

# Use Case Modularity using Aspect Oriented Programming

Manali Bhole and Karl Lieberherr

College of Computer and Information Sciences  
Northeastern University  
360 Huntington Avenue,  
Boston MA 02115.  
{manali, lieber}@ccs.neu.edu

**Abstract.** The issue of use case modularity in the context of aspect-oriented programming is studied. AspectJ and its Demeter extension, called DAJ, are used in an experiment to maintain the use case structure from requirements to implementation. The results of the experiment are (1) that DAJ, if used properly, offers better use case modularity than AspectJ alone and (2) that a finer grained join point model would be needed to avoid code duplication in the implementation of use cases that cut across other use cases.

**Keywords.** Aspect Oriented Programming, Use Case Modularity, Class Dictionary Graph, AspectJ, Demeter AspectJ, Persistence.

## 1 Introduction

Use cases offer a systematic and effective way of representing and communicating software systems' requirements. A use case specifies the behavior of a system or a part of a system and describes sets of sequences of actions, including variants that a system performs to yield an observable result of value to an actor [1]. Thus, all use cases together comprise of all the possible ways of using the system.

### 1.1 Use Case Modularity

A software system in its Analysis and Design phase has a well-defined structure involving actors and use cases. In implementation phase, these artifacts get dissolved into multiple, sometimes overlapping software modules. Thus, code implementing a use case is scattered in multiple program modules and a single program module contains code addressing multiple use cases, also referred to as tangling. This results in systems where distinction of which software module addresses which scenario becomes blurred and makes validation of the final product against the requirements cumbersome and complex. We lose "Use Case Modularity" at the implementation level and the resulting software has use cases scattered and tangled. The loss of use case modularity can result in an implementation that is difficult to maintain under changing business requirements.

In systems where use case modularity is not maintained, modifying an existing use case or adding a new use case due to system evolution cause multiple implementation modifications. Such modifications have potential for inconsistencies in the code and increase in maintenance cost. Such modifications to multiple modules can be avoided if each use case is realized into a

separate program module during the implementation phase. This design methodology in which the system is modularized according to use cases captured in requirements' analysis is known as Use Case Driven Software Development [2].

## 2 Background

### 2.1 Use Case Relationships

Use cases can be best described by Use Case Diagrams that comprise of Actors, Use Cases and Relationships among them. The various relationships among use cases are 'include', 'generalization' and 'extend'.

The include relationship is used to separate common or repeated behavior among several use cases. It is an example of delegation, as a set of responsibilities of the system are captured in one place and then other parts include the new aggregation of responsibilities when they need to use that functionality.

Generalization among use cases is similar to that among classes. The child use case inherits the behavior and meaning from its parent use case; the child may override or add to the behavior of its parent; and can be substituted at any place the parent appears.

The extend relationship is used to model the part of use case the user might see as optional system behavior. The base use case implicitly incorporates the behavior of another use case at a location specified indirectly by the extending use case. The base use case may stand alone, but under certain conditions, its behavior may be extended by the behavior of another use case.

### 2.2 Aspect Oriented Programming (AOP)

AOP is the name given to a set of techniques based on the idea that software is better programmed by separately specifying the various *concerns* (or *aspects*), properties, or areas of interest of a system, and describing their relationships [6].

In AOP, we have a Join Point Model, which allows code from aspects (advice) to be interwoven with code from an OO program (called *base program*) at specific points during the base program's execution (called *join points*). AOP along with Object-Oriented constructs (e.g. inheritance, association, etc.) make it possible to directly map use cases to implementation modules. A new language construct that better separates crosscutting use cases is provided by Aspect-Orientation.

Extension points in the base use case can be viewed as join points in the base code and the extending use cases are like advices that modify the program behavior at those join points.

Design Space	Implementation Space (AOP)
Base Use Case	Base Program
Extension Points	Join Points
Extended Use Case	Aspect Advice

### 3 Experimental Setting

This project undertakes two case studies to examine the use of AOP in Use Case Modular Software Development.

#### 3.1 Class Dictionary Graph Extension [3]

A class dictionary graph is described as a group of collaborating classes and their basic relations. According to the set of objects that can be modeled by a class dictionary graph, relationships like Object-Equivalence, Weak-Extension and Extension are recognized between two class dictionary graphs.

We apply use case modularity to the problem of finding correct the relationship between any given two class dictionary graphs. Each use case is modeled as a separate implementation module (class/aspect or a collaboration of classes and/or aspects). AOP is used to separate the cross cutting concerns between use cases and extend the functionality of the system.

#### 3.2 Library System

We simulated a Library System that has various use cases like `BookChechOut`, `BookCheckIn`, `Add/Remove Book`, `Add/Remove User` etc. The system is modularized according to these use cases and AOP is used to accomplish modifications in the requirements specification with minimal modifications to the base code. The extent to which AOP can be used to implement changes to the system's original specification is examined through the incorporation of authentication and data persistence to the Library System. Finally, an evaluation of the AOP solution to the Library System in terms of maintaining Use Case Modularity at the implementation level is presented.

## 4 Case Study 1: Class Dictionary Graph Extension

### 4.1 Introduction:

Class dictionary graph describes a group of collaborating classes and the relationships between them. The vertices of class dictionary graph are classes whereas the edges represent relationships between the vertices. From [3] edges and vertices are defined as:

- **Construction vertex** represents a concrete class that can have instances but can not have subclasses. Its graphical representation is a rectangle.
- **Alternation vertex** represents an abstract class that can not be instantiated. Its graphical representation is a hexagon in class dictionary graph.
- **Repetition vertex** represents a collection class that has indexed parts e.g. a List. Its graphical representation is an overlaid hexagon and rectangle.
- **Construction Edge:** An edge between a construction or alternation vertex and another vertex. Describes has-a or part-of relationship.
- **Alternation Edge:** An edge between alternation vertex and a construction or alternation vertex. It describes is-a relationship. Graphical representation is a double shaft arrow.
- **Repetition Edge:** An edge between repetition vertex and any other vertex. Describes a has-a or part-of relationship.
- **Inheritance Edge:** An edge from a construction or alternation vertex to an alternation vertex. Describes from where a class inherits.

Figure 1 illustrates an example of a class dictionary graph.

