point in the program flow.
Join Point Shadow	A join point shadow is the static projection of a join point onto the program code. Thus it is the code region which the existence of a join point depends on.
Mixin	A mixin encapsulates a piece of functionality that is “mixed into” existing classes without the use of conventional inheritance. In an AOP context, mixins are achieved through introductions. Mixins can be used to simulate multiple inheritance in Java.
OOP	Object Oriented Programming.
Orthogonal Concerns	Orthogonal concerns are separated from the base program in that the base program is oblivious to the concern's existence. This means that the concern can be added or removed without affecting the behavior of the base program. An example of this is logging. The addition or removal of a logging aspect does not affect the behavior of the program except to add logging functionality. Therefore the logging concern is orthogonal. However there are other ways that orthogonal concerns may indirectly affect program behavior. These include larger memory footprint, longer runtime, etc.

Pointcut	A Pointcut picks out certain join points and values at those points.
POJO	Plain Ordinary Java Objects. The term implies simplicity of an object as a component. It was coined by Martin Fowler, Rebecca Parsons and Josh MacKenzie. "We wondered why people were so against using regular objects in their systems and concluded that it was because simple objects lacked a fancy name. So we gave them one, and it's caught on very nicely." [28].
Scattering	The representation of a concern is scattered over an artifact if it is spread out rather than localized. Representations of concerns are tangled within an artifact if they are intermixed rather than separated. Scattering and tangling often go together, even though they are different concepts. Code scattering is caused because several instances of the patterns or of a given role are used within several classes of the program.
Tangling	See Scattering. Code tangling occurs when several pattern or role instances overlap in a single class.
Weaving	The process of coordinating aspects and non-aspects.

General Terms

Design pattern, Design, Languages.

Keywords

Design patterns, Aspect-oriented programming, and Enterprise application.

1 Introduction

Design patterns offer flexible solutions to common software development problems. A number of them involve crosscutting structures in the relationship between roles in the pattern and classes in each instance of the pattern.

I this work we will develop AspectJ implementations of design patterns and compare them to Java implementations. We will simply try to work out how the implementations of these patterns can be handled using a new implementation tool. This can be advantageous to other programmers.

For each OOP pattern we will analyze the pattern, multiple instances of the implementation. Then we will try to find out if it involves crosscutting of some form. If we will succeed, we will generate new design pattern implementation using AspectJ. We will create an example, where we will show the usage and benefits. We will summarize the advantages/disadvantages.

1.1 Design patterns

What is a design pattern? There is no exact definition of a design pattern. Probably the most appropriate definition is: “Design pattern is a general repeatable solution to a commonly-occurring problem in software design”. It is not finished design that can be directly transformed into code. It is a description or template for how to solve a problem that can be used in many different situations [27].

The benefits that design patterns provide as stated in [12]:

- They encapsulate experience.
- They provide a common vocabulary for computer scientists across domain barriers.
- They enhance the documentation of software design.

Patterns originated as an architectural concept by Christopher Alexander.

1.2 Aspect Oriented Programming

Here we will shortly describe aspect-oriented programming (AOP). AOP attempts to aid programmers in the separation of concerns, or the breaking down of a program into distinct parts that overlap in functionality as little as possible. In particular, AOP focuses on the modularization and encapsulation of cross-cutting concerns.

AOP decomposes systems into aspects/concerns, rather than objects. This is a different way of thinking about application structure than OOP, which considers objects hierarchically. OOP is a successful paradigm but there are areas where OOP isn't the best solution which can result in "the ultimate code smell" – code duplication. It is a sign that something is very wrong with implementation or design. Inheritance helps us leverage shared behavior. Polymorphism enables us to treat objects of different classes consistently when we're interested in their common characteristics. But there are some cases when the OOP solution is clumsy e.g. logging, security.

AOP gives us a way to separate the code for essential crosscutting concerns, such as logging, security, transaction handling and others, from program's core application logic cleanly. AOP can make the code more readable, less error-prone, and easier to maintain. AOP provides a powerful way of providing declarative enterprise services without use of any heavyweight framework or specification like EJB (Enterprise Java Beans). AOP can supplement/complement OOP where it is weak.

AspectJ is an implementation of aspect-oriented programming for Java. There are other implementations like Aspectwerkz, Spring-AOP, JBoss AOP, JAC, DynAOP, Nanning. AspectJ is the most complete aspect-oriented implementation.

1.3 AOP implementation strategies

The following are the main strategies used to implement AOP

- Java Code Generation – Old approach, java source code is generated. This code includes code to execute crosscutting code.
- Use of a Custom Class Loader – By defining custom class loader, we can apply advice automatically when a class is loaded. This works even if we use `new`

operator to create new objects. This approach may be problematic in some application servers (e.g. we cannot change the class loader). Used in JBoss AOP, AspectWerkz[1].

- Dynamic Proxies – Most obvious strategy to implement AOP is to use standard Java dynamic proxies (uses Java reflection). However it is not possible to proxy classes. Used in Spring Framework, Nanning.
- Dynamic Byte Code Generation – Similar to dynamic proxies. It adds the ability to proxy classes.
- Language extensions – We can extend the language to support AOP. This approach is taken by AspectJ, which extends Java.

1.4 Established challenges

Lot of authors have pointed out problems that arise from using concrete instances of a pattern in a particular software system. The most important challenges are related to implementation, documentation, and composition [1].

Patterns can “disappear into code” and lose their modularity. This makes hard to distinguish between the pattern, concrete instance of the pattern and the object involved. Adding or removing pattern is often an invasive task. In many cases patters introduce code tangling and scattering. It becomes difficult, for other programmers, to trace particular instance of a pattern (usually involved in pattern composition, see [3], [15]). Even though the design pattern is made to be reusable, the concrete instances are usually not. This makes code refactoring more difficult.

On the other side, important thing to keep in mind is that aspect oriented programming languages violate Dijkstra's principle of a “coordinate system” for understanding the evaluation of variables. AOP is a powerful technique which can be easily misused. Our understanding is that AOP should not be used for developing business logic, but its use for implementing design patterns is in our opinion reasonable.

[1] AspectWerkz is now part of AspectJ.

1.5 POJO

POJO is an acronym for Plain Old Java Object, and is favoured by advocates of the idea that the simpler the design, the better. We believe that the key strength to for writing reliable software lies in simplicity. Here are the advantages of POJOs:

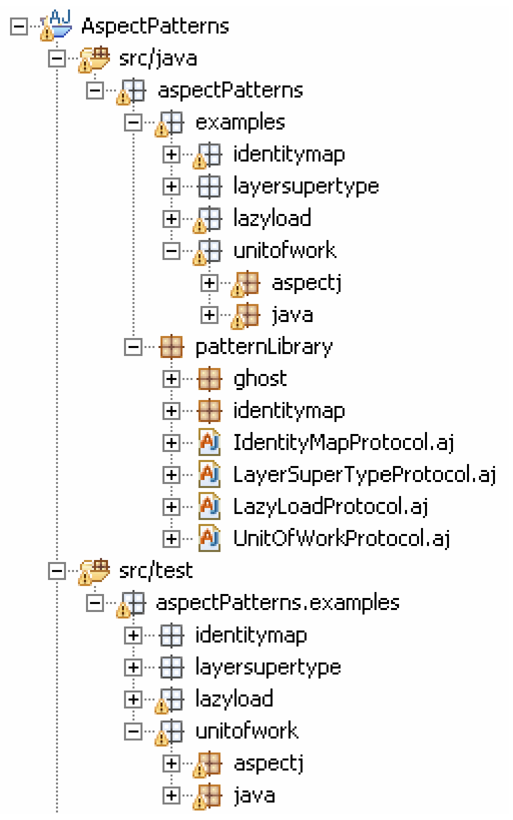
- Ease of testing, e.g. you don’t have to set-up whole environment to run tests (you can test your business logic outside of the application server and without a database), it reduces testing time (time that your tests are running), and code that needs to be written, so your are more likely to write those tests,
- you are not tying your code to particular API or framework, you can potentially reuse your code in different environments e.g. if you are running in an application server you can generate a thin EJB layer which will wrap your POJOs, but in the same time you can have you POJOs deployed in a servlet container, which is not so heavyweight,
- the simplicity enables you to focus on the important parts of the application,
- dependency inversion pattern can be applied more easily,
- POJOs simplify development, because you are not forced to think about everything at the beginning (business logic, persistence, transactions, etc.), instead you can implement the business logic and then you can deal with persistence, transactions.

With AOP, the use of POJOs is much easier as will be shown later.

1.6 Environment

We will implement the OOP and AOP solutions using Java 5 and AspectJ. Because Java is multiplatform language and AspectJ is an extension to Java, the target platform can be almost any platform (including Microsoft Windows, UNIX, Linux, Mac OS and so on). The code fragments shown by Martin Fowler were written using Java 1.4 or below. We will use features like generics, annotations, enhanced for-loop and others to simplify the readability of the source code. As for integrated development environment we will use Eclipse with AspectJ plugin – AJDT. We will also use the AspectJ Visualiser plugin to visualise the effects of our aspects on the source code.

For analysis and design will be used standard UML. We will use implementation structure which has been used by Hammerman [1]. Hammerman has created separate package structure for aspects that can be used as a library and separate package where he put all java and aspectj examples. The same structure for tests will be used.



```
AspectPatterns
    src/java
        aspectPatterns
            examples
                identitymap
                layersupertype
                lazyload
                unitofwork
                    aspectj
                    java
            patternLibrary
                ghost
                identitymap
                IdentityMapProtocol.aj
                LayerSuperTypeProtocol.aj
                LazyLoadProtocol.aj
                UnitOfWorkProtocol.aj
    src/test
        aspectPatterns.examples
            identitymap
            layersupertype
            lazyload
            unitofwork
                aspectj
                java
```

Figure 1. Project package structure.

Example implementations will be tested with JUnit and JMock. A sample test can be seen in Appendix C – UnitOfWork unit tests. Tests are very useful when the program needs to be refactored. Good written tests should fail if the invariants are violated[2].

[2] While we were creating java and aspectj implementations (examples and tests), we have tried not to duplicate the code. This seems to be challenging task because AspectJ modifies the bytecode of existing classes so it will automatically affect examples in java. Possible solution is to somehow turn off all aspects while running java examples.

2 Background Research

Here we will present related work that has been done in this area related work that we have come across and patterns that we want to analyze.

2.1 Gang-of-Four design patterns

Authors of this work show, that aspect based implementations of the Gang-of-Four (GoF) design patterns [21] showed modularity improvements in 17 of 23 cases. Results – 74% of GoF patterns implemented in a more modular way, and 52% reusable. Authors showed improvements in terms of better code locality, reusability, composability and (un)pluggability.

Authors found that the most important challenges are related to implementation documentation and composition. They stated that the impacts of design patterns on programs are of two different natures. They can:

- Superimpose roles – An initial functional class could be enhanced to define a role in the design pattern, an example is the Observer pattern which introduces roles Subject and Observer, this was discussed in more detail in [4].
- Define new roles – New classes are added to the program that are independent from the initial functional program, but this newly added role has to be used eventually by a class of the functional program, an example is the State pattern which introduces State.

Authors claim that the degree of improvement in implementation modularity varies, with the greatest improvement coming when the pattern solution structure involves crosscutting of some form, including one object playing multiple roles, many objects playing one role, or an object laying roles in multiple pattern instances.

They found that patterns with crosscutting structure between roles and participant classes see the most improvement. This improvement comes mainly from modularizing the implementation of the pattern. So the implementation is textually localized. This is achieved by removing code-level dependencies from the participant classes to the implementation of the pattern.

Authors developed the implementations iteratively. The AspectJ construct allowed them a number of implementations, usually with varying tradeoffs. They claim that they ended up creating a total of 57 different implementations, which ranged from 1 to 7 pre pattern. Finally they compared the Java and AspectJ implementations of concrete instances of the GoF design patterns. They created an AspectJ pattern library. We would like to use the same approach in this work.

They achieved three kinds of improvements:

- The general improvements observed in many pattern re-implementations.
- The specific improvements associated with particular patterns.
- The origins of crosscutting structure in patterns, and a demonstration that observed improvements correlate with the presence of crosscutting structure in the pattern.

Authors claim that implementations of 17 of the 23 GoF patterns were localized, for 12 of these the core part of the implementation was abstracted into reusable code. In 14 of the 17 they observed transparent composability of pattern instances, so that multiple patterns can have shared participants. The improvements in the AspectJ implementations are primarily due to inverting dependencies, so that pattern code depends on participants, not the other way round (all dependencies between patterns and participants are localized in the pattern code).

They also observed another improvement that all code related to a particular pattern instance is contained in a single module (which defines participants, assigns roles, etc.).

In 12 cases they developed reusable patterns by generalizing the roles, pattern code, communication protocols and relevant conceptual operations in an abstract reusable aspect. When the developer wants to use a concrete instance, he just defines the participants (assigns roles), and fills in instance specific code.

They also solved common problem with having multiple instances of a design pattern in one application. They have done this by reusing generalized pattern code a localizing the code for a particular pattern instance, so multiple instances of the same pattern in one application are not easily confused (composition transparency). The same participating object or class can even assume different roles in different instances of the same pattern.

They specific improvements are in the Singleton case, then with multiple inheritance in Java and they managed to break cyclic dependencies (e.g. Mediator pattern).

See [1] for more information.

However, other authors [11] have pointed out that attaining reusability is hard even with aspect-oriented programming. They detected problems and limitations in some of the AspectJ implementations. They showed 3 causes of these difficulties:

1. Difficulties in obtaining adequate joinpoints to weave the desired extra behavior in the intended point of the program,
2. difficulties in obtaining joinpoints exposing the context required by the aspect at the appropriate moments,
3. difficulties for aspects to quantify over objects without violating encapsulation.

Authors of this paper have asked interesting question: "Should abstract classes be considered bad style in the context of AspectJ?" This represents interesting mechanism for implementing mixins in java.

2.2 J2EE Business Tier Design Patterns

This work analyzes the problem of crosscutting within the implementation of J2EE patterns in the Enterprise Java Beans (EJB). J2EE patterns that were presented in book "Core J2EE patterns: Best Practices and Design Strategies".

Authors claim that non-modularization in the business layer of the J2EE applications due to patterns introduces:

- Code scattering – Caused because several instances of the patterns or of a given role will be used within several classes of the program. This relates to code locality.

- Code tangling – Occurs when several pattern or role instances overlap in a single class. This relates to pattern composability.

They noticed great improvements in those patterns where a single module of abstraction handles the original behaviour and the pattern specific behavior. The J2EE pattern implementations are more localized and reusable and hence the system is more adaptable.

See [10] for more information.

2.3 Types of Design patterns

It is important to classify design patterns, because this can affect their potential to benefit from aspect-oriented implementation.

Agebro [12] has pointed out that one person’s design pattern can be another person’s primitive building block, because the point of view affects one’s interpretation of what is and what is nor a design pattern. The point of view is influenced by the choice of programming language e.g. if one assume procedural language one might have include design patterns called inheritance, encapsulation or polymorphism. Authors have agreed that design patterns do not need to be language independent. They formed 3 guidelines:

1. Design patterns covered by language constructs are not Fundamental Design Patterns.
2. Applications and variations of design patterns are not Fundamental Design Patterns.
3. A design pattern may not be an inherent object oriented way of thinking.

Based on these guidelines they divided design pattern in two groups:

- Fundamental design patterns (FDPs) – these are not covered by any language construct.
- Language dependent design patterns (LDDPs) – these have different implementations depending on the language used.

2.4 Java enhancements constructs

Java language does not directly support multiple inheritance. It is only simulated by using interfaces. However while reusing the interface we have still code duplication. Even if we use composition to avoid code duplication we need to delegate the calls. Aspect-oriented approach can provide nicer way of solving this issue.

As was presented in some papers [5], [11], [19], we can use mixins to support multiple inheritance. Though, the basic problem one has to face when working with the multiple inheritance is the diamond problem. An inheritor AB multiple inherits from classes A and B which in turn inherit from a common ancestor O. The problems are:

- Encapsulation of inheritance – is it allowed for a client or inheritor of a class to depend on the inheritance structure of that class?
- Common ancestor duplication problem – must the common ancestor be duplicated or not?
- Common ancestor name conflict problem – name collisions that arise when attributes of a common ancestor are inherited along different paths.
- Duplicate parent operation invocation problem – multiple inheriting from two classes that both invoke the same parent method on a common ancestor can cause duplicate invocations of this parent operation.

2.5 Other work

Pattern composition has been shown as a challenge for applying design patterns [8]. Martin Fowler [17] introduced 5 patterns to deal with different requirements imposed on subjects in different contexts. Patterns:

1. Single Role Type – combine all features of the role into a single type
2. Separate Role Type – threat each role as a separate type
3. Role Subtype – common behavior in supertype, have subtype for each role
4. Role Object – common features on a host object with a separate object for each role. Clients ask the host object for the appropriate role to use a role’s features
5. Role Relationship – role is a relationship with an appropriate object (example – employee in a group).

This patterns show an object-oriented attempt to deal with superimposed roles. However each has some drawback, for example the Role Object pattern creates cyclic references: `ComponentCore` stores a list of roles, and each `ComponentRole` has reference to the core object they are attached to. While this enables dynamically adding and removing roles from an object this approach creates tight coupling between core and role.

Gray et al. [7] has showed how aspect-oriented programming can help to handle crosscutting constraints in domain-specific modelling. They have proposed a different type of weaver from those others have constructed in the past (e.g. the weaver for AspectJ). Normal weaver process source code, but the proposed weaver works with the structured textual descriptions of a model. They give example of embedded systems where constraints often have contradictory goals (for example latency and resource usage). By using aspects they can isolate these concerns and this allows them to study the effects of constraints across the entire system. This improves the modular understanding of the effect of each constraint. They can create various “what if” scenarios, because they can easily plug/unplug various set of aspects into the model. See also [6].

AspectJ already offers support for specifying constraints on the object-oriented application to be woven by declaring errors which refer to pointcuts. However these pointcuts are related to object-oriented features in Java but not aspect-oriented features in AspectJ. Hanenberg et. al. [20] has pointed out that it is not possible to specify constraints on the usage of the new composition mechanisms – pointcut, advice and introduction. They have presented a tool which permits to specify design constraints on the usage of the aspect-oriented features of AspectJ (this helps to check if certain design constraints are obeyed in a given application or not).

Nicolas Lesiecki – Author of AOP@Work series [26] has wrote an article about testing crosscutting behavior implemented with aspects - aspect unit testing. Testing of aspects has many same practical and design benefits as testing objects. He recommends breaking the behavior into components that one can test independently. The crosscutting concerns should be divided into two areas:

1. Crosscutting specification – what parts of the program the concern affects?

2. Functionality – what happens at those points?

In pure object world these two areas intertwine as the concern tangles itself through the application. However by using aspects one can target one or both of these areas in isolation. He also recommends moving the advice body into an independently testable class so one can analyze the behavior without necessarily needing to understand the way it crosscuts the application. This also increases the flexibility and pluggability of the system as a whole.

Code scattering and tangling reduces reusability, extensibility, and traceability of code. Another paper discusses the removal of these properties [16]. Scattering and tangling exist both in designs and code and must therefore be addressed in both. Authors suggest separating designs and code of crosscutting behaviour into independent models or programs. They present an investigation into a means to maintain separation of crosscutting behaviour seamlessly across the lifecycle. To achieve this, they work with composition patterns at the design level, AspectJ and Hyper/J at the code level, and investigate a mapping between the two levels. Since a composition pattern encapsulates details within it, these details can be altered while the concrete classes bound to the CP remain untouched.

Many in the aspect-oriented community are nowadays working to extend the concepts of interface-based design from the object world to the emerging aspect/object world. Interfaces define appropriate abstractions for coupling components without exposing too many details. Hence, interfaces tend to be more stable than the underlying components as the software evolves. Recent summary of promising work on aspect interfaces can be found in [18]. If we compare aspect and object worlds, by their nature aspects touch the system pervasively whereas object components are more localized. Aspects can be used to modify object state and behavior in more fine-grained ways raising worries about maintaining system integrity, robustness, and even comprehension. To address these issues many thinks that interfaces should not only contain the methods and state information, but also contractual information that constrains the modifications aspects are allowed to make to other components. So, instead of having pointcuts coupling directly to component details, like particular naming conventions, aspects will couple to interfaces that are implemented by the components. The interfaces have to be designed to expose allowed join points and state information. The aspects will advise components only through the exposed interfaces. The coupling will be more stable and resilient to change as the system evolves because interfaces tend to be more stable than the components that implement them, achieving easier to build robust, non-trivial, yet reusable aspect/object systems.

2.6 Analyzed patterns

Results of work that has been done in this area suggest that it would be worthwhile to undertake the experiments of applying AspectJ to more patterns and/or applying other aspect-oriented techniques to pattern implementations.

In our study we would like to analyze patterns presented by Martin Fowler in book: Patterns of Enterprise Application Architecture [22]. The author of this book noticed that despite changes in technology, from Smalltalk to CORBA to Java to .NET, the same basic design ideas can be adapted and applied to solve common problems. With the help of an expert group of contributors, author distils over forty recurring solutions into patterns that are applicable to any enterprise application platform.

Patterns are divided to multiple sections focused on enterprise application development.

We have studied these patterns and we have tried to identify those sections that would benefit from aspect oriented programming at most. Another important thing to consider is – how often are the pattern used. We have decided to analyze these sections:

- Base Patterns – these patterns are dealing with the basic problems in enterprise applications.
- Object-Relational Behavioral Patterns – patterns for simplifying the transition from relational world to object world, relational databases are still dominant in enterprise applications.
- Domain Logic Patterns – they are dealing with the domain itself, patterns like Domain Model, Service Layer. These patterns are heavily used in enterprise applications.

We have omitted some sections like Distribution patterns or Offline Concurrency patterns because they already have aspect-oriented solutions found, see [9], [13], [14] for distributed patterns and [2], [9] for concurrency patterns.

It would be ideal to analyze all patterns that were presented in this book but because of the time constraints we have limited our studies only to those areas.

To our knowledge to date we haven't found any similar work that is dealing with this type of patterns.

3 Object-Relational Behavioral Patterns

In the sections that will follow, we will present our findings. We provide a short summary of the existing object-oriented pattern with UML diagrams. We create an example that will use this object-oriented pattern, with the focus on the shortcomings of the implementation. Then we show our proposed aspect oriented version of the pattern - library aspect. Finally we summarize the advantages/disadvantages.

3.1 Lazy Load – Lazy initialization

Lazy load is a design pattern, which describes an object that doesn't contain all of the data that you need but knows how to get it. With lazy loading you can save memory resources when necessary by loading and instantiating an object at the point in time when it is absolutely necessary.

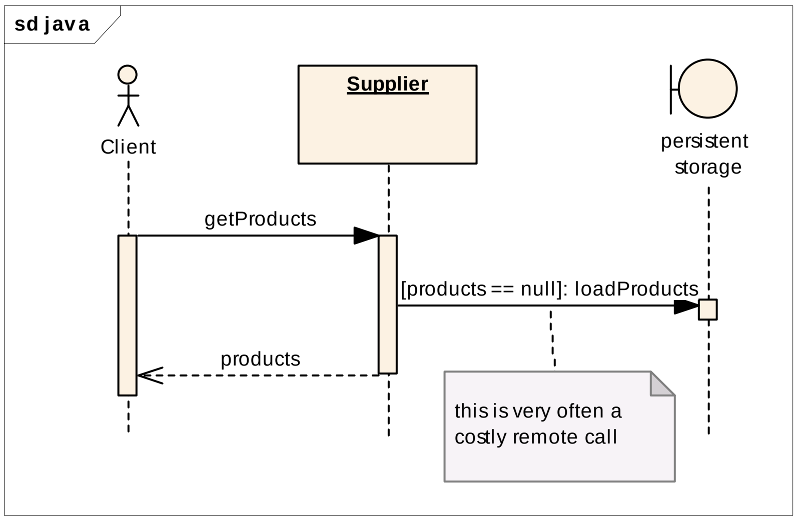

sd java
Client
Supplier
persistent storage
getProducts
[products == null]: loadProducts
products
this is very often a costly remote call

Figure 2. Example lazy loading sequence diagram.

This approach has also the advantages of reducing persistent storage accesses in complex object graphs. Disadvantages are ripple loading. For more information see [22].

3.1.1 OOP solution

```
public class Supplier {
  private List<Product> products;
  /**
   * this impementation is not synchroznized
   *
   * @return products for this supplier
   */
  public List<Product> getProducts() {
    //check if products are already loaded, if not lazy-load them
    if (products == null) {
      products = Product.findForSupplier(this);
    }
    return products;
  }

  public List<Product> getProductsSale() {
    List<Product> productsSale = new ArrayList<Product>();
    for (Product product : products) {
      if (product.isOnSale()) {
        productsSale.add(product);
      }
    }
    return productsSale;
  }
}
```

Code 1. Example OO Supplier implementation (the arrow in the picture shows the crosscutting concern).

The OO solution shown in Code 1 has some drawbacks:

- The lazy loading behavior is coded directly into target class, one can argue that this shouldn't be like this; you should be able to create `Supplier` with or without lazy loading.
- The code for lazy loading is spread throughout the application, i.e. we have to code the lazy loading pattern to each class that wants to use this type of behavior.
- The lazy loaded field needs to be self-encapsulated, e.g. all access to `products` field needs to be done through `getProducts` method.
- It forces the dependency between the object and the database.

Note:

Probably the biggest advantage of using virtual proxy (another type of lazy loading) is that it frees the domain developer from lazy loading. However you need to have virtual implementations of objects that you want to lazy load, which can be quite complex.
Value holder pattern (yet another type of lazy loading) tries to simplify this at the cost of introducing other boilerplate code.
Instead of using these two patterns one can consider to use AOP implementation of lazy initialization.

3.1.2 AOP solution

By putting lazy-load behaviour into a separate aspect allows us to change the lazy loading strategy separately. It will also free the domain developers from having to deal with lazy loading issues.

```
public aspect SupplierLazyLoad extends LazyLoadProtocol {

  protected pointcut requestTriggered():
    //execution(public List Supplier.getProducts());
    get(List Supplier.products);

  /**
   * Extends the requestTriggered() pointcut to
   * include a field for the joinpoint. Used
   * internally only.
   */
  private pointcut accessBySupplier(Supplier supplier):
     requestTriggered()
     && this(supplier);

  before(Supplier supplier): accessBySupplier(supplier) {
    if (supplier.products == null) {
      supplier.products = Product.findForSupplier(supplier);
    }
  }

}
```
Code 2. Example lazy loading aspect.

In the listing above we are declaring `requestTriggered` pointcut which captures all joinpoints for accessing Supplier’s product list. This aspect extends from abstract `LazyLoadProtocol`, which only declares abstract pointcut `requestTriggered`. All the logic needs to be implemented in subclass aspect. There are many possibilities how can this be implemented so we didn’t tried to abstract this into the protocol.

```
  public List<Product> getProducts() {
    return products;
  }
```
Code 3. AOP implementation of Supplier's getProducts method.

As can be seen from Code 3, the implementation of the `getProducts` method is unaware of the lazy loading.

Advantages:

- Automatically enforces the self-encapsulation of fields (because we are using getter pointcut to capture the joinpoints),
- locality: the lazy loading behavior is separated from the object itself, it is in one place,
- (un)pluggability: one can disable the lazy loading behavior (then the data must be loaded during instantiation of the object, this can be done nicely if this pattern is used with the dependency inversion pattern).

3.2 Lazy load – Ghost

A ghost is the real object in a partial state (it contains just its id). If this object is accessed it loads its full state. It has fine granularity of loading – there is no need to load all the data in one go. You may group it in groups that are commonly used together. Also, commonly used / quick to get data can be loaded when the ghost is created. For more information see [22].