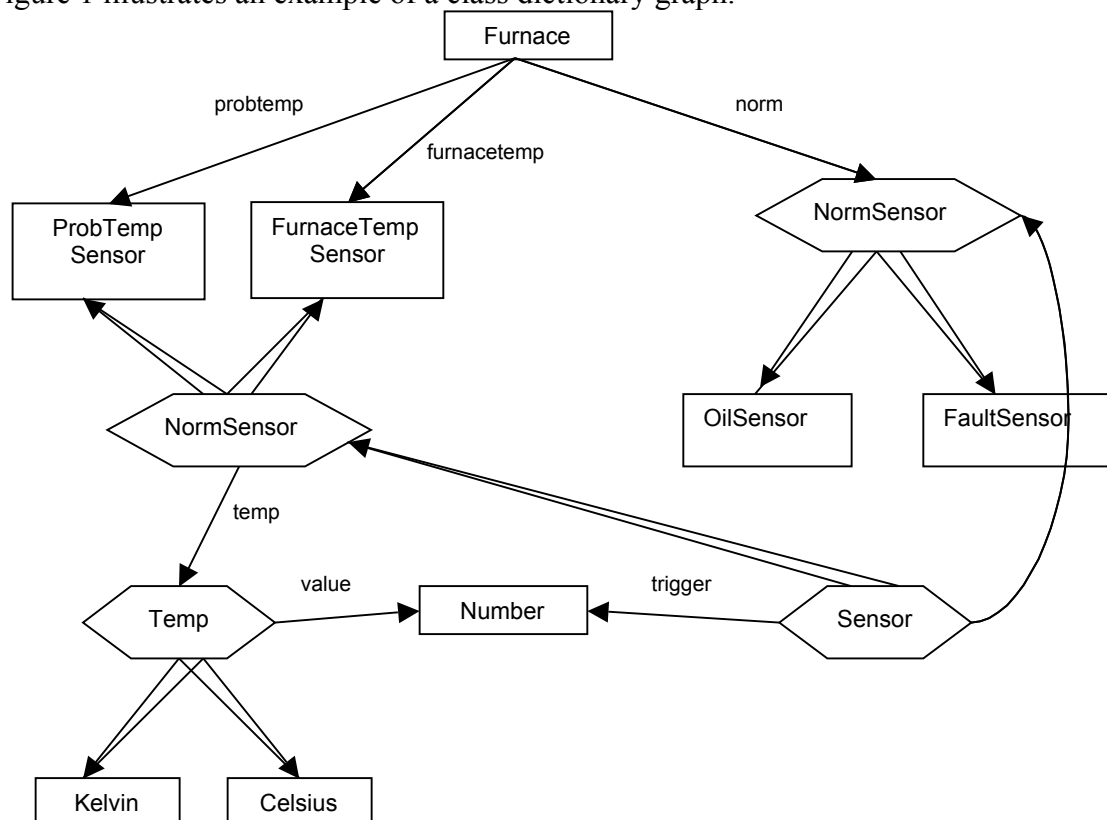


Figure 1: Experimental heating system: class dictionary graph Furnace

## 4.2 Problem Formulation: Class Dictionary Graph Extension.

The extensions of class dictionary graphs capture their transformation as a whole, by analyzing the set of objects that can be modeled by the class dictionary graph. The three extension relations are: object-equivalence, weak-extension, and extension. Object-equivalence preserves the set of objects modeled by the class dictionary graph, weak extension enlarges the set of objects, and extension enlarges and augments the set of objects. Given two class dictionary graphs the task is to find out the relationship between them.

The following use cases can be realized:

1. Given two object-equivalent class dictionary graphs  $G_1, G_2$ , the program will report:  $G_1$  is object-equivalent to  $G_2$ .
2. Given two non object-equivalent class dictionary graphs  $G_1, G_2$ , the program will report:  $G_1$  is not object-equivalent to  $G_2$ . One of the following reasons will be given:
  - $G_1$  weakly extends  $G_2$ ,
  - $G_1$  extends  $G_2$ ,
  - $G_1$  and  $G_2$  are not in any extension relationship.

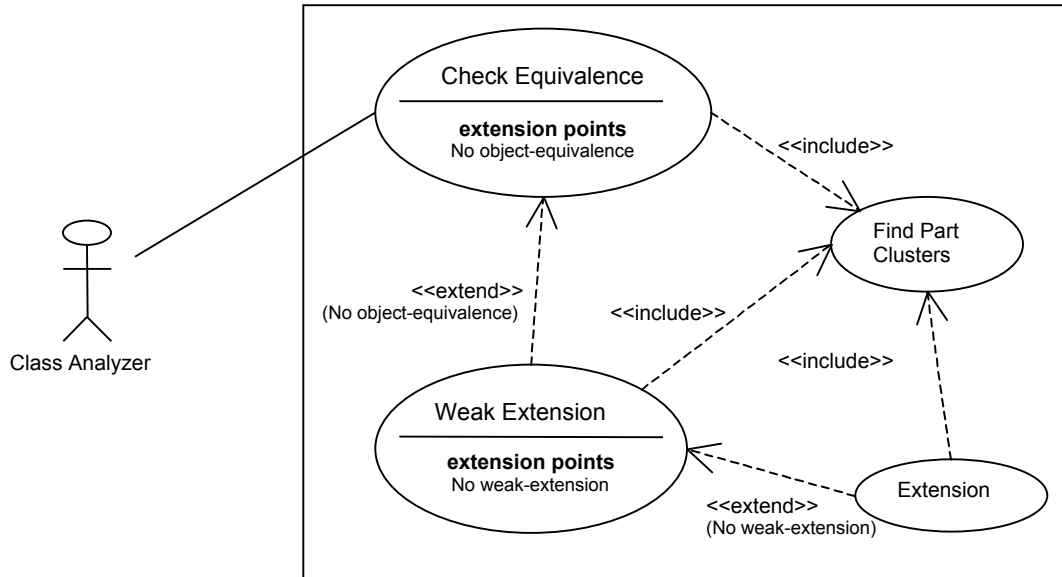


Figure 2: Class dictionary graph extension Use Cases

Figure 2 shows the use case diagram for Class Dictionary Graph Extension. ‘Check Equivalence’ is the base use case that parses two class dictionary graphs. It checks for object equivalence, which is direct result of the comparison of the part clusters of the two class dictionary graphs according to following definition taken from [3].

Let  $G_1$  and  $G_2$  be two class dictionary graphs, where for  $i \in \{1, 2\}$ :

$$G_i = (VC_i, VA_i, \_i; EC_i, EA_i)$$

$VC_i$ : Set of concrete vertices for the  $i^{\text{th}}$  class dictionary graph.

$VA_i$ : Set of alternation vertices for the  $i^{\text{th}}$  class dictionary graph.

$\_i$ : Set of labels for the  $i^{\text{th}}$  class dictionary graph.

$$V_i = VC_i \cup VA_i.$$

$EC_i$ : a ternary relation for the  $i^{\text{th}}$  class dictionary graph on  $V_i \times V_i \times \_i$ . An element  $(v, w, l) \in EC_i$  is a labeled construction edge from vertex  $v$  to vertex  $w$  with label  $l$ .

$EA$ : a binary relation for the  $i^{\text{th}}$  class dictionary graph on  $VA_i \times V_i$ . An element  $(v, w)$  is called an alternation edge from vertex  $v$  to  $w$ .

Part Clusters of a vertex  $v$  is a list of pairs, one for each part of  $v$ . Each pair consists of the part name and the set of construction vertices whose instances can be assigned to the part.

E.g. in Figure 1,

$PartClusters_{Furnace}(ProbTempSensor) = \{(temp, \{Kelvin, Celsius\}), (trigger, \{Number\})\}$ .

The vertex `ProbTempSensor` has two parts `temp` and `trigger`. The part `temp` is inherited from the class `TempSensor` and instance of the class `Kelvin` or `Celsius` can be assigned to it. The part `trigger` is inherited from the class `Sensor` and instance of class `Number` can be assigned.

Class dictionary graph  $G_1$  and  $G_2$  are object-equivalent if  $VC_1 = VC_2$  and for all  $v \in VC_1$ :

$$PartClusters_{G_1}(v) = PartClusters_{G_2}(v).$$

The ‘Find Part Clusters’ use case parses two input class dictionary graphs and finds part clusters of each concrete vertex in both the class dictionary graphs. Using the part clusters found by this use case, ‘Check Equivalence’ use case checks if object-equivalence exists between the two class dictionary graphs. In cases where object-equivalence fails, the ‘No object-equivalence’ extension point occurs and the ‘Weak Extension’ use case is executed. It checks for weak-extension and if weak-extension fails, the ‘No weak-extension’ extension point occurs and the ‘Extension’ use case is executed. It checks for extension relationship between the two class dictionary graphs. If extension fails, there is no relationship between the class dictionary graphs and the program exits by stating the same. The ‘Find Part Cluster’ use case is included by other use cases as they use its result (Figure 2).

### 4.3 Use Case Modular Implementation

First step in the implementation is recognizing the components and subcomponents of the class dictionary analyzer system. For two class dictionaries to be object-equivalent, they must have the same set of construction vertices. So first we check that the set of construction vertices of  $G_1$  equals the set of concrete vertices in  $G_2$ .

Computing part clusters is an important subcomputation. Testing object-equivalence means: find part clusters for each construction class of both class dictionaries, and check that name-equivalent construction classes have the same part clusters. To compute part clusters we need to find all inherited parts. Once the parts are recognized we need to find all the classes whose instances can be assigned to that part. Thus the `findPartCluster` sub problem can be decomposed as follows:

```
part clusters of a class
  compute parts
    compute super classes
    compute all parts
      immediate parts +
      parts of all super classes
  for each part class compute
    set of associated classes
```

Each of the steps involves vigorous traversals through the input class dictionary graphs. The Demeter method [3] is a powerful variant of object-oriented programming that makes programs structure-shy by using only the minimal information about the implementation specific class structure when writing the behavior, otherwise known as Adaptive Programming (AP). We deploy adaptive programming technique using DAJ (Demeter AspectJ) [4] to implement these subcomponents.