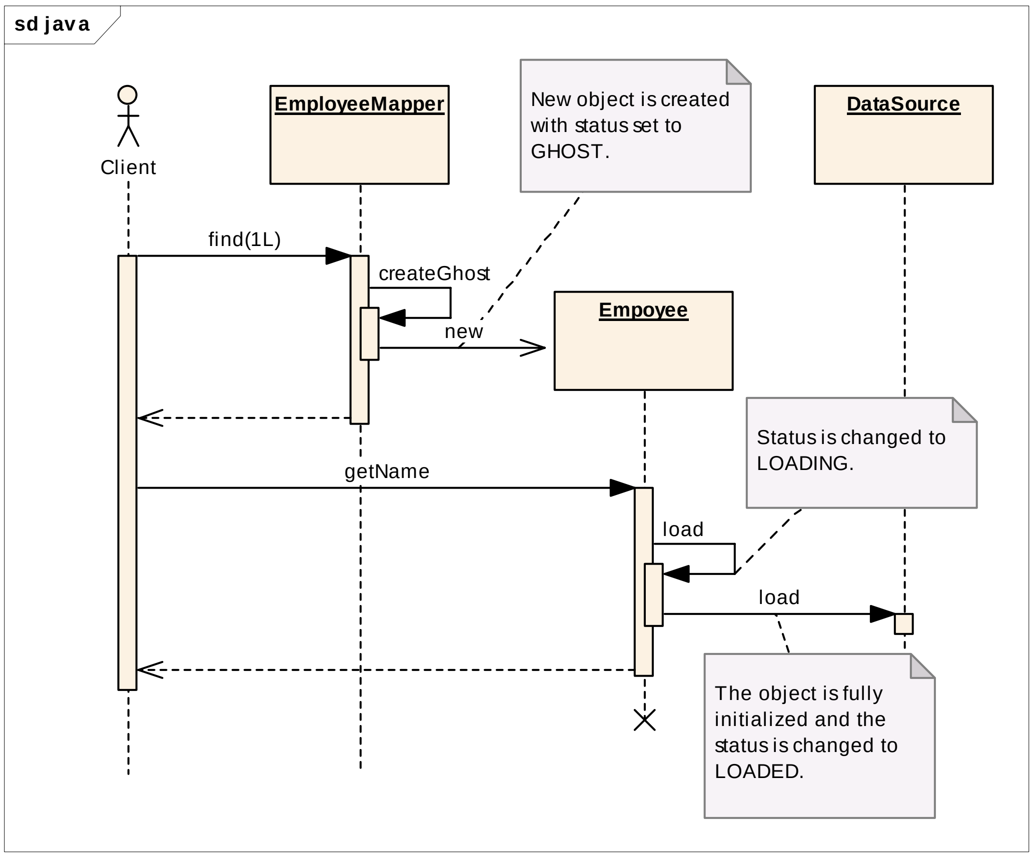

sd java
Client
EmployeeMapper
New object is created with status set to GHOST.
DataSource
find(1L)
createGhost
new
Empoyee
Status is changed to LOADING.
getName
load
load
The object is fully initialized and the status is changed to LOADED.

Figure 3. Ghost sequence diagram - domain layer.

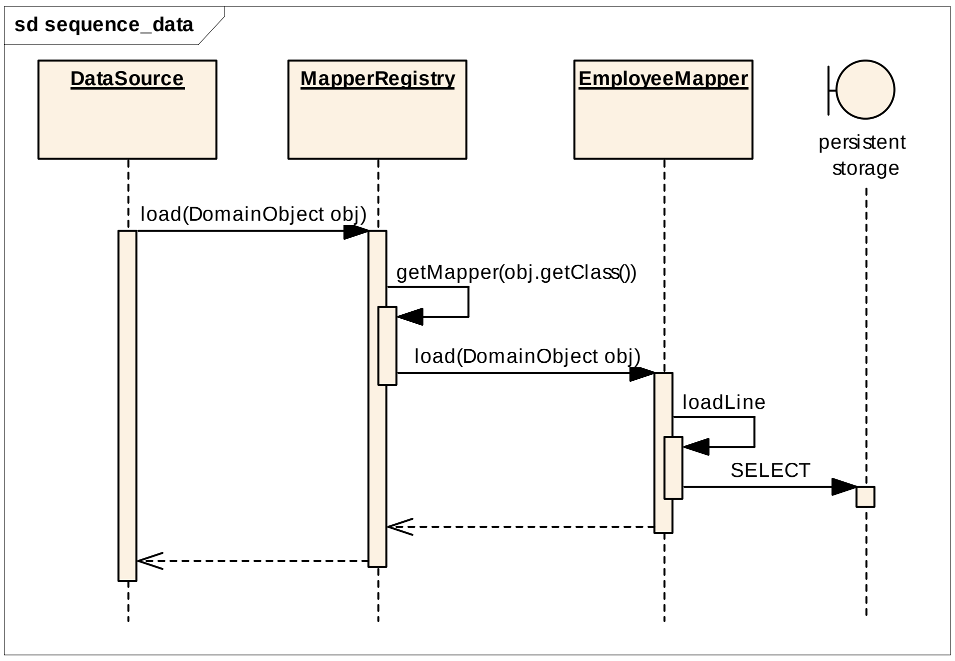


Figure 4. Ghost sequence diagram - data layer.

3.2.1 OOP solution

We will describe the ghost on a real-world example. We are using 2 layers: domain and persistence layer. Domain model consists of employee which has one or more time-records and our employee is in some department.

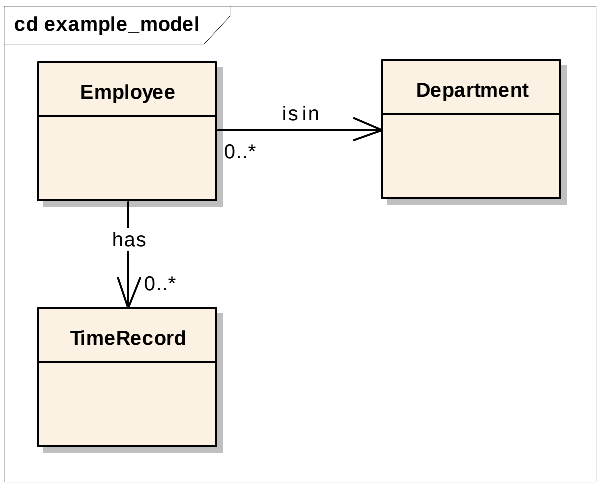


Figure 5. Simple example domain model.

We have chosen this example so we can describe how ghost handles value objects, object references and collections of other objects.

Persistence layer has dependency on domain layer. Class diagram of our example is shown below.

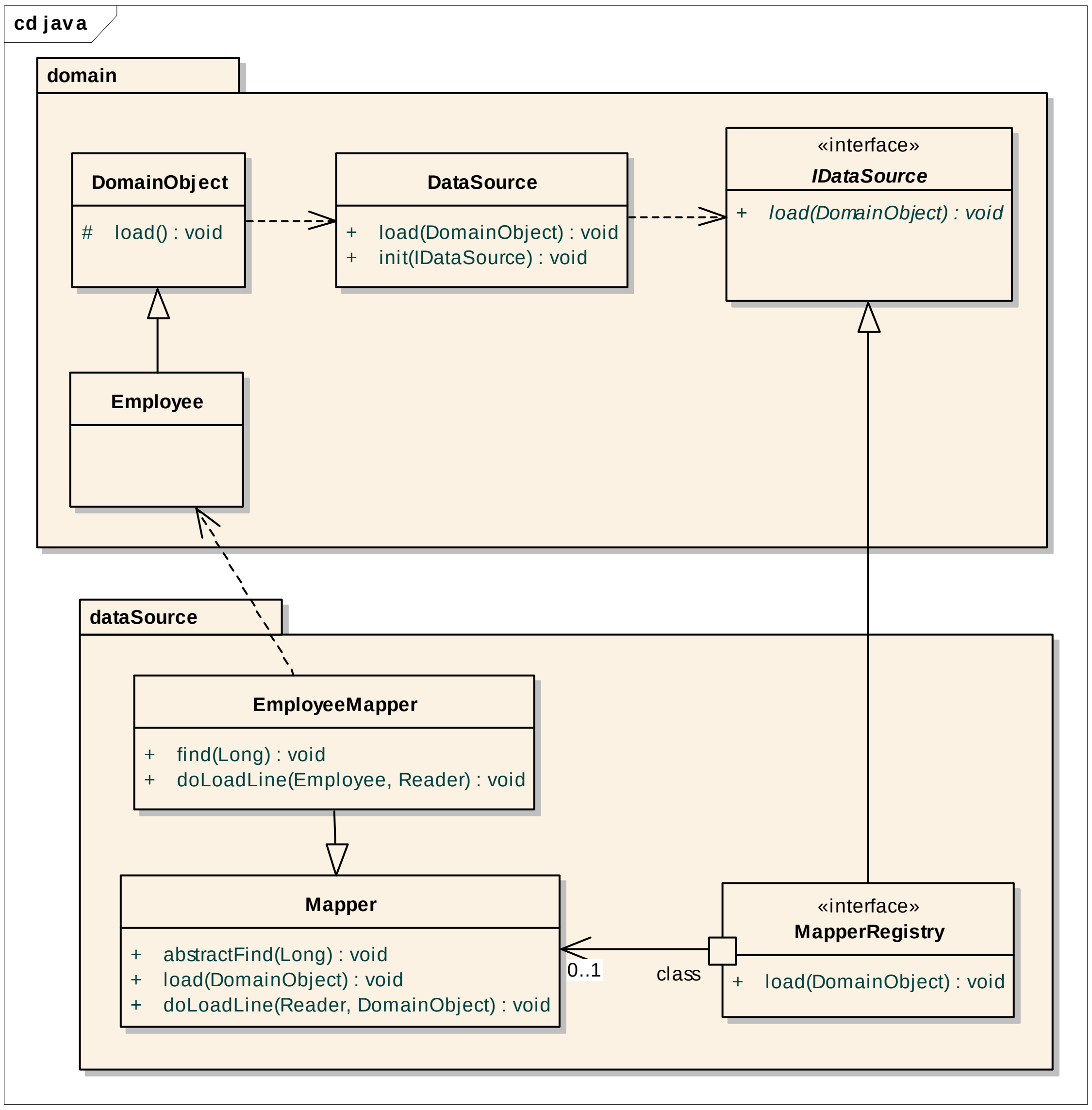

cd java
domain
DomainObject
# load() : void
DataSource
+ load(DomainObject) : void
+ init(IDataSource) : void
«interface»
IDataSource
+ load(DomainObject) : void
Employee
dataSource
EmployeeMapper
+ find(Long) : void
+ doLoadLine(Employee, Reader) : void
Mapper
+ abstractFind(Long) : void
+ load(DomainObject) : void
+ doLoadLine(Reader, DomainObject) : void
0..1
class
«interface»
MapperRegistry
+ load(DomainObject) : void

Figure 6. Class diagram – example that uses ghost design pattern.

Implementation of Employee object follows:

```
public class Employee extends DomainObject {
  // example of a value object
  private String name;
  // example of a reference
  private Department department;
  // example of a collection
  private DomainList<TimeRecord> timeRecords;
  public Employee() {
    // just for testing
  }
  public Employee(Long key) {
    super(key);
  }
  public String getName() {
    load();
    return name;
  }
  public void setName(String name) {
    load();                                  ←
    this.name = name;
  }
  public Department getDepartment() {
    load();                                  ←
    return department;
  }
  public void setDepartment(Department department) {
    load();                                  ←
    this.department = department;
  }
  public DomainList<TimeRecord> getTimeRecords() {
    load();                                  ←
    return timeRecords;
  }
  public void setTimeRecords(DomainList<TimeRecord> timeRecords) {
    load();                                  ←
    this.timeRecords = timeRecords;
  }
}
```

Code 4. Ghost OO implementation of Employee object (arrows show crosscutting concerns).

As can be seen from Code 4, the most intrusive element of ghosts is that every accessor needs to be modified so that it will trigger a `load` if the object is a ghost. It has big disadvantage that the domain is aware of lazy loading.

3.2.2 AOP solution

With this AOP solution the domain layer will be unaware of lazy loading. AOP implementation of `Employee` object is shown below.

```
public class Employee {
  // example of a value object
  private String name;
  // example of a reference
  private Department department;
  // example of a collection
  private List<TimeRecord> timeRecords;
  public Employee() {
    // just for testing
  }
  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public Department getDepartment() {
    return department;
  }
  public void setDepartment(Department department) {
    this.department = department;
  }
  public List<TimeRecord> getTimeRecords() {
    return timeRecords;
  }
  public void setTimeRecords(List<TimeRecord> timeRecords) {
    this.timeRecords = timeRecords;
  }
}
```

Code 5. POJO Employee object.

We can see that the `load` method is not present at all. By using this solution is natural to use also the dependency enforcement aspects i.e. we can easily say that our domain logic code should never depend on our data layer code. Also our Employee object doesn't extend anything.

```
public abstract aspect GhostLazyLoadProtocol extends LazyLoadProtocol
{
  // BEGIN Ghost inter-type ************************
  private LoadStatus Ghost.status;
  private Long Ghost.key;
  public void Ghost.load() {
    if (isGhost()) {
      GhostLazyLoadProtocol.getDataSource().load(this);
    }
  }
  public boolean Ghost.isGhost() {
    return status == LoadStatus.GHOST;
  }
  public boolean Ghost.isLoaded() {
    return status == LoadStatus.LOADED;
  }
  public void Ghost.markLoading() {
    assert isGhost();
    status = LoadStatus.LOADING;
  }
  public void Ghost.markLoaded() {
    assert status == LoadStatus.LOADING;
    status = LoadStatus.LOADED;
  }
  public Long Ghost.getKey() {
    return key;
  }
  public void Ghost.setKey(Long key) {
    this.key = key;
    this.status = LoadStatus.GHOST;
  }
  //END Ghost inter-type ************************
  //BEGIN DataSource ***********************************
  protected static IDataSource dataSource;
  public static IDataSource getDataSource() {
    return dataSource;
  }
  public static void setDataSource(IDataSource dataSource) {
    GhostLazyLoadProtocol.dataSource = dataSource;
  }
  //END DataSource ************************************
  /**
   * captures all methods executed on classes that
   * extend Ghost
   */
  protected pointcut requestTriggered():
    execution(public * Ghost+.*(..)) &&
    !execution(public * Ghost.*(..)) &&
    !execution(public * DomainList.getLoader(..)) &&
    !execution(public * DomainList.setLoader(..)) &&
    !cflow(adviceexecution());
  //divide this pointcut into accepted join points and exception
joinpoints
  /**
   * Extends the requestTriggered() pointcut to
   * include a field for the joinpoint. Used
   * internally only.
   */
  private pointcut accessByDomainObject(Ghost obj):
     requestTriggered()
     && this(obj);
  before(Ghost obj): accessByDomainObject(obj) {
    //System.out.println("in advice - going to execute obj.load()");
    obj.load();
  }

}
```
Code 6. GhostLazyLoadProtocol abstract aspect.

`GhostLazyLoadProtocol` provides default implementation for `DomainObject` interface (this is an example of mixin based inheritance). It also holds a reference to data source object. Finally it implements the abstract pointcut, which will trigger on each method execution of any object that implements `DomainObject` unless already executing. It also declares before advice which will simply call the actual `load` method on the domain object.

In order to create a true POJO from domain objects we have put every method from `DomainObject` to `Ghost` interface. One can however argue that a `key` is sometimes part of the domain. We can use this approach, too. The lazy loading behaviour is not part of the domain so I have extracted it. This is more natural.