Using DAJ for the class analyzer system we have the following classes/components:

1. **ClassGraph.cd file:** This is the class dictionary that defines the textual notation for the class dictionary graphs. The class dictionary we used deals with simplified data ignoring optional parts, syntax and repetition vertices. It is used to interpret the parsed-in class dictionary graphs thereby categorizing the vertices as construction or alternation vertices and edges as alternation or construction edges.

The class dictionary is as follows:

```
Cd_graph = <adjacencies> List (Adjacency).
Adjacency = <source> Vertex <ns> Neighbors ".".
Neighbors: Construct_ns | Alternat_ns common <construct_ns> List
(Labeled_vertex).
Labeled_vertex = "<" <label_name> Ident ">" <vertex>
Comma_list(Vertex).
Alternat_ns = ":" <alternat_ns> Bar_list(Vertex) [<common> Common].
Common = "common".
Construct_ns = "=".
Vertex = <vertex_name> Ident.
List(S) ~ {S}.
Comma_list(S) ~ S {"," S}.
Bar_list(S) ~ S {"|" S}.
```

2. **cg.trv:** This file contains the traversal strategies for each sub task. A traversal strategy can be understood in terms of a subgraph. The subgraph describes a group of collaborating classes and the traversal scope which in turn determines how objects are going to be traversed. e.g. a traversal strategy for finding parents of a given class is as follows:

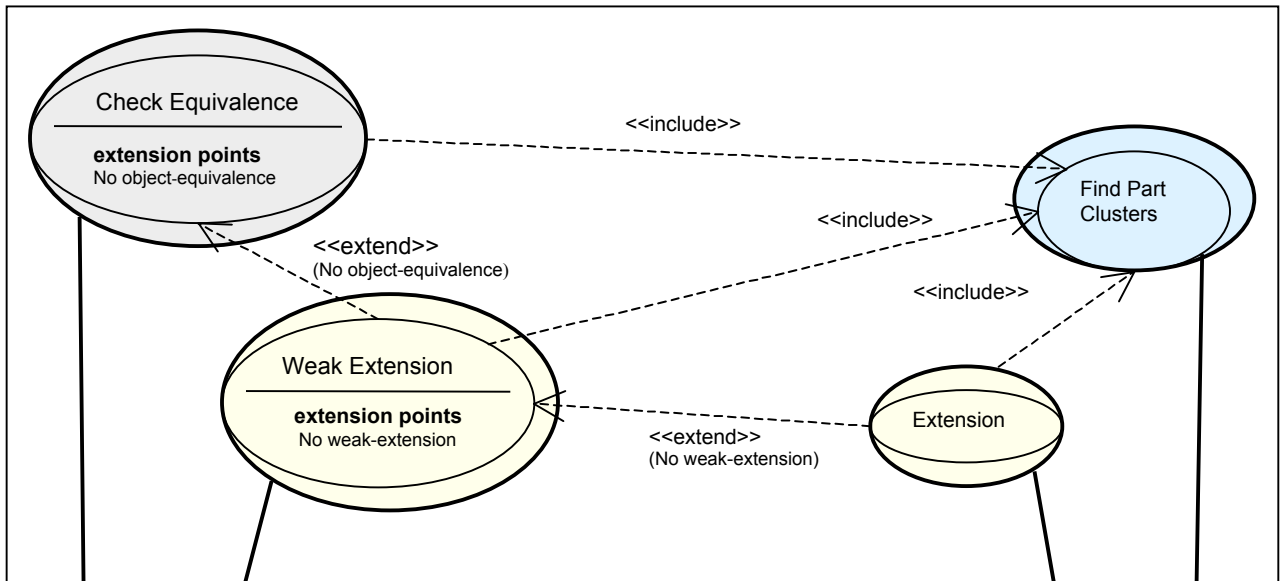
```
1. declare traversal:
2. List find_super(Vertex cs, Cd_graph cdg):
3. "from Cd_graph via Adjacency through -> *,alternat_ns,* to Vertex"
4. (ParentVisitor);
```

This traversal strategy declares a method ‘find\_super’ (line 2) to return a list of parents of a vertex (parameter 1) in given class dictionary graph (parameter 2). The input class dictionary graph contains nodes according to the class dictionary in the file ClassGraph.cd (Bullet 1, section 4.3). The traversal strategy defines how to traverse in the input class dictionary graph to find parent of the given vertex (line 3). It also declares a visitor (line 4) that contains advice for the traversal.

3. **Visitors:** A traversal strategy tells a visitor very precisely how to travel. We need a mechanism to specify what needs to be done on top of the traversal when a node of interest is found. This advice code is defined in a visitor. The visitors declared in this system are PartVisitor, ParentVisitor, VertexVisitor, SubclassVisitor. Each has advice defined for selected nodes.

#### 4.4 Mapping between Use Cases and Implementation modules

Figure 3 shows the mapping between use cases and implementation modules. The use case 'Check Equivalence' is mapped to the class `ObjectEquivalence` that compares two input class dictionary graphs. The `<<include>>` relationship between 'Check Equivalence' and 'Find Part Cluster' use cases is realized as association between classes `ObjectEquivalence` and `PartClusters`. The 'Find Part Cluster' use case is mapped to a collaboration of Visitor classes like `PartVisitor`, `ParentVisitor`, `SubclassVisitor` and `VertexVisitor`. Visitor pattern is used to traverse the class dictionary graph multiple times collecting different information in each traversal. E.g. `ParentVisitor` traverses the class dictionary graph (according to its traversal strategy as explained in previous sub section 4.3, bullet 2) and collects information about vertices that are parents of the vertex under consideration. Similarly, `VertexVisitor` finds vertices representing concrete classes; `SubclassVisitor` finds all sub-vertices and `PartVisitor` finds all the parts of a vertex. The `<<extend>>` relationship between `CheckEquivalence` and `WeakExtension` use case is realized as an aspect `WeakExtension`. The extension point 'No object-equivalence' is mapped to the pointcut `checkWeakExtension`. It intercepts the return value from the method `objectEquivalence` and if it is false, executes the method `weakExtension` which inside the aspect, compares the class dictionary graphs for weak extension. The `<<extend>>` relationship between use cases `WeakExtension` and `Extension` is similarly mapped inside the aspect `Extension` with the pointcut `checkExtension`. The direct mapping of `<<include>>` relationships between use cases 'Weak Extension', 'Extension' and 'Find Part Cluster' is the association between aspects `WeakExtension`, `Extension` and the collaboration of visitor classes. However, for optimization purposes, the part clusters computed in class `ObjectEquivalence` are stored in hash maps that are later reused for look up by the aspects `WeakExtension` and `Extension`, thus eliminating the association between aspects and visitors in the implementation phase as shown in Figure 3.