Concrete implementation of `GhostLazyLoadProtocol` may look as follows:

```
public aspect AppGhostLazyLoad extends GhostLazyLoadProtocol {
  declare parents: Employee || Department || TimeRecord
    implements Ghost;
}
```

Code 7. Concrete implementation of GhostLazyLoadProcotol. This is the only part the developer needs to code.

As we can see, we only need to specify which domain model objects will be lazy loaded.

Advantages

- Use of POJOs,
- removed the boilerplate code – i.e. calling load() in each method,
- locality - it will be applied automatically on each newly added object, the load invocation is in one place,
- domain objects don't have to extend anything,
- domain layer is completely unaware of lazy loading,
- partial (un)pluggability, (our data layer is still aware of lazy loading, so if we want to remove/add lazy loading behavior we will have to change the data layer),
- fine tuned granularity in the sense that we can specify what objects will be loaded in one go, we can specify this for each type of objects independently (feature of ghost itself),
- reusability: `GhostLazyLoadProtocol` abstract aspect can be reused.

3.3 Unit of Work

Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems [22]. When your work is done it figures out everything that needs to be done to alter the database as a result of your work.

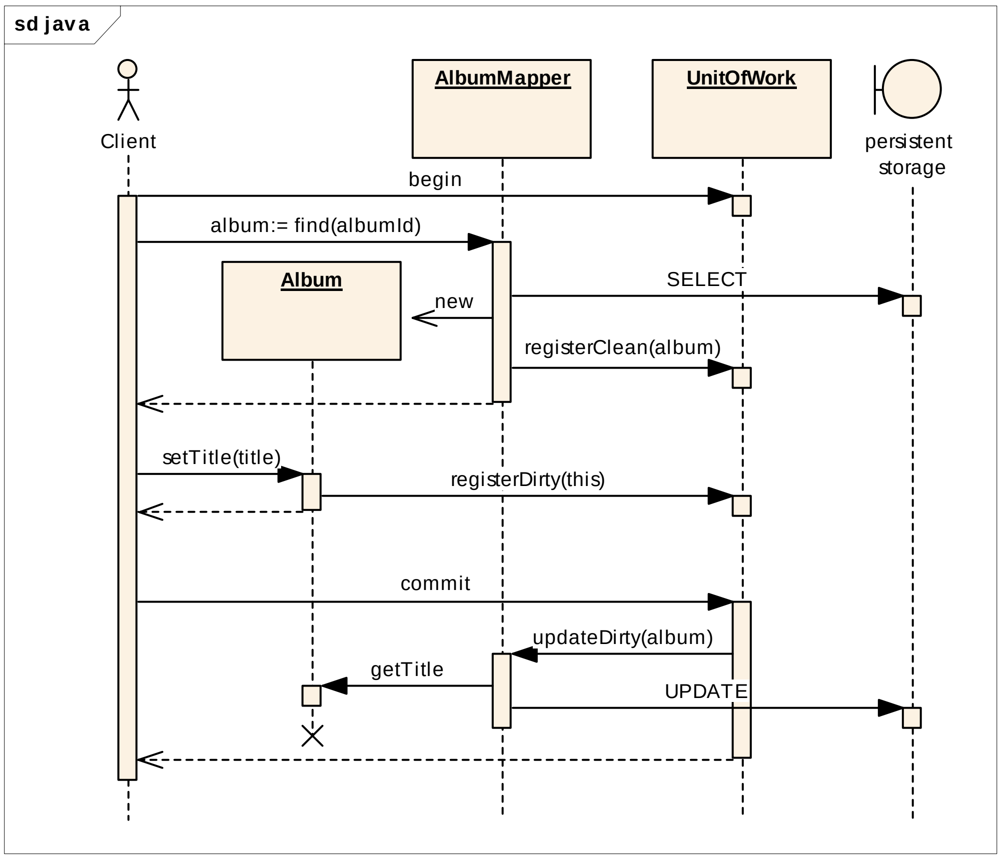


Figure 7. Unit of Work, example sequence diagram. Client is changing title of an album.

In the example above the client was creating `UnitOfWork`, however this is not usually the case. `UnitOfWork` is usually stored within a thread-scoped storage, so that the client is completely unaware of it. By using this pattern we can very easily batch updates.

3.3.1 OOP solution

```
public class Album extends DomainObject {
  private String title;
  private Date created;
  public Album() {
  }
  public Album(Long id, String name) {
    this.id = id;
    this.title = name;
  }
  public static Album create(String name) {
    Album obj = new Album(name);
    obj.markNew();
    return obj;
  }
  public void remove() {
    markRemoved();
  }
  public void setTitle(String title) {
    this.title = title;
    markDirty();          <-------------------
  }
  public String getTitle() {
    return title;
  }
  public Date getCreated() {
    return created;
  }
  public void setCreated(Date created) {
    this.created = created;
    markDirty();          <-------------------
  }
}
```
Code 8. Example OO implementation of Album domain object that uses UnitOfWork.

The OO implementation involves significant amount of crosscutting concerns. Basically there are 3-4 crosscutting areas in every domain object:

1. The remove method – should register the object as removed,
2. setter method- should register object as being dirty,
3. creating a new instance – should register object as new,
4. loading object from persistent storage (optional) – should register object as clean.

All of these crosscutting concerns are addressed in the AOP solution.

3.3.2 AOP solution

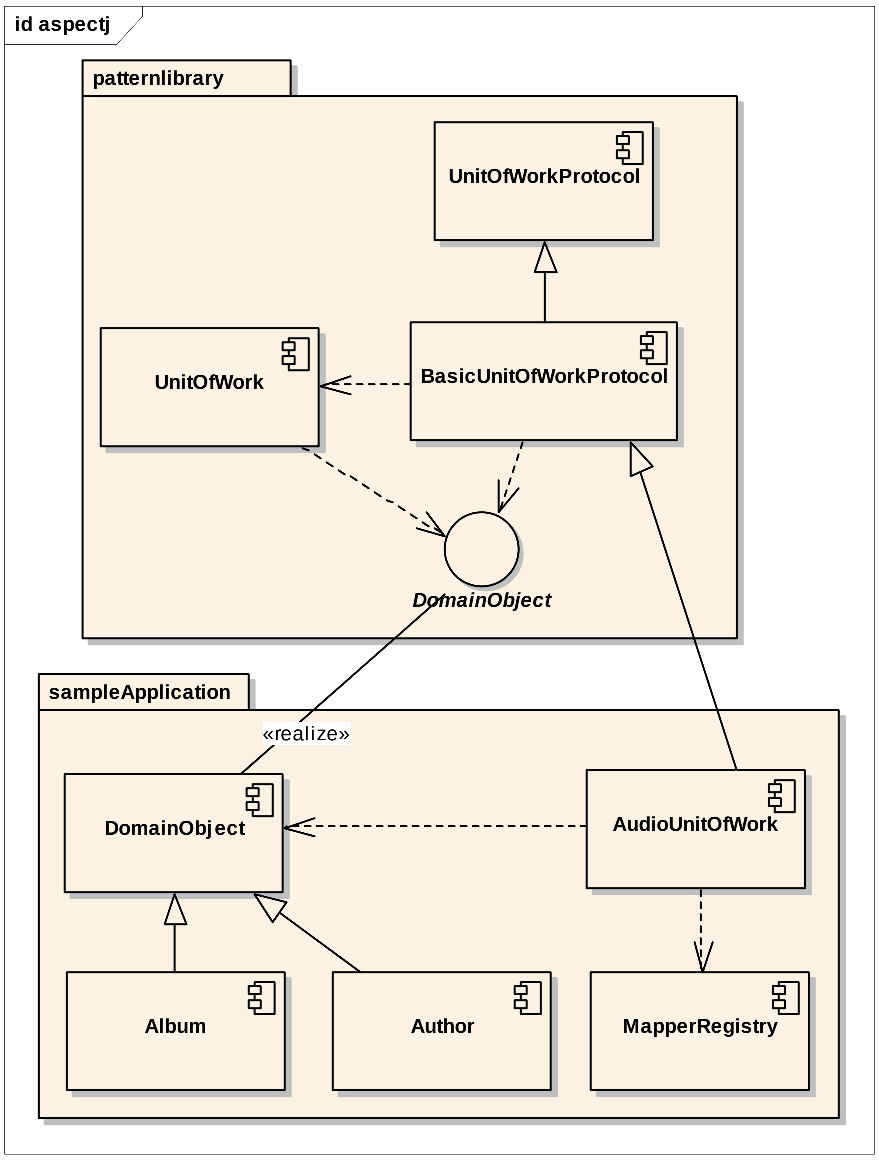


Figure 8. Sample instance of UniOfWork design pattern. This figure also demonstrates the reusability of the pattern (see the dependencies between components).

`UnitOfWorkProtocol` is a root abstract aspect used to declare pointcut names (abstract pointcuts). These pointcuts – `newObject, deleteObject, cleanObject, dirtyObject,` are there for capturing joinpoints when a new object is created, deleted, retrieved, modified respectively. `UnitOfWorkProtocol` also declares following four abstract methods `doInsertNew, doUpdateDirty, doDeleteRemoved`, each of them takes one object parameter. These methods are called when an object needs to be `inserted/updated/deleted` into/in/from persistent storage.

`BasicUnitOfWorkProtocol`, is an abstract aspect, which provides generic pointcut implementations and introduces an after advice for each pointcut. Each advice simply delegates work to thread-scoped `UnitOfWork` object.

```
public abstract aspect BasicUnitOfWorkProtocol extends
UnitOfWorkProtocol {

  protected abstract void setUp();

  protected pointcut dirtyObject():
    execution(* DomainObject+.set*(..)) &&
    !execution(* DomainObject.set*(..));
    //set(* Entity+.*);

  protected pointcut newObject():
    execution(DomainObject+.new(..)) &&
    cflowbelow(execution(* DomainObject+.create(..))) &&
    within(DomainObject+) &&
    !within(DomainObject);

  protected pointcut deleteObject():
    execution(* DomainObject+.remove(..)) ||
    execution(* DomainObject.remove(..));

  /*protected pointcut cleanObject():
    execution(DomainObject+.new(..)) &&
    !execution(DomainObject.new(..)) &&
    cflowbelow(execution(* Mapper+.find*(..)));*/

  after(DomainObject obj) returning():
    dirtyObject() && this(obj) {
    UnitOfWork.getCurrent().registerDirty(obj);
  }

  after(DomainObject obj) returning():
    newObject() && this(obj) {
    UnitOfWork.getCurrent().registerNew(obj);
  }

  after(DomainObject obj) returning():
    deleteObject() && this(obj) {
    UnitOfWork.getCurrent().registerRemoved(obj);
  }

  after(DomainObject obj) returning():
    cleanObject() && this(obj) {
    UnitOfWork.getCurrent().registerClean(obj);
  }
}
```

Code 9. BasicUnitOfWorkProtocol. Basic implementation of the UnitOfWorkProtocol.

As can be seen from Code 9 listing. The `BasicUnitOfWorkProtocol` introduces some naming conventions into the code:

- Each setter method should name should start with ‘`set`’,

- instantiation of new objects should be made in a '`create`' method inside a `DomainObject`,
- removal of an object should be placed inside '`remove`' method inside a `DomainObject`.

Each thread has independent `UnitOfWork` object. Part of the reusable library is also `DomainObject` interface. This is a marker interface, which also declares `id` field (this can be a primary key). Each object that has to be handled by the `UnitOfWork` needs to implement this interface.

```
public interface DomainObject {
  Long getId();
  void setId(Long id);
}
```

Code 10. DomainObject marker interface.

```
public class UnitOfWork {
  private List<DomainObject> newObjects = new 
ArrayList<DomainObject>();
  private List<DomainObject> dirtyObjects = new 
ArrayList<DomainObject>();
  private List<DomainObject> removedObjects = new 
ArrayList<DomainObject>();
  private static ThreadLocal<UnitOfWork> current = new 
ThreadLocal<UnitOfWork>();
  //current UnitOfWorkProtocol
  private static UnitOfWorkProtocol uowProtocol;
  public void registerNew(DomainObject obj) {
    assert obj.getId() != null : "id is null";
    assert !dirtyObjects.contains(obj) : "object is dirty";
    assert !removedObjects.contains(obj) : "object is removed";
    //assert !newObjects.contains(obj) : "object already registered 
new";
    if (!newObjects.contains(obj)) {
      newObjects.add(obj);
    }
  }
  public void registerDirty(DomainObject obj) {
    assert obj.getId() != null : "id is null";
    assert !removedObjects.contains(obj) : "object is removed";
    //check if this object has already been registred as dirty/new
    if (!dirtyObjects.contains(obj) && !newObjects.contains(obj)) {
      dirtyObjects.add(obj);
    }
  }
  public void registerRemoved(DomainObject obj) {
    assert obj.getId() != null : "id is null";
    //if this object is new -> just remove him from 'new' collection
    if (newObjects.remove(obj))
      return;
    dirtyObjects.remove(obj);
    if (!removedObjects.contains(obj)) {
      removedObjects.add(obj);
    }
  }
  public void registerClean(DomainObject obj) {
    assert obj.getId() != null : "id is null";
  }
  public void commit() {
    insertNew();
    updateDirty();
    deleteRemoved();
  }
  private void insertNew() {
    for (DomainObject obj : newObjects) {
      uowProtocol.doInsertNew(obj);
    }
  }
  private void updateDirty() {
    for (DomainObject obj : dirtyObjects) {
      uowProtocol.doUpdateDirty(obj);
    }
  }
  private void deleteRemoved() {
    for (DomainObject obj : removedObjects) {
      uowProtocol.doDeleteRemoved(obj);
    }
  }
  public static void newCurrent() {
    setCurrent(new UnitOfWork());
  }
  public static void setCurrent(UnitOfWork uow) {
    current.set(uow);
  }
  public static UnitOfWork getCurrent() {
    return current.get();
  }
  public static void setUowProtocol(UnitOfWorkProtocol uowProtocol) {
    UnitOfWork.uowProtocol = uowProtocol;
  }
}
```
Code 11. UnitOfWork class. Core UnitOfWork behavior.