Use cases in Requirement Phase

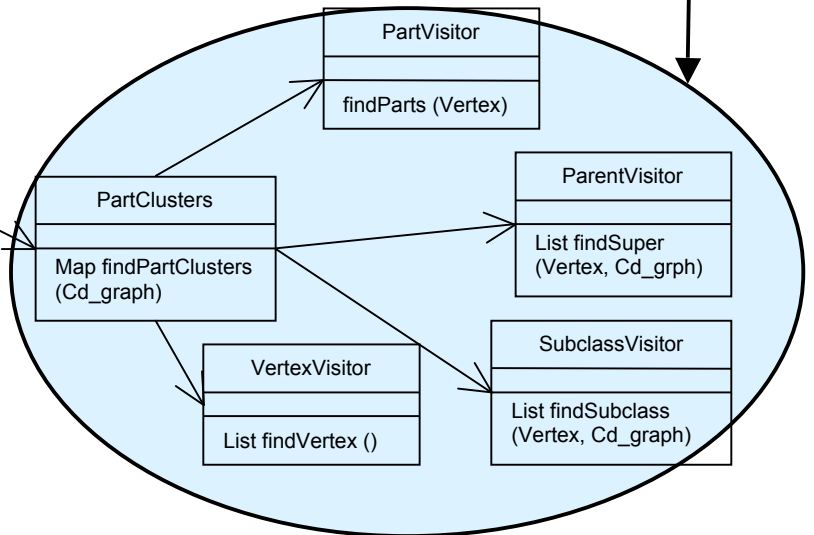
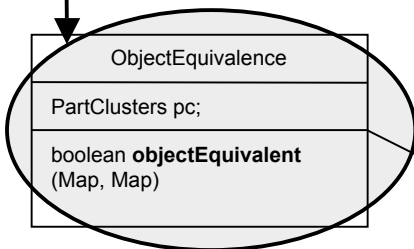
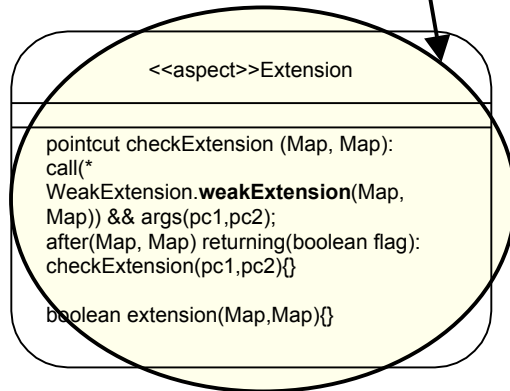
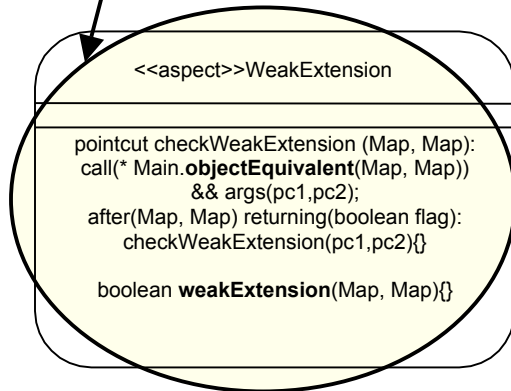


Figure 3: Mapping from Requirement Phase to Implementation Phase

**Figure 4** shows the WeakExtension aspect. It defines a pointcut checkWeakExtension that intercepts the call to the method objectEquivalence in the class Main. It examines the return value of the method in an after advice. Depending on the return value, it checks for weak extension between the two class dictionary graphs reusing the hash maps computed in the method objectEquivalence. Similarly, **Figure 5** shows the aspect Extension with its pointcut checkExtension. It checks the return value of method weakExtension and compares the class dictionary graphs for extension if required.

```
Aspect WeakExtension{

    pointcut checkWeakExtensions(Map pc1, Map pc2):
    call(* Main.objectEquivalent(Map, Map)) && args(pc1,pc2);

    after(Map pc1, Map pc2) returning(boolean flag):
    checkWeakExtensions(pc1,pc2){
        if(!flag)
            if(weakExtension(pc1, pc2))
                System.out.println("The Class Dictionary 1 Weakly Extends
                Class Dictionary 2");
    }
}
```

Figure 4 Pointcut and advice for Weak Extension

```
Aspect Extension{

    pointcut checkExtensions(Map pc1, Map pc2):
    call(* WeakExtension.weakExtension(Map, Map)) && args(pc1,pc2);

    after(Map pc1, Map pc2) returning(boolean flag):
    checkExtensions(pc1,pc2){
        if(!flag)
            if(!isSubSet(pc1.keySet(),pc2.keySet()))
                System.out.println(" There is no exention relationship between
                the class dictionaries");
            else
                if(extensionRelation(pc1,pc2))
                    System.out.println("The Class Dictionary 1 Extends Class
                    Dictionary 2");
            else
                System.out.println("There is no exention relationship
                between the class dictionaries");
    }
}
```

Figure 5 pointcut and advice for Extension.

## 4.5 Extending the existing system to support more complex input using AOP

### 4.5.1 Optional Parts

The class dictionary is extended to include optional parts in the input class dictionary graphs. A part label can be either required or optional; meaning optional parts do not have to be present during an instantiation of a class declaring optional parts. An optional part is represented as a labeled vertex enclosed in square braces '['']. The definitions of object-equivalence, weak extension and extension are modified to check if the corresponding parts have the same type. This extension to the existing system to support more complex input is incorporated by adding aspects in the program thereby without modifying the existing code. The class dictionary is modified to add the textual notation of optional part as shown in by the highlighted lines below:

```
Cd_graph = <adjacencies> List(Adjacency).
Adjacency = <source> Vertex <ns> Neighbors ". ".
Neighbors : Construct_ns | Alternat_ns common <construct_ns> List(Any_vertex) .
Any_vertex : Labeled_vertex | Optional_vertex.
Optional_vertex = "[" <optional_name> Labeled_vertex "]" .
Labeled_vertex = "<" <label_name> Ident ">" <vertex> Comma_list(Vertex).
Alternat_ns = ":" <alternat_ns> Bar_list(Vertex) [<common> Common].
Common = "common".
Construct_ns = "=".
Vertex = <vertex_name> Ident.

List(S) ~ {S}.
Comma_list(S) ~ S {"", " S}.
Bar_list(S) ~ S { "|" S}.
```

Even though the class dictionary is modified no modifications are required in the traversal strategies. The traversal strategies are structure-shy. They define the traversal path for the visitors on the input class dictionary graph by specifying only the key path nodes. In this case, the optional part addition in this class dictionary did not require any traversal strategies to be modified. This emphasizes the robustness of the Demeter method and adaptive programming.

The application logic that computes and compares the part clusters is modified to support optional parts. The aspect `OptionalParts` is responsible for providing this added support. It intercepts the execution of methods that compute the part clusters and checks for extension relationships between class dictionary graphs. Around advices contain the behavior to handle the optional vertices which is executed instead of the base code. With the use of AOP techniques this system extension is possible without modifying the base code at various places but having the optional parts functionality localized in the aspect.

### 4.5.2 Problems

1. The execution flow of the system is difficult to comprehend as the base code is oblivious of the pointcut and advice. Due to around advice, the code that actually gets executed is in the aspects and not in the base classes. This also causes to lot of 'dead code' that never gets executed.
2. Even though in the `OptionalParts` aspect each use case is a separate advice, thus claiming the use case modular implementation; it results in tangling of optional part concern with the business logic of the use cases. The aspect `OptionalParts` redefines the entire functionality of each use case with the desired modifications.

To illustrate the tangling of optional part concern with use cases logic consider the following code segment:

```

1. Aspect WeakExtension{
2.     boolean weakExtension(Map pc1,Map pc2){
3.         for(Iterator it = pc1.keySet().iterator(); it.hasNext();){
4.             :
5.             :
11.             for(Iterator iter = partClusters1.iterator(); iter.hasNext();){
12.                 lb1 = (Labeled_vertex)iter.next();
13.                 boolean labelFound = false;
14.                 int j = 0;
15.                 while(!labelFound) && j < partClusters2.size()){
16.                     lb2 = (Labeled_vertex)partClusters2.get(j);
17.                     j++;
18.                     if(lb1.label_name.equals(lb2.label_name))
19.                         labelFound = true;
20.                 }
21.                 if(!labelFound)
22.                     return false;
23.             }
24.             :
25.             :
60.     }
99. }

```

Figure 6: Base Code for Weak Extension

```

1. Aspect OptionalParts{
2.     pointcut checkWeakExtension(Map pc1, Map pc2): execution(*
3.     weakExtension(Map, Map)) && args(pc1, pc2);
4.     boolean around(Map pc1, Map pc2): checkWeakExtension(pc1, pc2){
5.         :
6.         :
10.         for(Iterator iter = partClusters1.iterator(); iter.hasNext();){
11.             anyVer1 = (Any_vertex)iter.next();
12.             flag1 = anyVer1.isOptional();
13.             boolean labelFound = false;
14.             int j = 0;
15.             while(!labelFound) && j < partClusters2.size()){
16.                 j++;
17.                 anyVer2 = (Any_vertex)partClusters2.get(j);
18.                 if(anyVer1.getLabel().equals(anyVer2.getLabel())){
19.                     flag2 = anyVer2.isOptional();
20.                     if(xor(flag1, flag2))
21.                         return false;
22.                     labelFound = true;
23.                 }
24.             }
25.             if(!labelFound){
26.                 return false;
27.             }
28.         }
29.         :
30.         :
65. }}

```

Figure 7: Optional Part aspect: Advises an advice from the WeakExtension aspect

The method `weakExtension` takes in two hash maps each containing the part clusters of the input class dictionaries. Part clusters of a vertex are stored in a hash map as a list of labeled vertices. Taken two vertices with the same name, the weak extension method iterates over their part cluster list to compare the labels of the corresponding parts (**Figure 6**; lines 12, 16, 18). As we extend the ‘Weak Extension’ use case to support the optional parts the implementation changes in the following manner:

The part cluster list now contains `Any_vertex` (parent of `Labeled_vertex`, `Optional_vertex`). It iterates over the part cluster list and retrieves the `Any_vertex` and checks if it is optional (**Figure 7**; lines 11, 12) then it iterates over the other part cluster list from the second class dictionary graph and finds an `Any_vertex` with same label (**Figure 7**; lines 15, 16, 17, 18). If such label is found it checks if they both match in type i.e. required / optional (**Figure 7**; lines 19, 20). We observe that the code addressing the optional part concern is tangled in the business logic of the ‘Weak Extension’ use case and present AOP technology is not sufficient to separate the optional part concern. With current AOP constructs we need to use around advice to modify behavior that causes repeated code in aspect definitions leading to dead code in the base program. More fine grained joinpoints that capture Type Casting, Assignments and Control Statements are required to avoid code repetition in the base and the aspect definition. e.g. in **Figure 6** line 12, the Type Casting to `Labeled_vertex` could be intercepted with a fine grained joinpoint and around advice can be used to type cast it to `Any_vertex`. Similarly, if we can capture the ‘if’ statement (**Figure 6** line 18) using a fine grained pointcut and give it an around advice to compare any\_vertices (**Figure 7** line 18) we can reuse the remaining of the code of the ‘`weakExtension`’ method. Thus we can avoid the around advice to the entire method and modify only those lines of the method that required to be changed for addressing the corresponding change to the new system requirements.

#### 4.5.3 Observation:

AspectJ style aspects work well for adding new use cases that don’t modify the underlying data structure. Demeter style aspects work well for adding new use cases that will later be generalized to handle more complex inputs.

Approach	Works well	Does not work well
AspectJ	If input data structure is not modified.	If modified input data structure as it crosscut several use cases: AspectJ’s coarse-grained pointcut model leads to code duplication.
DemeterAspectJ (DAJ)	If class dictionary changes in a way that does not need traversal strategies to be modified and adaptive methods still work for modified inputs. Adding new adaptive methods for new use cases.	If we need around advice for existing code. Current implementation of DAJ does not support around advice.

#### 4.5.4 Conclusion:

AOP is a key in the implementation of Use Case Modular software. It provides the ability to map each use case from system requirements to a separate implementation module. It also has capability of capturing specific points in the execution flow of base use cases advising the code there by adding extra behavior or modifying the existing behavior.

It provides the ability to extend the behavior to support additional functionality or handle more complex input. Aspects are very effective for some extensions to the system but they can have an adverse effect on system modularity when used for some other modifications.

Modifications can be categorized as:

1. ***Additions to the existing functionality:*** this can be viewed as adding a new use case in the system. Aspects can be used to weave this new functionality with the existing one. Aspects help us to add the functionality without modifying the existing code and also keep the new functionality as a separate module. Aspects are of great help due to their intertype declaration capabilities that are often required for adding completely new behavior. Occasionally, adding completely new functionality can be accomplished by using standard OO techniques without using aspects but that is possible only if the new module does not crosscut any other existing modules.
2. ***System extensions that affect all the existing functionality:*** e.g. supporting different input data types. These modifications affect all the use cases. They require the implementation of all the use cases to be modified. With existing AOP technology this brings the effect of rewriting code with added features (e.g. around advice). Using aspects and rewriting code in advice hampers the modular structure. Instead of separating the concerns it tangles all the use cases with the extended functionality. However, the alternate way to incorporate these modifications is to change the code addressing each use case. This maintains the original use case modularity but scatters the concern addressing the extended functionality. Thus, we can conclude that with help of AOP, we can improve system modularity as compared to that with OOP but it does not solve the problem of cross-cutting completely. As we design the system to separate the use cases, the data structure concern is scattered in all modules and requirement modifications referring to the input data structure, are difficult to separate in a module with existing AOP techniques.

## 5 Case Study 2: Library System

### 5.1 Introduction

A simplified library system is used as another case study to test the effectiveness of AOP in use case modular software development. This case resembles more to a typical software system than the previous case in terms of functionality, complexity and features. In this case the Library system undergoes a series of changes in requirements specification after the first iteration of implementation is complete. The library system is then tested extensively for modularity against a series of changes each modifying a different concern. While the system evolved with the changes in the requirements the corresponding implementation modifications tried to maintain the use case modularity of the system.

### 5.2 Iterations:

- **Iteration 1:** The library system consists of Books, BookCopies, Users and Operations. The books and users are stored in the library system in the form of lists. Operations supported: AddBook, RemoveBook, AddUser, RemoveUser, AddBookcopy, RemoveBookcopy. These operations are only the Administrative operations.
- **Iteration 2:** More operations (e.g. CheckInBook, CheckOutBook) were added in the second iteration of the system. Two aspects CheckInOperation and CheckOutOperation were used to add these operations in the system as these operations modified the Book, BookCopy and User classes in terms of added behavior.
- **Iteration 3:** The library system was modified to authenticate every user with a username and password. The operations were categorized as 'Member' or 'Administrator'. Depending upon the user's type (member or administrator) certain operations were allowed to be performed by the user. Library system was modified to recognize logged-in users. An Authentication aspect and a helper class Display were added.
- **Iteration 4:** Persistence was added to the library system, using MySQL to store information about books, users, operation types and authentication information. The accessor methods for all data were modified using the Database aspect to retrieve and modify data in the database, accordingly.

### 5.3 Design and Implementation

#### 5.3.1 Use Case Diagram (Iteration 3)

Figure 8 shows the use case diagram for the Library system (iteration 3). This use case diagram shows two kinds of actors: Members and Administrators. A member can perform only two operations: CheckInBook and CheckOutBook. The administrator in addition, can also perform operations like Add/RemoveBooks, Add/RemoveUsers, etc. The 'Authentication' use case has the responsibility of validating a user and categorizing it as member or administrator. It has three extension points. One for member users; the extending use case displays member's menu. The second for administrators; the extending use case displays the administrator's menu. The third for invalid users, the extending use case displays appropriate message and exits. The show member menu and show admin menu use cases also have extension points that invoke the appropriate use cases depending upon the choice made by the user.