`AudioUnitOfWork` is a concrete instance of this pattern. It uses inter-types to declare which objects implements `DomainObject` interface. Our example uses its own `DomainObject` so we must specified fully qualified name of it. `AudioUnitOfWork` also overrides `cleanObject` pointcut because it depends on the database layer (it refers to `Mapper` class).

```
public aspect AudioUnitOfWork extends BasicUnitOfWorkProtocol {

  declare parents : DomainObject implements
sk.misobali.aspectPatterns.patternLibrary.unitofwork.DomainObject;

  public void setUp() {
    UnitOfWork.setUowProtocol(AudioUnitOfWork.aspectOf());
  }

  /**
   * inserts objects into persistent storage
   * @param obj
   */
  public void doInsertNew(Object obj) {
    MapperRegistry.getMapper(obj.getClass())
      .insert((DomainObject) obj);
  }
  /**
   * updates objects in persitent storage
   * @param obj
   */
  public void doUpdateDirty(Object obj) {
    MapperRegistry.getMapper(obj.getClass())
      .update((DomainObject)obj);
  }
  /**
   * removes objects from persistens storage
   * @param obj
   */
  public void doDeleteRemoved(Object obj) {
    MapperRegistry.getMapper(obj.getClass())
      .delete((DomainObject)obj);
  }

  protected pointcut cleanObject():
    execution(DomainObject+.new(..)) &&
    !execution(DomainObject.new(..)) &&
    cflowbelow(execution(* Mapper+.find*(..)));

}
```

Code 12. Example use of UnitOfWork design pattern - AudioUnitOfWork.

After applying the `UnitOfWorkAspect` we get following simplified domain object:

```
public class Album extends DomainObject {
  private String title;
  private Date created;
  public Album() {
  }
  public Album(Long id, String name) {
    this.id = id;
    this.title = name;
  }
  public static Album create(String name) {
    Album obj = new Album(idGenerator++, name);
    return obj;
  }
  public void remove() {
    //can be overriden here
  }
  public void setTitle(String title) {
    this.title = title;
  }
  public String getTitle() {
    return title;
  }
  public Date getCreated() {
    return created;
  }
  public void setCreated(Date created) {
    this.created = created;
  }
}
```

Code 13. Example AOP implementation of Album domain object that is completely unaware of UnitOfWork.

We can clearly see that all the crosscutting code has gone.

The figure below shows the crosscutting structure in a graphical form. There are three bars, each represents a source file. Colored stripes represent lines of code affected by aspects. The length of each bar is proportional to the number of lines of code in file. We can see that the crosscutting is significant.

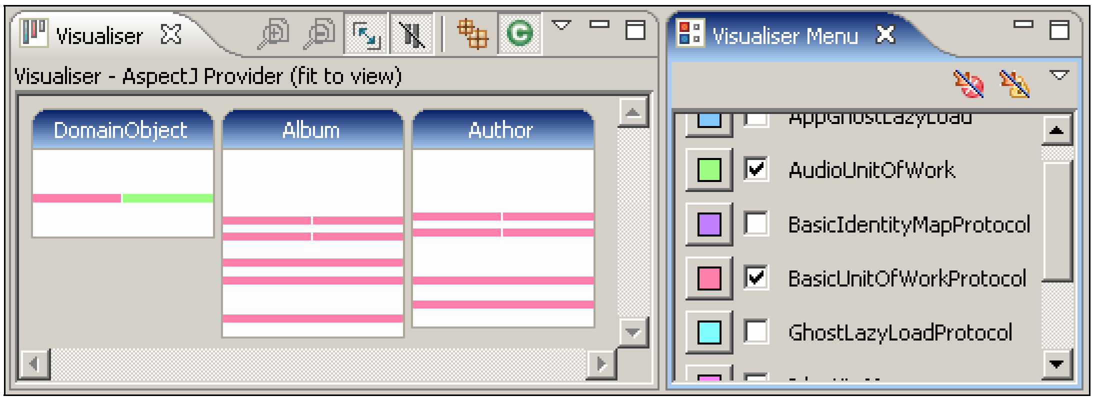


Figure 9. Visualisation of the crosscutting structure.

In the example shown I have left the primary key in the domain model as opposed to the previous pattern.

Advantages

- It is easier to place application logic into ‘mark dirty’ statements (e.g. in a setter - do not mark the object as dirty when the changed value is the same as the original),
- composition transparency – e.g. we can have more unit-of-work protocols applied at the same time, i.e. normal one and another one for auditing / history purposes,

- benefits of (un)pluggability, i.e. we can switch the history (see above) on or off.

3.4 Identity Map

Ensures that each object gets loaded only once by keeping objects in a map. It looks up objects using the map when referring to them [22].

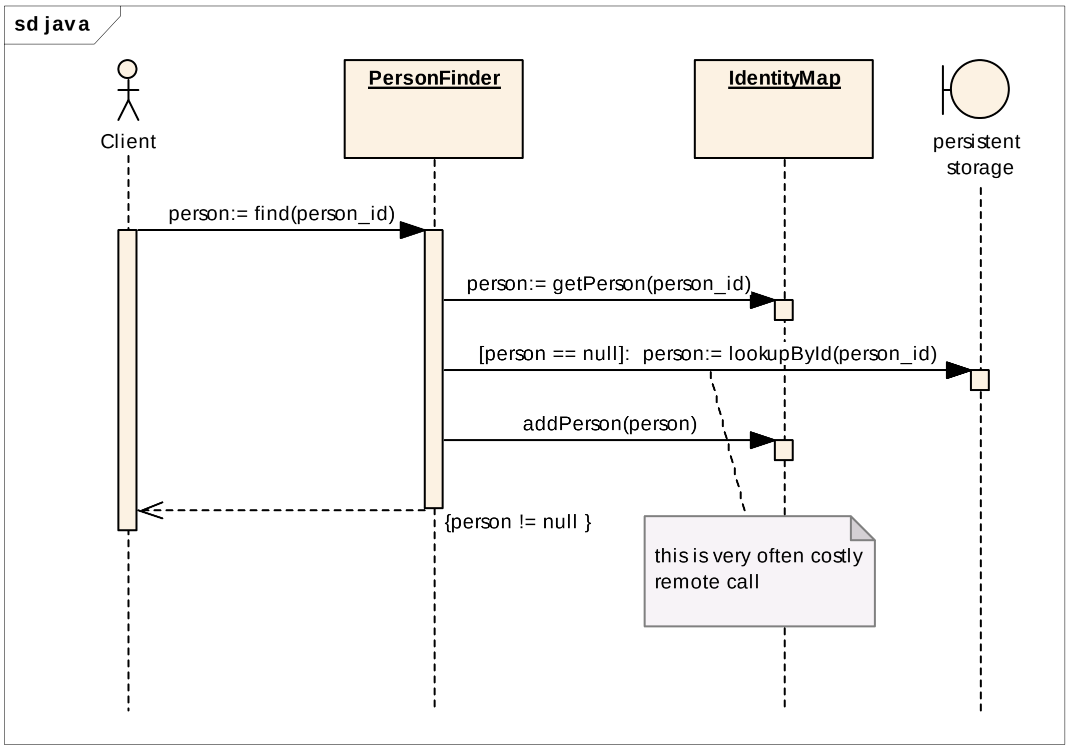

sd java
Client
PersonFinder
IdentityMap
persistent storage
person:= find(person_id)
person:= getPerson(person_id)
[person == null]: person:= lookupById(person_id)
addPerson(person)
{person != null }
this is very often costly remote call

Figure 10. Example identity map sequence diagram.

This design pattern has the following advantages:

- It reduces persistent storage access which increases performance,
- it saves memory by loading an object only once,
- it can prevent problems from loading the same object multiple times.

Disadvantages:

- You must be the only one who is accessing the database, because the identity map will get out-of-sync,
- each time you want an object, first you have to look into the identity map, i.e. you must be aware of this identity map and you must keep in mind to use it.

3.4.1 OOP solution

Now, we will look at one example that uses identity map design pattern. A `PersonFinder` class that is used to look up persons from persistent storage.

```
...
public static Person find(Long key) {
  //first look if this person has been already loaded
  Person person = IdentityMap.getPerson(key);
  if (person != null) {
    return person;
  }

  //load Person from database
  person = (Person)database.lookupById(key);

  //put this person into identity map
  IdentityMap.addPerson(person);

  return person;
}
...
```

Code 14. Code fragment from PersonFinder, which uses IdentityMap design pattern (arrows show crosscutting concerns).

The `PersonFinder` in the listing above involves two types of behavior. It interacts with the persistent storage and the identity map. Ideally, `PersonFinder` should be dealing only with persistent storage. The code dealing with identity map is similar for every finder object. It forms the crosscutting-concern of this pattern.

3.4.2 AOP solution

The `PersonFinder` with aspect-oriented version of the identity map may look as follows:

```
...
public static Person find(Long key) {
  //load Person from database
  Person person = (Person)database.lookupById(key);
  return person;
}
...
```

Code 15. Code fragment from PersonFinder which uses AOP version of IdentityMap design pattern.

All the crosscutting has gone. To cope with the problem we have created the following hierarchy of aspects. This three-level hierarchy is common to many our solutions. We have created it to support multiple solutions.

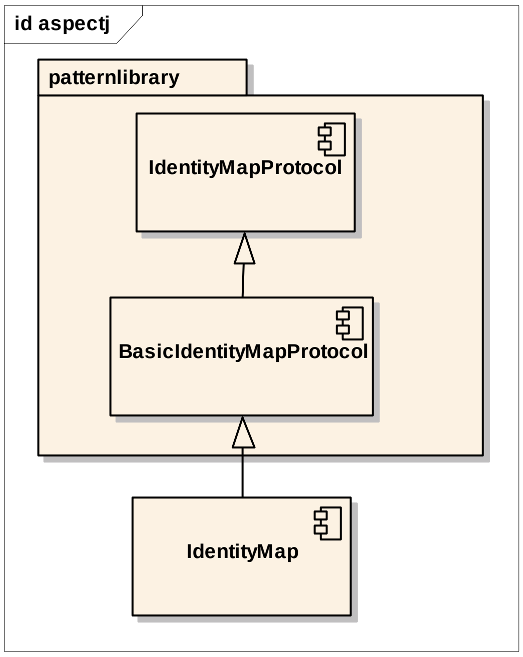


Figure 11. IdentityMap dependency diagram (AOP implementation).

Aspect `IdentityMapProtocol` just introduces abstract finder pointcut. It's more of an interface.

```
public abstract aspect IdentityMapProtocol {
  /**
   * Captures all accesses to finder methods
   */
  protected abstract pointcut finder(Long key);
}
```

Code 16. IdentityMap abstract aspect.

Aspect `BasicIdentityMapProtocol` is a concrete implementation. It introduces two marker interfaces `Entity` and `Finder`. It contains a map of loaded objects, implementation of finder pointcut and an around advice.

```
public abstract aspect BasicIdentityMapProtocol extends
IdentityMapProtocol {
  protected interface Entity {
    Long getId();
  }
  protected interface Finder { }
  private Map<Long, Entity> entities = new WeakHashMap<Long,
Entity>();
  public void addEntity(Entity entity) {
    entities.put(entity.getId(), entity);
  }
  public Entity getEntity(Long key) {
    return entities.get(key);
  }

  protected pointcut finder(Long key) :
    //execution(public static Entity+ Entity+.find*(Long)) &&
    execution(public static * Finder+.find*(Long)) &&
    args(key);

  Object around(Long key) : finder(key) {
    //first look if this entity has been already loaded
    Entity entity = getEntity(key);
    if (entity != null) {
      return entity;
    }

    //load entity
    entity = (Entity)proceed(key);

    //put this entity into identity map
    addEntity(entity);

    return entity;
  }
}
```
Code 17. Abstract implementation of IdentityMapProtocol.

The most interesting part of this implementation is the around advice which contains all of the identity map behavior. It uses finder pointcut to trigger its execution. This finder pointcut captures every execution of `public static` method of any subclass of `Finder` that starts with ‘`find`’ and takes one parameter of type `Long` and returns anything. The around advice then checks if the object, with the specified key, is already loaded in map. If it is loaded it returns the object, if not it proceeds with the loading and then puts the object into the map. Finally, it returns the object.

The implementation of `BasicIdentityMapProtocol` uses `WeakHashMap` to hold objects. We have chosen this implementation so the map doesn’t keep references to object that are no longer needed because an entry in a `WeakHashMap` will automatically be removed when its key is no longer in ordinary use. This prevents memory leaks.

This implementation of the pattern introduces some conventions:

- Naming conventions of the finder methods,

- each object we will store in identity map must have `Long getId();` method implemented. This method should return object's unique identifier.

The developer who wants to use this pattern must meet this conventions or he can create similar implementation of `IdentityMapProtocol` that suits his needs.

Concrete usage of `BasicIdentityMapProtocol` may look as follows:

```
public aspect IdentityMap extends BasicIdentityMapProtocol {
  declare parents : Person implements Entity;
  declare parents : PersonFinder implements Finder;
}
```

Code 18. Concrete implementation of BasicIdentityMapProtocol.

We only specify which our object plays role of an `Entity` and which of a `Finder`.

Properties of this implementation:

- Locality – All the code that implements the `IdentityMap` pattern is in the abstract and concrete `IdentityMap` aspects, none of it is in the participant classes. The participant classes are entirely free of the pattern context, and as a consequence there is no coupling between the participants. Potential changes to each `IdentityMap` pattern instance are confined to one place.
- Reusability – The core pattern code is abstracted and reusable. The implementation of `IdentityMapProtocol` is generalizing the overall pattern behavior. The abstract aspect can be reused and shared across multiple `IdentityMap` pattern instances. For each pattern instance, we only need to define one concrete aspect.
- Composition transparency – Because a pattern participant's implementation is not coupled to the pattern, if an Entity takes part in multiple identity map relationships their code does not become more complicated and the pattern instances are not confused. Each instance of the pattern can be reasoned about independently (i.e. we can have one entity in multiple identity maps)
- (Un)pluggability – Because Entities and Finders need not be aware of their role in any pattern instance, it is possible to switch between using a pattern and not using it in the system.

Note:

> While we were creating this implementation our main aim was to make it as lightweight as possible. There is no layering no domain model/objects, no Data Mappers.

4 Domain Logic Patterns

This category includes patterns: Transaction script, Domain model, Table module, Service layer. See [22] for more information.

We didn't find any crosscutting concerns in those patterns. The advantages of converting these patterns are minimal.

5 Base Patterns

5.1 Layer supertype

"A type that acts as the supertype for all types in its layer" [22]. It also helps in defeating code duplication. Common examples are identity fields for domain objects (then your Data Mappers relies on this fact), other examples include validation (business rules), journaling/auditing/history, clone functionality.

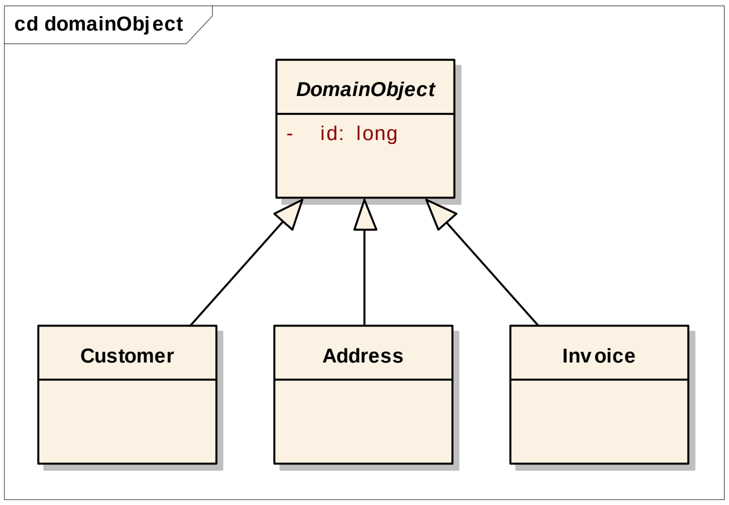


Figure 12. DomainObject Layer Supertype example.

5.1.1 OOP solution

If you are dealing with functionality that is common to most objects in a layer it may be straightforward to put it in one class and then extend all of your classes from this class. This works fine for one type of behavior (e.g. primary keys). As soon as you start adding more types of behaviors (e.g. journaling) it starts to get messy, this is because java supports only single inheritance.

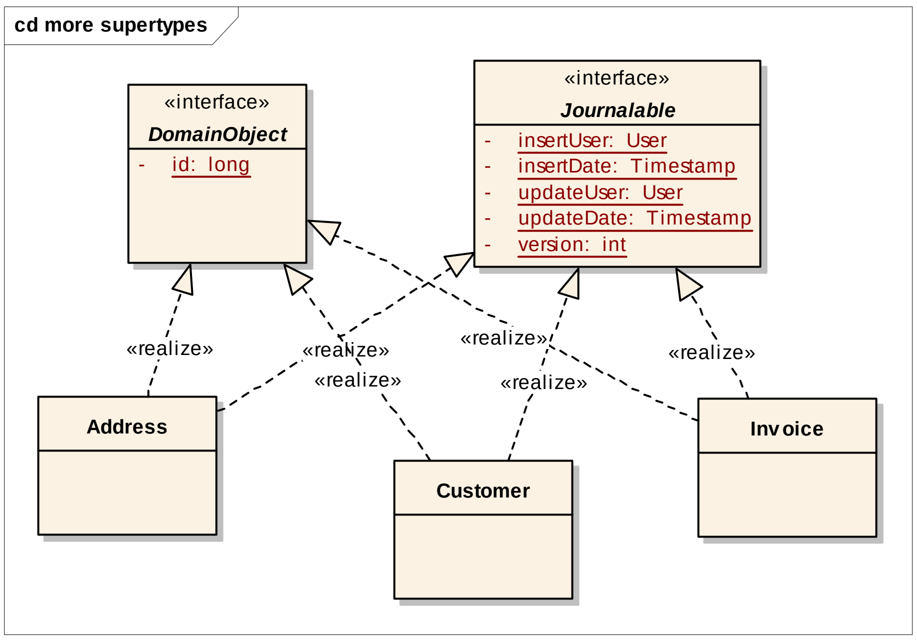


Figure 13. Domain model with two Layer Supertypes.

Possible approaches to solve this problem:

- Create multiple interfaces and implement them in one concrete class, you will end up with class like `JournalDomainObject`,
- create multiple implementations that extend themselves, this is not a good solution since you are using inheritance the wrong way – to simulate multiple inherintace.

It gets even worse when not all of your classes need all behaviors (e.g. you may have a Period object which is in some cases stored in it’s own table, in other cases stored in the table of the owning class)

```
public class Customer extends JournalDomainObject {
  ...
}
```

Code 19. Possible Customer implementation.

5.1.2 AOP solution

With AOP we can use mixins to enhance classes. Then we can have a `JournalAspect` to encapsulate all behavior dealing with journaling.

```
...
/* Implementation of LayerSuerType interface ****** */
private User Journalable.insertUser;
private Timestamp Journalable.insertDate;
private User Journalable.updateUser;
private Timestamp Journalable.updateDate;
public User Journalable.getInsertUser() {
  return insertUser;
}
public void Journalable.setInsertUser(User insertUser) {
  this.insertUser = insertUser;
}
...
```

Code 20. Journal mixin.

Then we simply declare which classes will avail of this functionality. (If there is a name conflict we get a normal compilation error.)

```
/* declare parents declarations ***************** */
declare parents :
  Customer || Address implements Journalable;
```

Code 21. Use of Journal mixin.

Advantages:

- We can introduce Layer Supertype at some later stage during development.
- Easy simulation of multiple inheritance.
- We are not forced to put everything into one layer supertype (or have chain of Layer Supertypes), instead we can use multiple separate Layer Supertypes.

Disadvantages

- We cannot access fields introduced in layer supertype (fro our object) unless we declare it as public but then it can be accessed also from outside (i.e. public field – breaks encapsulation).

5.2 Others

This category also includes patterns: Gateway, Mapper, Separated interface, Registry, Value object, Money, Special case, Plugin, Service stub, Record set. See [22] for more information.

We didn't find any significant crosscutting concerns in those patterns. The advantages of converting these patterns are minimal.

6 Techniques for converting patterns

Here, we will list recommendations for converting patterns.

The conversion process has 2 main anchor points:

1. The first is to find the crosscutting concerns and then create appropriate aspect.
2. Second is to make this aspect reusable, we have to find out which part should be overridable and which should be part of the pattern library.

It's important to maximize the isolation of aspects from the code they are collaborating with. We were trying to avoid coupling by minimizing dependencies. It is often desirable to configure aspects with dependency injection and to depend only on interfaces, which makes it easier to change implementations.

Recommendations:

- the greatest improvement is achieved, when the OO pattern solution structure involves crosscutting of some form, i.e. one object playing multiple roles, many objects playing one role, or an object playing roles in multiple pattern instances,
    - Crosscutting structures in the relationship between roles in the pattern and classes in each instance of the pattern, e.g. in the observer pattern [1], an operation that changes any Subject must trigger notifications of its Observers i.e. the act of notification crosscuts one or more operations in each subject. By using AOP we can modularize the implementation of the pattern (we will also remove code level dependencies from the participant classes to the implementation of the pattern). The modularity

enables a core part of the implementation to be abstracted into reusable code, it can also lead to transparent composition of pattern instances, so that multiple patterns can have shared participants, it also leads to (un)pluggability.

- Find patterns that superimpose roles or define new roles, the roles are defining when the participants have no functionality outside the pattern (e.g. Façade pattern), the roles are superimposed when they are assigned to classes that have functionality and responsibility outside the pattern (e.g. Observer pattern). In object-oriented programming superimposed roles are realized with interfaces and defining roles are realized by inheritance.
- Mixins for simulating multiple inheritance, i.e. with AOP one can supply implementation for an interface and then declare which classes implements this interface[3].
- In order to override some protected behavior of a class we don't have to create a new class that will extend it, we can use AOP inter-types, this can also simulate friendly methods as in C language. We can place the aspect in the same package, by doing this we gain access to package-protected class members. Or we can use the inter-types to extend the target class so we gain access to protected class members, this subclass will exist only for this purpose.
- AOP may be used to support dependency inversion even in classes that wasn't designed to support it.
- To further improve our patterns we can use dependency enforcement, e.g. we can say that the domain layer should not depend on data access layer, we can issue a warning/error message during compilation, this is well supported in some development environments.

7 Performance/Analysis

In this section we will present an analysis of previously observed benefits of implementing patterns with AOP.

[3] One drawback of this approach is that current development environments don't support this well, e.g. with Eclipse AJDT plugin, code that calls such class will be displayed with red underlines (as if there was error), however It will compile without errors.

The advantage of using an aspect is that code changes can be localized to the aspect, even if their effects aren't.

We have analysed three groups of patterns. In the AspectJ implementation we were able to find modularity benefits: locality, reusability, composition transparency, (un)pluggability and inversion of control (dependency inversion).

The biggest benefits were observed in Object-Relational Behavioral Patterns. AspectJ implementations of 6 of 6 patterns were localized. For 2(3) of these the locality enables a core part of the implementation to be abstracted into reusable code. All patterns in this group have the benefit of transparent composability so that multiple patterns can have shared participants. In 2 of 6 we observed (un)pluggability of the pattern instances.

In the other two groups we didn't observed bigger advantages, the exception is Layer Supertype which has all of the benefits. The results are depicted in Table 1.

The following are the most common benefits that patterns have:

- Locality – All the code that implements the pattern is localized in the aspect/protocol (and the associated classes) and none of it is in the participating client classes. Potential changes to the pattern instance are confined to one place. Also, the existence of a single named 'unit of pattern code' makes the presence and the structure of the pattern more explicit (improved documentation of the code).
- Reusability – The core pattern is abstracted and reusable. The pattern protocol generalizes overall pattern behavior. The abstract aspect can be shared and reused across multiple pattern instances. For each pattern instance, we only need to define one concrete aspect.
- Composition transparency – The client implementation is not coupled to the pattern, so it can participate in other kinds of pattern relationships and the resulting code does not become more complicated.

- (Un)Pluggability – The client implementation is not aware of its role in a pattern, it is possible to switch effortlessly between using the pattern and not using it n the system.

Table 1. Design patterns, roles, and properties of their AOP implementations.

	Modularity Properties				Kinds of Roles	
Pattern Name	Locality	Reusability	Composition Transparency	(Un)pluggability	Defining	Superimposed
Object-Relational Behavioral Patterns						
Lazy initialization	yes	no	yes	no		(Lazy initialization)
Virtual proxy	yes	no	yes	no	Virtual proxy	
Value holder	yes	no	yes	no	(Value holder)	Value holder
Ghost	yes	yes	n/a	no		Ghost
Unit of work	yes	yes	yes	yes		(Unit of work)
Identity map	yes	yes	yes	yes	(Identity map)	
Domain Logic Patterns						
Transaction script	Same implementation for Java and AspectJ.					(Transaction script)
Domain model	Same implementation for Java and AspectJ.					(Domain model)
Table module	Same implementation for Java and AspectJ.					(Table module)
Service layer	Same implementation for Java and AspectJ.					(Service layer)
Base Patterns						
Gateway	Same implementation for Java and AspectJ.				(Gateway)	
Mapper	Same implementation for Java and AspectJ.				Mapper, MapperImpl	
Layer supertype	yes	yes	yes	(yes)		(Layer supertype)
Separated	Same implementation for Java and AspectJ.				(Implementat	(Separated

interface		ion)	interface)
Registry	Same implementation for Java and AspectJ.	Registry	
Value object	Same implementation for Java and AspectJ.	Value object	
Money	Same implementation for Java and AspectJ.	Money	
Special case	Same implementation for Java and AspectJ.		Special case
Plugin	Same implementation for Java and AspectJ.	Plugin	
Service stub	Same implementation for Java and AspectJ.		Service stub
Record set	Same implementation for Java and AspectJ.	Record set	

(yes) - means that that some restrictions apply.

One major advantage in the AOP implementation is primarily due to inverting dependencies, so that pattern code depends on participants, not the other way round. This is directly related to locality – dependencies between patterns and participants are localized in the pattern code. By moving the dependency inside an aspect and weaving it afterwards we can easily remove cyclic dependencies.

In each case we were trying to create three level structure as can be seen below.

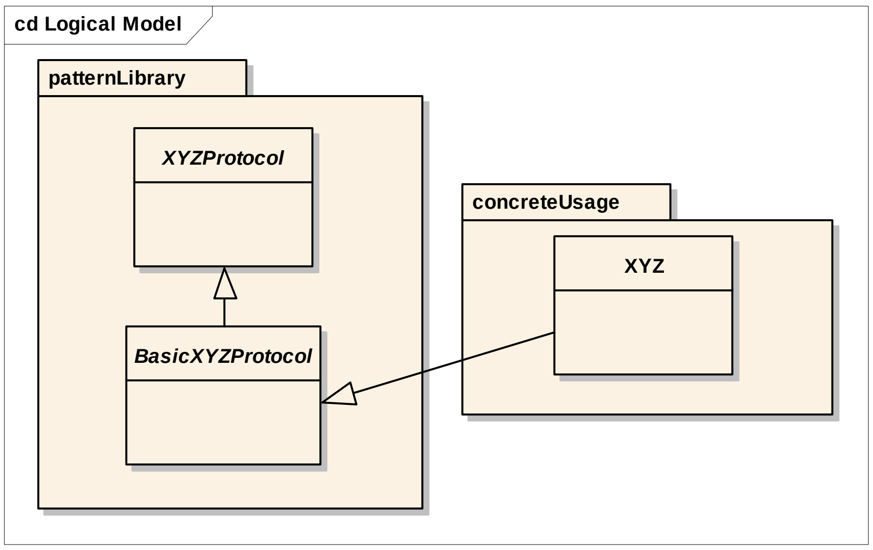


Figure 14. Proposed pattern model.

The root of the hierarchy – abstract aspect `XYZProtocol` tries to provide generic functionality (abstract pointcuts, generic method implementations that use this pointcuts). `BasicXYZProtocol` provides our proposed implementation of the protocol (can introduce naming conventions). This is an abstract reusable aspect. Finally, `XYZ` represent a concrete instance of this pattern. The aim was to make it as simple as possible, because this is the part that each developer, that wants to use this pattern, must implement. It usually just defines which class plays which role, this is done with `declare parents` construct.

If an object or class is unaware of its role in a pattern it can be used in different contexts (such as without applying the pattern) without modifications or redundant code, which increases the reusability of participants. If participants don't need to have pattern specific code, they can be easily removed from or added to a particular pattern instance – effectively providing (un)pluggability. The participant must have a meaning outside of the pattern implementation to benefit from this. Nice example are participants of Identity Map, because they have other responsibilities in the application they are in (e.g. `PersonFinder` – accessing the persistent storage), while `Money` just encapsulates internal state.

The benefit of locality also allows developers to easily implement global policies related to design patterns, such as adding thread safety, logging facilities or performance optimizations. Changes to communication protocols or methods that are part of the abstract classes or interfaces involved do not require adjusting all participants.

Composition of design patterns is a complex case in a traditional OO application, as pattern code tends to be diffused across multiple classes and tangled with code from multiple concerns. Aspect oriented programming offers means to abstract the behavioural relationship between concerns, which makes it easier to model the way design patterns interact. This leads to the following problem: either we can express a composition of design patterns as a composition of their aspectized forms (when pattern instances are defined in separate modular units), or either we can only extract an aspectization of the composition. The first case is obviously more interesting as it allows for a reusable expression of composition. This solves a common problem with having multiple instances of a design pattern in one application.