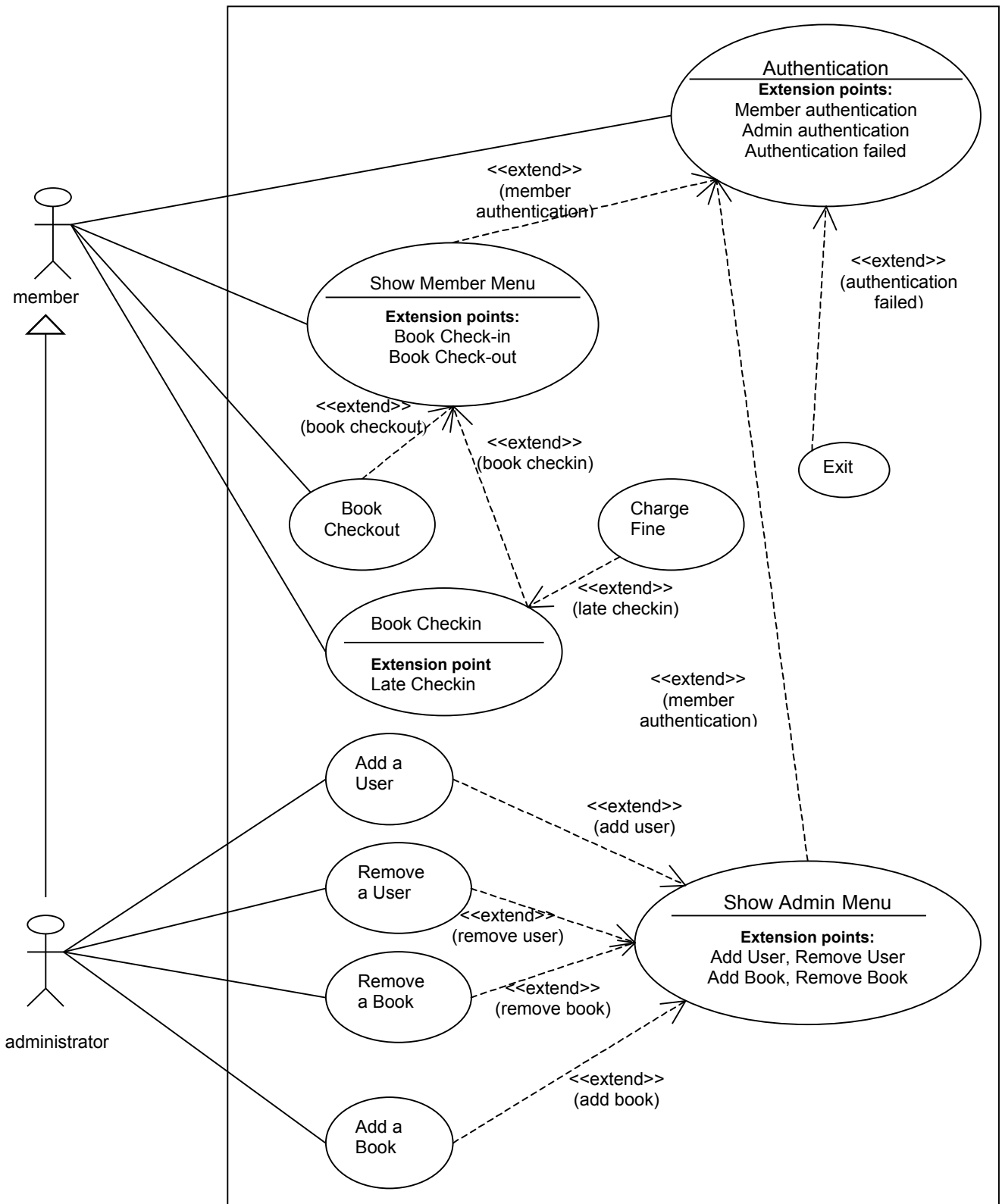
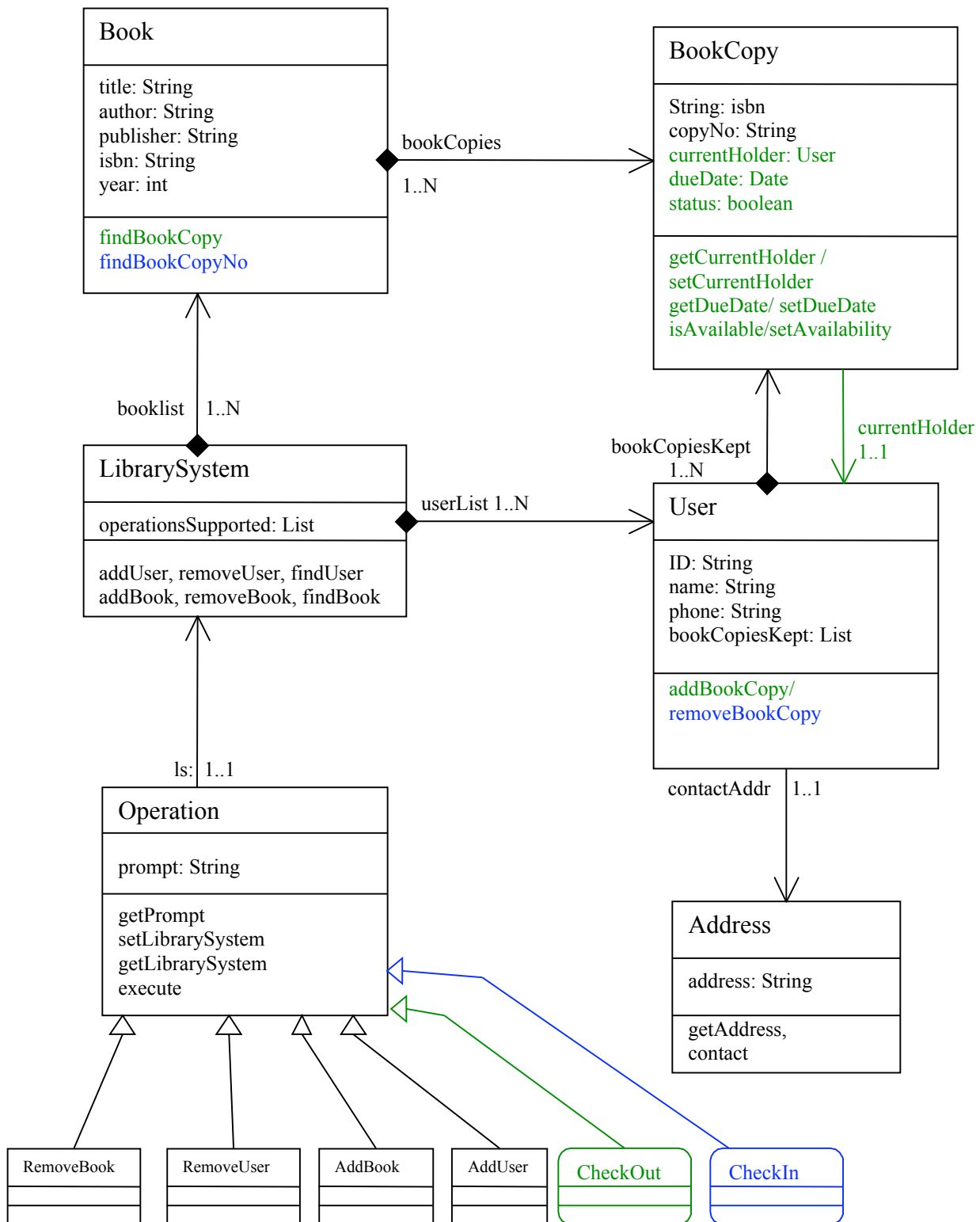


Figure 8: Use Case diagram for Library System (Iteration 3)



**Figure 9: Partial Class Diagram for Library System showing the basic classes.**

(Without implementation constructs for Authentication and Persistence.)

### **5.3.2 Class Diagram (Iteration 2)**

The class diagram in Figure 9 shows the major classes and the associations in the iteration 2 of library system. Observe that all the operations are subclasses of class Operation. Each of them defines the body for the abstract method 'execute()' in class Operation. This method is responsible for the interaction between the user and the library system and initiates further steps to perform an operation. CheckIn and CheckOut aspects are used to implement the modifications of iteration 2. Aspects help us to separate crosscutting concerns and modify / extend the behavior without physically changing the existing implementation. E.g. The operation CheckOut requires a bookCopy to store information like DueDate, CurrentHolder, Availability. Though these fields and their accessor methods are members of the class BookCopy they logically belong to the CheckOut concern. Using intertype declarations in AspectJ, the aspect CheckOut contains these members thus modifying data and behavior of the class BookCopy without actually modifying it and also maintaining CheckOut use case as a separate module.

### **5.3.3 Authentication Feature (Iteration 3)**

The authentication feature was later added on top of all the existing operations. This is used to authenticate a user as a member or administrator against a username and password. This additional feature modified the original control flow as authentication is now the first step for using the library system. The effect is similar to appending extra functionality to each use case. In the implementation phase an aspect is used to incorporate the functionality.