The modularity of design pattern results also in documentation benefit. Since the pattern is localized the entire description of a pattern instance is also localized and does not get lost or degenerate in the system [1].

Improvement in the Layer Supertype case is directly related to lack of multiple inheritance in Java. In Java, if a participant has to inherit more kinds of functionality, it has to be realized as an interface. Unfortunately, interfaces in Java cannot contain code, making it impossible to attach default implementations of methods. We have showed how this can be done by using mixins in AOP. This allows us to attach both interfaces and implementation to existing classes.

Enforcement aspects provide a good way to further improve the modularity of our code (they can be useful for enforcing dependencies in our patterns). AOP enables us to write enforcement aspects that will ensure the integrity of the design is maintained as the code evolves (i.e. we don’t inadvertently create a dependency from the domain model on the user interface component, or call data access API outside of the data access package). This is especially useful if a new developer joins the project, and he is not familiar with the application blueprint.

Some of presented patterns are already part of some frameworks. However, we have showed how these patterns can be applied without use of heavyweight frameworks.

We realize that it is very important to ensure that reusable aspects handle all the valid interactions among modules with which they interact. This is especially true for reusable libraries. A reusable library that performs tasks for another library needs to track all the valid patterns of use for that library, not just ones that we are aware of. Since this thesis is about implementing existing design patterns by using AOP approach, this isn’t an issue.

8 Conclusion

The importance of having high quality of code is growing more and more in importance. One part of achieving this is to use well known design patterns. Design pattern is a general repeatable solution to a commonly-occurring problem in software design. They encapsulate experience, provide a common vocabulary for computer scientists and they also enhance the documentation of software design.

Enterprise applications often consist of hundred thousands lines of code. The importance of using design patterns is much more important. Martin Fowler has identified common used patterns in enterprise application architecture. Patterns he identified are designed for object oriented world with all the advantages and disadvantages of the object-oriented solution. Some of these patterns involve significant crosscutting concerns with their object-oriented implementation.

The advantage of using an aspect is that code changes can be localized to the aspect, even if their effects aren't. The improvement from using aspect-oriented solution in pattern implementation directly relates to the presence of crosscutting structure in the pattern. The crosscutting structure comes mainly from patterns that superimpose behavior on their participants (i.e. in UniOfWork design pattern every getter method in every object must mark itself as being dirty).

Aspect oriented programming does not conflict with or displace object oriented design and programming. In reality it is a complementary technology. Crosscutting concerns cause code pollution across modular units that interfere with core business logic and create tangled, difficult-to-maintain, and brittle code. Existing programming languages and design methodologies for enterprise applications have no mechanisms to separate crosscutting concerns cleanly. AOP lets the developer to encapsulate and organize code for crosscutting concerns in the form of aspects. Control is acquired over when and where the crosscutting code executes by selecting join points with pointcuts. Advice can be used to specify the code that is executed when join points are matched. With inter-type declarations one can add new fields and methods to existing classes as part of an aspect.

Our aspect-oriented solution has many improvements. The biggest are modularity improvements. In many cases the pattern implementations are more localized, reusable, composable and also (un)pluggable. They help to defeat the 'ultimate code smell' [23] – code duplication.

By using our pattern library the developer can focus more on what he is actually doing instead of how is he doing it. We have tried to use the POJO approach as much as possible. Many of the patterns can be applied later on in the development phase.

On the other side AOP is still a new, for most developers, an unknown field. One has to be careful when using this powerful tool. Our belief is that its use in libraries or frameworks is reasonable. Also the use of an integrated development environment can be of great advantage when developing with aspects. It can visualise the effect of aspects on the code base.

Our results suggest that aspect-oriented programming should be strongly considered in the design and implementation of enterprise applications.

8.1 Future developments

Further work can be done in:

- Applying aspect-oriented implementation to more object-oriented patterns.
- Further improve the aspect-oriented implementations, consider using different aspect-oriented implementation strategy than AspectJ.
- Identifying design patterns for AOP, i.e., commonly used idioms/patterns/code fragments in AOP languages. Very little work has been done there so far, most likely because it is still a new field and it requires big code bases of projects using AOP to be able to identify such patterns.

8.2 Learning outcomes

- Knowledge of various existing design patterns.
- Knowledge of aspect-oriented programming.
- Practical experience with design and implementation.

References

[1] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, pages 161–173. ACM Press, 2002.

[2] L. Bergmans and M. Aksit. Reusability problems in object-oriented concurrent programs. In Proc. ECOOP'92 Workshop Object-Based Concurrency and Reuse, June 1992.

[3] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. Comm. ACM, 44(10):51–57, October 2001.

[4] L. Bougé and N. Francez. A compositional approach to superimposition. In 15th Symp. Principles of Programming Languages (POPL), pages 240–249. ACM, 1988.

[5] Gilad Bracha and William Cook. Mixin-based inheritance. In Conf. Object-Oriented Programming: Systems, Languages, and Applications; European Conf. Object-Oriented Programming, pages 303– 311. ACM, 1990.

[6] Carine Courbis and Anthony Finkelstein. Towards aspect weaving applications. In ICSE '05: Proceedings of the 27th international conference on Software engineering, pages 69–77, New York, 2005. ACM Press.

[7] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck. Handling crosscutting constraints in domain-specific modeling. Comm. ACM, 44(10):87–93, October 2001.

[8] Elizabeth A. Kendall. Role model designs and implementations with aspect-oriented programming. In Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 353–369. ACM Press, 1999.

[9] Rajeev R. Raje, Ming Zhong, and Tongyu Wang. Case study: A distributed concurrent system with AspectJ. ACM SIGAPP Applied Computing Review, 9(2):17–23, 2001.

[10] Therthala Murali, Renaud Pawlak and Houman Younessi. Applying Aspect Orientation to J2EE Business Tier Patterns. In the AOSD 2004 Aspects, Components, and Patterns for Infrastructure Software workshop (ACP4IS), pages 55–61, Lancaster, United Kingdom, March 2004.

[11] Monteiro, M. P., Fernandes, J. M., “Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns”, proceedings of the DSOA'2004 workshop at JISBD 2004 (IX Jornadas de Ingeniería de Software y Bases de Datos), Málaga, Spain, November 2004.

[12] Agerbo, E., Cornils, A. How to preserve the benefits of Design Patterns. Proceedings of OOPSLA 1998, pp. 134-143.

[13] Vander Alves and Paulo Borba. A design pattern for distributed applications. In Adriano Souza et al., editors, XIV Brazilian Symposium on Software Engineering- Minicourses and Tutorials, pages 191–219, October 2000.

[14] Vander Alves and Paulo Borba. Distributed adapters pattern: A design pattern for object-oriented distributed applications. In First Latin American Conference on Pattern Languages Programming, SugarLoaf- PLoP 2001, October 2001. Published in UERJ Magazine: Special Issue on Software Patterns, June 2002, pages 132-142.

[15] Siobhán Clarke and Robert J.Walker. Composition patterns: An approach to designing reusable aspects. In Proc. 23rd Int’l Conf. Software Engineering (ICSE), pages 5–14, May 2001.

[16] Siobhán Clarke and Robert J. Walker. Separating crosscutting concerns across the lifecycle: From composition patterns to AspectJ and Hyper/J. Technical Report TCDCS- 2001-15, Trinity College, Dublin, May 2001.

[17] Fowler M., Dealing with Roles, Collected papers from the PLoP '97 and EuroPLoP '97 Conference, Technical Report #wucs-97-34, Dept. of Computer Science, Washington University Department of Computer Science, September 1997. URL:http://www2.awl.com/cseng/titles/0-201-89542-0/apsupp/roles2-1.html.

[18] Griswold G. William, Shonle Macneil, Sullivan Kevin, Song Yuanyuan, Tewari Nishit, Cai Yuanfang, Rajan Hridesh, Modular Software Design with Crosscutting Interfaces, IEEE Software, vol. 23, no. 1, pp. 51-60, Jan/Feb, 2006.

[19] Boyen N., Lucas C., and Steyaert P., Generalized mixin-based inheritance to support multiple inheritance, Technical Report vub-prog-tr-94-12, Vrije Universiteit Brussel, 1994.

[20] S. Hanenberg and R. Unland. Specifying aspect-oriented design constraints in AspectJ. In Workshop on Tools for Aspect-Oriented Software Development at OOPSLA 2002.

[21] Gamma, E. et al. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[22] Fowler M., Patterns of Enterprise Application Architecture, on-line available at http://www.martinfowler.com/eaaCatalog/, last check at 24 July 2006.

[23] Fowler M et. al., Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999, ISBN: 0-201-485672, 464 pages.

[24] Colyer A., Clement A., Harley G., Webster M. Eclipse AspectJ, Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. Addison Wesley. December 2004.

[25] Coyler A., on-line available at http://aspectprogrammer.org/, last check at 24 July 2006.

[26] developerWorks AOP@Work: within Java technology, on-line available at http://www-128.ibm.com/developerworks/search/searchResults.jsp?searchType=1&searchSite=dW&searchScope=javaZ&query=AOP@Work%3A&Search.x=42&Search.y=10, last check at 24 July 2006.

[27] http://en.wikipedia.org/wiki/Design_pattern_(computer_science), last check at 24 July 2006.

[28] http://www.martinfowler.com/bliki/POJO.html, last check at 24 July 2006.

Appendix A – UnitOfWork library aspect

Here we will list full source code of the library aspect that we have created for UnitOfWork design pattern.

```
/**
 * This file is part of diploma project -
 * Design Patterns in Aspect Oriented Programming.
 * Created on 23.4.2006.
 */
package sk.misobali.aspectPatterns.patternLibrary;

/**
 * dclares standard pointcuts for UnitOfWork design pattern
 *
 * @author Michal Bali
 */
public abstract aspect UnitOfWorkProtocol {

  /**
   * inserts object into persistent storage
   * @param obj
   */
  public abstract void doInsertNew(Object obj);

  /**
   * updates object in persitent storage
   * @param obj
   */
  public abstract void doUpdateDirty(Object obj);

  /**
   * removes object from persistens storage
   * @param obj
   */
  public abstract void doDeleteRemoved(Object obj);

  /**
   * captures joinpoints when an object gets dirty
   */
  protected abstract pointcut dirtyObject();

  /**
   * captures joinpoints when an objct is created
   */
  protected abstract pointcut newObject();

  /**
   * captures joinpoints when an object is deleted
   */
  protected abstract pointcut deleteObject();

  /**
   * captures joinpoints when an object is loaded
   * used by IdentityMap design pattern
   */
  protected abstract pointcut cleanObject();

}
```

```
/**
```

```
 * This file is part of diploma project -
 * Design Patterns in Aspect Oriented Programming.
 * Created on 4.7.2006.
 */
package sk.misobali.aspectPatterns.patternLibrary.unitofwork;

import sk.misobali.aspectPatterns.patternLibrary.UnitOfWorkProtocol;

/**
 * basic implementation of UnitOfWork design pattern
 *
 * @author Michal Bali
 */
public abstract aspect BasicUnitOfWorkProtocol extends UnitOfWorkProtocol {

  protected abstract void setUp();

  protected pointcut dirtyObject():
    execution(* DomainObject+.set*(..)) &&
    !execution(* DomainObject.set*(..));
    //set(* Entity+.*);

  protected pointcut newObject():
    execution(DomainObject+.new(..)) &&
    cflowbelow(execution(* DomainObject+.create(..))) &&
    within(DomainObject+) &&
    !within(DomainObject);

  protected pointcut deleteObject():
    execution(* DomainObject+.remove(..)) ||
    execution(* DomainObject.remove(..));

  /*protected pointcut cleanObject():
    execution(DomainObject+.new(..)) &&
    !execution(DomainObject.new(..)) &&
    cflowbelow(execution(* Mapper+.find*(..)));*/

  after(DomainObject obj) returning():
    dirtyObject() && this(obj) {
    UnitOfWork.getCurrent().registerDirty(obj);
  }

  after(DomainObject obj) returning():
    newObject() && this(obj) {
    UnitOfWork.getCurrent().registerNew(obj);
  }

  after(DomainObject obj) returning():
    deleteObject() && this(obj) {
    UnitOfWork.getCurrent().registerRemoved(obj);
  }

  after(DomainObject obj) returning():
    cleanObject() && this(obj) {
    UnitOfWork.getCurrent().registerClean(obj);
  }
}
```

```
/**
 * This file is part of diploma project -
 * Design Patterns in Aspect Oriented Programming.
 * Created on 11.7.2006.
 */
package sk.misobali.aspectPatterns.patternLibrary.unitofwork;

/**
 *
 * @author Michal Bali
 */
public interface DomainObject {
  Long getId();
  void setId(Long id);
}
```

```
/**
 * This file is part of diploma project -
 * Design Patterns in Aspect Oriented Programming.
 * Created on 23.4.2006.
 */
package sk.misobali.aspectPatterns.patternLibrary.unitofwork;

import java.util.ArrayList;
import java.util.List;

import sk.misobali.aspectPatterns.patternLibrary.UnitOfWorkProtocol;

/**
 *
 * @author Michal Bali
 */
public class UnitOfWork {
  private List<DomainObject> newObjects = new ArrayList<DomainObject>();
  private List<DomainObject> dirtyObjects = new ArrayList<DomainObject>();
  private List<DomainObject> removedObjects = new ArrayList<DomainObject>();
  private static ThreadLocal<UnitOfWork> current = new
ThreadLocal<UnitOfWork>();
  //current UnitOfWorkProtocol
  private static UnitOfWorkProtocol uowProtocol;
  public void registerNew(DomainObject obj) {
    assert obj.getId() != null : "id is null";
    assert !dirtyObjects.contains(obj) : "object is dirty";
    assert !removedObjects.contains(obj) : "object is removed";
    //assert !newObjects.contains(obj) : "object already registered new";
    if (!newObjects.contains(obj)) {
      newObjects.add(obj);
    }
  }
  public void registerDirty(DomainObject obj) {
    assert obj.getId() != null : "id is null";
    assert !removedObjects.contains(obj) : "object is removed";
    //check if this object has already been registred as dirty/new
    if (!dirtyObjects.contains(obj) && !newObjects.contains(obj)) {
      dirtyObjects.add(obj);
    }
  }
  public void registerRemoved(DomainObject obj) {
    assert obj.getId() != null : "id is null";
    //if this object is new -> just remove him from 'new' collection
    if (newObjects.remove(obj))
      return;
    dirtyObjects.remove(obj);
    if (!removedObjects.contains(obj)) {
      removedObjects.add(obj);
    }
  }
  public void registerClean(DomainObject obj) {
    assert obj.getId() != null : "id is null";
  }
  public void commit() {
    insertNew();
    updateDirty();
    deleteRemoved();
  }
  private void insertNew() {
    for (DomainObject obj : newObjects) {
      uowProtocol.doInsertNew(obj);
    }
  }
  private void updateDirty() {
    for (DomainObject obj : dirtyObjects) {
      uowProtocol.doUpdateDirty(obj);
    }
  }
  private void deleteRemoved() {
    for (DomainObject obj : removedObjects) {
      uowProtocol.doDeleteRemoved(obj);
    }
  }
  public static void newCurrent() {
    setCurrent(new UnitOfWork());
  }
  public static void setCurrent(UnitOfWork uow) {
    current.set(uow);
  }
  public static UnitOfWork getCurrent() {
    return current.get();
  }
  public static void setUowProtocol(UnitOfWorkProtocol uowProtocol) {
    UnitOfWork.uowProtocol = uowProtocol;
  }
}
```