Figure 10 below shows the mapping between the Authentication use cases and its implementation. The authentication aspect is responsible for the user interface for entering a username and password, validation of the entered information and performing the necessary steps so that the system recognizes the logged in user. The responsibility of the user interface is delegated to the Display class which is associated with authentication aspect. The authentication use case has two extension points each describing the system's response based on the result of the authentication. Partial or complete menu is presented based on the type of user (member or administrator). The aspect MenuCustomizer is responsible for displaying the correct menu. This aspect iterates through the list of operations available in the system and depending upon the privileges of the logged in user, creates a list of operations to be displayed in the menu. Both the use cases ShowMemberMenu and ShowAdminMenu are implemented in a single aspect as shown in Figure 10.

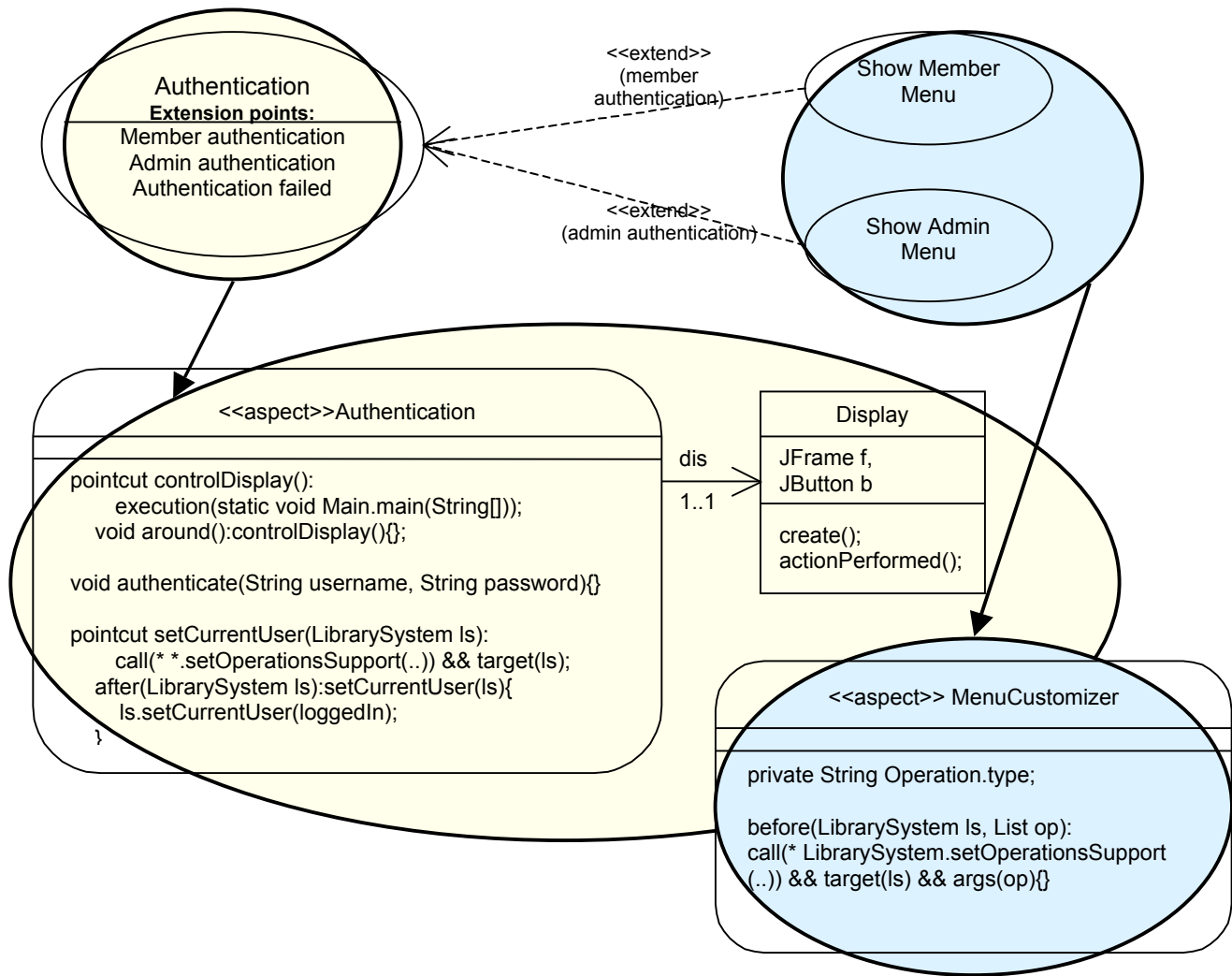


Figure 10: Mapping from Requirements phase to Implementation Phase

#### 5.4 A test for Use Case Modularity

We observe that in this use case modular implementation, user interaction for each use case is implemented in a separate module but the use cases are not separate all the way to the underlying building blocks of the library system viz. classes `LibrarySystem`, `Book`, `User`, `BookCopies`. All the use cases appear tangled in these components of the system. Each class has several methods, each addressing functionality from a different use case. E.g. in class `LibrarySystem` we find methods like `addBook`, `removeBook`, `addUser`, `removeUser` each is logically a part of a different use cases. As these methods are in a single class the use cases might appear as ‘tangled’ but each operation is implemented as a separate method hence, which code addresses which use case is easy to determine. Also, it is better than the case where different lines of a method address multiple use cases leading to severe tangling. We tested the use case modularity of the Library System by conducting a simple test. The operation `Remove Book` was modified in the requirements space after iteration 3 was complete.

#### **5.4.1 Original use case RemoveBook – Requirement’s Analysis Phase**

*Main flow of events:* The use case starts when the user selects to remove a book from the menu.

The system prompts the user for ISBN of the book to be removed. The system checks for the book and its copies and deletes the book from the library if there are no copies existing. The system displays the successful completion of the operation and terminates.

*Exceptional flow of events:* Book with entered ISBN is not found. System displays appropriate message and no book is removed.

*Exceptional flow of events:* The book has copies existing in the library system. The system displays a message directing the user to remove the book copies first. No book is removed from the library.

#### **5.4.2 Modified use case Remove Book – Requirement’s Analysis Phase**

*Main flow of events:* The use case starts when the user selects to remove a book from the Menu.

The system prompts the user for ISBN of the book to be removed. The system checks for the book and book copies. A message indicates that all the book copies will also be removed if this operation is continued and prompts the user to confirm the proceed. If the user enters ‘yes’ all the book copies and the book are removed from the library.

*Exceptional flow of events:* Book with entered ISBN is not found. System displays appropriate message and no book is removed.

*Exceptional flow of events:* User chooses to discontinue the operation at the second prompt. No book or copies are removed.

*Exceptional flow of events:* The operation fails as one or more copies are checked out of the library. The system displays appropriate message and no book or book copy is removed from library.

#### **5.4.3 Modifications – Implementation Phase**

1. The method RemoveBook.execute() is responsible for the user interaction for the use case. It was modified to incorporate changes in the user interface. The method was changed directly for inserting the code segment for added interaction. An aspect also can be used to give around advice to the existing execute() method that rewrites its functionality with added user interaction. The later approach was not chosen to avoid code duplication.
2. LibrarySystem.removeBook(Book) is modified to iterate over the list of copies to find if any copy is checked out. If no copy is checked out it initiates removal of all the copies and then the book itself. If a copy is checked out no copy or book is removed. This method was also modifies directly without using an aspect for simplicity.

#### **5.4.4 Conclusion of the test**

We observe that modifications in one use case resulted in changes to code at two places i.e. use case modularity is not preserved through all the lower level constructs of the system. Use cases refer to the ways of using a system, but they have to share the building blocks of the system e.g. Books, Users, BookCopies, LibrarySystem etc. With the present technology, the use case can not be separated as the lower level of system design. Though a class contains code that is executed for multiple use cases (e.g. class LibrarySystem with methods addBook/ removeBook, addUser/ removeUser), it is separated in different methods. Thus which code segment addresses which use case is clear and modifications in a use case can be implemented without modifying code addressing other use cases. We can achieve sufficient use case modularity to accomplish maintainability in changing business requirements.