Appendix B – UnitOfWork pattern instance

Here we will list full source code of sample program that is using AspectJ version of UnitOfWork design pattern.

```
/**
 * This file is part of diploma project -
 * Design Patterns in Aspect Oriented Programming.
 * Created on 23.4.2006.
 */
package sk.misobali.aspectPatterns.examples.unitofwork.aspectj;

import sk.misobali.aspectPatterns.patternLibrary.unitofwork.UnitOfWork;
import
sk.misobali.aspectPatterns.patternLibrary.unitofwork.BasicUnitOfWorkProtocol;

import
sk.misobali.aspectPatterns.examples.unitofwork.aspectj.domain.DomainObject;

import sk.misobali.aspectPatterns.examples.unitofwork.aspectj.data.Mapper;
import
sk.misobali.aspectPatterns.examples.unitofwork.aspectj.data.MapperRegistry;

/**
 * basci implementation of UnitOfWorkProtocol
 * all advices forwards the call to UnitOWork object
 * introduces some naming conventions:<br>
 *   - each setter should be named set...
 *   - creation of new objects should happen in create... method
 *   inside DomainObject object
 *   - removal of an object should happen inside remove... method
 *   inside DomainObject object
 * @author Michal Bali
 */
public aspect AudioUnitOfWork extends BasicUnitOfWorkProtocol {

  declare parents : DomainObject implements
sk.misobali.aspectPatterns.patternLibrary.unitofwork.DomainObject;

  public void setUp() {
    UnitOfWork.setUowProtocol(AudioUnitOfWork.aspectOf());
  }

  /**
   * inserts objects into persistent storage
   * @param obj
   */
  public void doInsertNew(Object obj) {
    MapperRegistry.getMapper(obj.getClass())
      .insert((DomainObject) obj);
  }
  /**
   * updates objects in persitent storage
   * @param obj
   */
  public void doUpdateDirty(Object obj) {
    MapperRegistry.getMapper(obj.getClass())
      .update((DomainObject)obj);
  }
  /**
   * removes objects from persistens storage
   * @param obj
   */
  public void doDeleteRemoved(Object obj) {
    MapperRegistry.getMapper(obj.getClass())
      .delete((DomainObject)obj);
  }

  protected pointcut cleanObject():
    execution(DomainObject+.new(..)) &&
    !execution(DomainObject.new(..)) &&
    cflowbelow(execution(* Mapper+.find*(..)));

}
```

Domain model:

```
/**
 * This file is part of diploma project -
 * Design Patterns in Aspect Oriented Programming.
 * Created on 23.4.2006.
 */
package sk.misobali.aspectPatterns.examples.unitofwork.aspectj.domain;

import java.util.Date;

/**
 *
 * @author Michal Bali
 */
public class Album extends DomainObject {
  private String title;
  private Date created;
  private static long idGenerator = 1000L; //for testing
  public Album() {
  }
  public Album(Long id, String name) {
    this.id = id;
    this.title = name;
  }
  public static Album create(String name) {
    Album obj = new Album(idGenerator++, name);
    return obj;
  }
  public void remove() {
    //can be overriden here
  }
  public void setTitle(String title) {
    this.title = title;
  }
  public String getTitle() {
    return title;
  }
  public Date getCreated() {
    return created;
  }
  public void setCreated(Date created) {
    this.created = created;
  }
}
```

```
/**
 * This file is part of diploma project -
 * Design Patterns in Aspect Oriented Programming.
 * Created on 24.4.2006.
 */
package sk.misobali.aspectPatterns.examples.unitofwork.aspectj.domain;
```

```
/**
 *
 * @author Michal Bali
 */
public class Author extends DomainObject {
  private String firstName;
  private String surName;
  private static long idGenerator = 1000L; //for testing
  public Author() {
  }
  public Author(Long id, String firstName, String surName) {
    this.id = id;
    this.firstName = firstName;
    this.surName = surName;
  }
  public static Author create(String firstName, String surName) {
    Author obj = new Author(idGenerator++, firstName, surName);
    return obj;
  }
  public String getFirstName() {
    return firstName;
  }
  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }
  public String getSurName() {
    return surName;
  }
  public void setSurName(String surName) {
    this.surName = surName;
  }

}
```

```
/**
 * This file is part of diploma project -
 * Design Patterns in Aspect Oriented Programming.
 * Created on 23.4.2006.
 */
package sk.misobali.aspectPatterns.examples.unitofwork.aspectj.domain;

/**
 *
 * @author Michal Bali
 */
public class DomainObject {
  protected Long id;
  public Long getId() {
    return id;
  }
  public void setId(Long id) {
    this.id = id;
  }
}
```

Data layer:

```
package sk.misobali.aspectPatterns.examples.unitofwork.aspectj.data;

import
sk.misobali.aspectPatterns.examples.unitofwork.aspectj.domain.DomainObject;

public interface Mapper<T extends DomainObject> {
  T find(Long key);
  void insert(T obj);
  void update(T obj);
  void delete(T obj);
}
```

```
/**
 * This file is part of diploma project -
 * Design Patterns in Aspect Oriented Programming.
 * Created on 23.4.2006.
 */
package sk.misobali.aspectPatterns.examples.unitofwork.aspectj.data;

import java.util.HashMap;
import java.util.Map;
import sk.misobali.aspectPatterns.examples.unitofwork.aspectj.domain.Album;
import sk.misobali.aspectPatterns.examples.unitofwork.aspectj.domain.Author;

/**
 * for tesing
 *
 * @author Michal Bali
 */
public class MapperRegistry {

  protected static Map<Class, Mapper> mappers =
    new HashMap<Class, Mapper>();

  static {
    mappers.put(Album.class, new AlbumMapper());
    mappers.put(Author.class, new AuthorMapper());
  }

  public static Mapper getMapper(Class clazz) {
    return mappers.get(clazz);
  }

}
```

```
/**
 * This file is part of diploma project -
 * Design Patterns in Aspect Oriented Programming.
 * Created on 23.4.2006.
 */
package sk.misobali.aspectPatterns.examples.unitofwork.aspectj.data;

import sk.misobali.aspectPatterns.examples.unitofwork.aspectj.domain.Author;

/**
 * example implementation for testing
 *
 * @author Michal Bali
 */
public class AuthorMapper implements Mapper<Author> {
  public Author find(Long key) {
    Author author;
    System.out.println("running SELECT * FROM AUTHOR WHERE AUTHOR_ID=" + key);
    author = new Author(key, "Stephen", "King");
    return author;
  }
  public void insert(Author obj) {
    System.out.println("running INSERT INTO AUTHOR ...");
  }
  public void update(Author obj) {
    System.out.println("running UPDATE AUTHOR SET ... WHERE AUTHOR_ID=" +
obj.getId());
```

```
  }
  public void delete(Author obj) {
    System.out.println("running DELETE FROM AUTHOR WHERE AUTHOR_ID=" +
obj.getId());
  }
}
```

```
/**
 * This file is part of diploma project -
 * Design Patterns in Aspect Oriented Programming.
 * Created on 23.4.2006.
 */
package sk.misobali.aspectPatterns.examples.unitofwork.aspectj.data;

import sk.misobali.aspectPatterns.examples.unitofwork.aspectj.domain.Album;

/**
 * example implementation for testing
 *
 * @author Michal Bali
 */
public class AlbumMapper implements Mapper<Album> {
  public Album find(Long key) {
    Album album;
    System.out.println("running SELECT * FROM ALBUM WHERE ALBUM_ID=" + key);
    album = new Album(key, "QUEEN");
    return album;
  }
  public void insert(Album obj) {
    System.out.println("running INSERT INTO ALBUM ...");
  }
  public void update(Album obj) {
    System.out.println("running UPDATE ALBUM SET ... WHERE ALBUM_ID=" +
obj.getId());
  }
  public void delete(Album obj) {
    System.out.println("running DELETE FROM ALBUM WHERE ALBUM_ID=" +
obj.getId());
  }
}
```

```
/**
 * This file is part of diploma project -
 * Design Patterns in Aspect Oriented Programming.
 * Created on 23.4.2006.
 */
package sk.misobali.aspectPatterns.examples.unitofwork.aspectj.data;

import sk.misobali.aspectPatterns.examples.unitofwork.aspectj.domain.Album;
import sk.misobali.aspectPatterns.patternLibrary.unitofwork.UnitOfWork;
import sk.misobali.aspectPatterns.examples.unitofwork.aspectj.AudioUnitOfWork;

/**
 *
 * @author Michal Bali
 */
public class EditAlbumScript {

  public static void updateTitle(Long albumId, String title) {
    AudioUnitOfWork.aspectOf().setUp();

    UnitOfWork.newCurrent();
    Mapper<Album> mapper = MapperRegistry.getMapper(Album.class);
    Album album = mapper.find(albumId);
    album.setTitle(title);
    UnitOfWork.getCurrent().commit();
 }
}
```

Main class:

```
/**
 * This file is part of diploma project -
 * Design Patterns in Aspect Oriented Programming.
 * Created on 23.4.2006.
 */
package sk.misobali.aspectPatterns.examples.unitofwork.aspectj;

import
sk.misobali.aspectPatterns.examples.unitofwork.aspectj.data.EditAlbumScript;

/**
 *
 * @author Michal Bali
 */
public class Main {
  /**
   * @param args
   */
  public static void main(String[] args) {
    System.out.println("AOP example of unit-of-work");

    EditAlbumScript.updateTitle(1L, "Title ABBA");
  }
}
```

Appendix C – UnitOfWork unit tests

Here we will list full source code of unit tests for one of the converted design patterns – UnitOfWork.

```
public class UnitOfWorkTest extends TestCase {

  Mock mapperMock, unitOfWorkMock;

  Album album;

  @Override
  protected void setUp() throws Exception {
    super.setUp();

    AudioUnitOfWork.aspectOf().setUp();

    mapperMock = new Mock(Mapper.class);
    unitOfWorkMock = new Mock(new CGLIBCoreMock(UnitOfWork.class));

    album = new Album(1L, "Test album");

    UnitOfWorkTestHelper.setMapper(Album.class, (Mapper)mapperMock.proxy());
  }

  @Override
  protected void tearDown() throws Exception {
    super.tearDown();

    UnitOfWorkTestHelper.setMapper(Album.class, new AlbumMapper());
  }

  /**
   * test if nothing changes with objects, no interaction with DB will take
place
   */
  public void testNothing() {
    UnitOfWork.newCurrent();

    mapperMock.expects(new TestFailureMatcher("should not be invoked"));

    UnitOfWork.getCurrent().commit();

    mapperMock.verify();
  }

  public void testUpdate() {
    UnitOfWork.newCurrent();

    album.setTitle("New Title");

    mapperMock.expects(new InvokeOnceMatcher()).method("update").with(
        new IsSame(album));

    UnitOfWork.getCurrent().commit();

    mapperMock.verify();
  }

  public void testUpdateMultiple() {
    UnitOfWork.newCurrent();

    album.setTitle("AAA");
    album.setTitle("BBB");
    album.setCreated(new Date());

    mapperMock.expects(new InvokedRecorder()).method("update").with(
        new IsSame(album));

    UnitOfWork.getCurrent().commit();

    mapperMock.verify();
  }

  public void testInsert() {
    UnitOfWork.newCurrent();

    Album album = Album.create("Newly created title");

    mapperMock.expects(new InvokeOnceMatcher()).method("insert").with(
        new IsSame(album));

    UnitOfWork.getCurrent().commit();

    mapperMock.verify();
  }

  public void testDelete() {
    UnitOfWork.newCurrent();

    album.remove();

    mapperMock.expects(new InvokeOnceMatcher()).method("delete").with(
        new IsSame(album));

    UnitOfWork.getCurrent().commit();

    mapperMock.verify();
  }

  public void testUpdateDelete() {
    UnitOfWork.newCurrent();

    album.setTitle("New Title 2");
    album.remove();

    mapperMock.expects(new InvokeOnceMatcher()).method("delete").with(
        new IsSame(album));

    UnitOfWork.getCurrent().commit();

    mapperMock.verify();
  }

  public void testClean() {
    UnitOfWork.setCurrent((UnitOfWork)unitOfWorkMock.proxy());

    unitOfWorkMock.expects(new InvokeOnceMatcher()).method("registerClean");

    Mapper authorMapper = MapperRegistry.getMapper(Author.class);
    DomainObject author = authorMapper.find(1L);

    unitOfWorkMock.verify();
  }

  public void testCRUD() {
    UnitOfWork.newCurrent();

    album.setTitle("New Title 2");
    album.remove();
    Album album2 = Album.create("Newly created title 2");
    Album album3 = Album.create("Newly created title 3");

    mapperMock.expects(new InvokedRecorder()).method("insert").with(
        new IsSame(album2));
    mapperMock.expects(new InvokedRecorder()).method("insert").with(
        new IsSame(album3));
    mapperMock.expects(new InvokedRecorder()).method("delete").with(
        new IsSame(album));

    UnitOfWork.getCurrent().commit();

    mapperMock.verify();
  }

  public void testCRUDMultipleObjects() {
    UnitOfWork.newCurrent();
    Mapper<Author> authorMapper = MapperRegistry.getMapper(Author.class);

    album.setTitle("New Title 2");
    album.remove();
    Author author = authorMapper.find(1L);
    author.setFirstName("Joe");

    mapperMock.expects(new InvokedRecorder()).method("delete").with(
        new IsSame(album));
    mapperMock.expects(new InvokedRecorder()).method("update").with(
        new IsSame(author));

    UnitOfWork.getCurrent().commit();

    mapperMock.verify();
  }

}
```