## 5.5 Adding Persistence to the Library System

An attempt to extend the Library System to support persistence.

Database: MySQL; JDBC Driver for MySQL: MySQL Connector/J 3.0.7 Stable

### 5.5.1 Persistence Concern

Database stores the registered users and books in the library system. Also it stores information about which book copy is available and which is checked out by a user. If a copy is checked out, the user who checked it out and the due date. Figure 11 below shows the database tables:

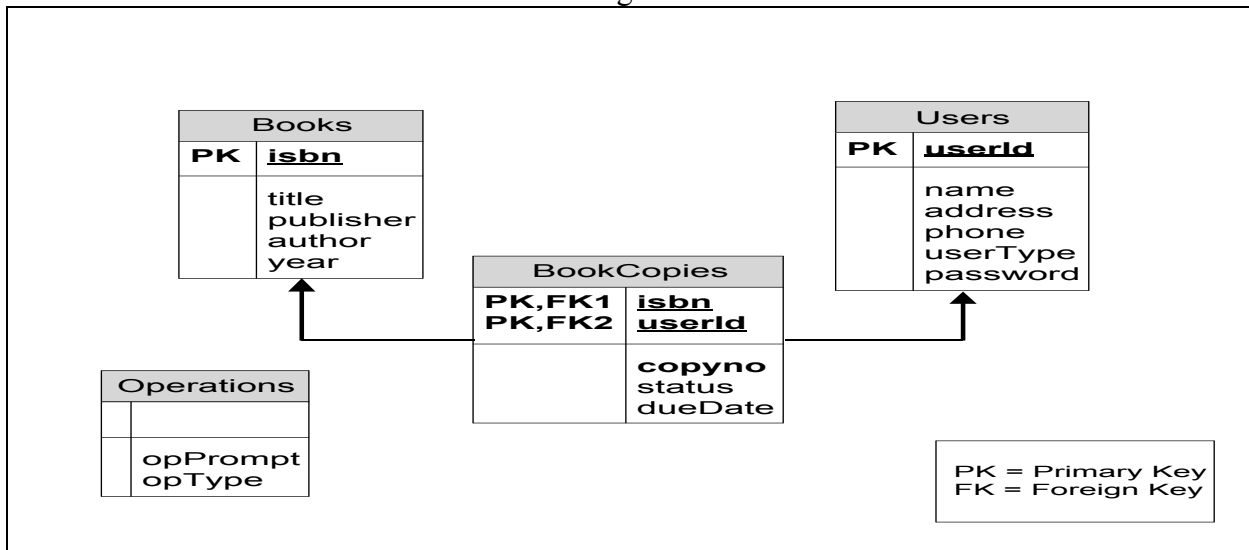


Figure 11: Library System Database Schema

The main task is the Object- Relational mapping between persistent data and transient data. All the use cases change the state of the library system in terms of books, book copies available or members. These changes are reflected to the persistent state by the database concern thus it crosscuts all the use cases. We deploy an AOP approach to capture and implement persistence due to its crosscutting nature.

The Database aspect contains all the database related code: making a connection, query or update and is responsible for following tasks:

1. Retrieve book, book copy and user information from the database, construct the appropriate objects when the application tries to refer to these objects.
2. Changes made to the runtime instances are propagated to the appropriate entries in the database system.
3. We have two types of operations: member and administrator. Member operations are performed by all users and administrator operations are performed only by administrators. Operation type is stored in the database. It is used to secure operations that can be performed only by privileged users.

The base implementation is oblivious of the persistence aspect hence the library system application does not know that the data is now stored in the database versus in memory as in the original implementation. The class LibrarySystem is responsible for maintaining the books and users. The aspect intercepts the methods of this class and runs SQL queries and updates on the database to reflect the changes in the library system according to the various use cases.

### 5.5.2 Problems

1. **Possibilities of inconsistencies:** the Object-Relational mapping is done corresponding to data accessed by each operation e.g. there are no operations to change address of a member so there is no mapping provided for the method `user.setAddress()` in the database aspect. Thus, as more operations are supported by the application, the database aspect will also have to be extended to support those functions.
2. **Braking the cycle:** The transient data structures have cycles in them. e.g. a book has a reference to its copies. Each book copy has a reference to the user who checked it out (if the copy is not available). A user has a reference to all the copies he has checked out. With persistence added, to create a book instance by retrieving information from the database first we read from the table “Books”, then create all the book copies reading from table “BookCopies”, put them in a list and add the list in book. For creating a book copy we need to first create the user who checked it out. To create a user we need to create all the book copies he has checked out. Thus creating a book can result into a cycle of object creation. This has the potential of bringing the entire persistent data into transient memory. To avoid this, we had to break the cycle by assigning “null” to some of the references. This adds the extra responsibility of keeping track of the null objects and protecting the application from throwing exceptions.
3. **Modifications to the base code:** Some method signatures were modified. E.g. `LibrarySystem.addBook(Book) / removeBook(Book)`, etc. were originally declared to return void but they were modified to return a boolean value indicating the success or failure of the requested operation. In the original implementation, the methods accessed the transient collection data objects locally and report any failure to the user. With persistence, the methods had to be modified to keep the database aspect off the responsibility of indicating success or failure of the operation to the user.
4. **Limitation for use of Java Reflection API:** Two operations `CheckInOperation` and `CheckOutOperation` were added to the library system at a later stage of development and aspects `CheckIn` and `CheckOut` were used to introduce this additional behavior respectively. After compilation with `ajc 1.0.6` the class files found were `CheckIn$CheckInOperation.class` and `CheckOut$CheckOutOperation.class`. The Java Reflection API could not be used to instantiate `Operation` by reading names from the database (`Operations` table) as the constructors in these classes were still named as `CheckInOperaiton` and `CheckOutOperation` respectively.
5. **Tangling of Database concern with application logic:** Database aspect intercepts methods that change the state of the data in the system. It mainly uses `around` advice to replace the actual method body with the advice body. The advice manipulates the database where the original method manipulates the application data. For methods that are responsible for functionality other than manipulating the application data, the `around` advice also rewrites that functionality resulting in database concern tangled with all the other concerns. This is due to the coarse grained pointcuts in AspectJ. If finer grained pointcuts were available to capture `Type Casts`, `Assignments` or `Control statements` method behavior can be partially modified without rewriting the entire code in an advice. This also reduces dead code in the system.

### 5.5.3 Future Work:

1. All the database related functionality is localized in the Database aspect but it has several tangled concerns like connecting to the database, SQL retrievals, updates, instantiations, exception handling in each advice code. Deployment of the persistence framework [5] will better modularize the Database aspect for its reusability and extendibility e.g. changing the database management system to another vendor will require modifications to only the Connection portion of the deploying aspect versus all the advice code of the current implementation of the persistence aspect.

## 6 Conclusion

This project evaluates the effectiveness of AOP for Use Case Modular software development. It performs two case studies, Class Dictionary Graph Extension and Library System. The first one is theoretical and simpler as compared to the second which is closer to a typical software system in terms of data, complexity and user interface. Both systems undergo iterative development cycles with changes of various severities applied to the business requirements in each iteration. For each system, use cases and their relationships are realized in the requirements phase. The base use case(s) is (are) implemented using collaboration of classes. The extending use cases are implemented using aspects in AspectJ in later iterations of the system development phase. AOP is a key in this iterative use case driven development where each use case is implemented as a separate implementation module. It provides an ability to extend system behavior to support additional functionality or handle more complex data. New use cases can be easily added to the system using aspects, also existing use cases can be modified maintaining the use case modularity provided the modification does not change cross-cutting concerns. Though the software is modularized according to the use cases, the underlying building blocks of the system like Data, Data structure, Syntax of input data etc are shared by all the use cases. In cases where modifications to system requirements cause changes to these shared entities, current AOP techniques fail to maintain the use case modularity. The aspect(s) addressing this change gives around advice to methods from the multiple use case modules and rewrites their functionality with modifications. It results in an implementation with problems like repeated code in aspects, dead code in base modules and multiple use cases tangled with the modified concern in the aspect. For such modifications, AOP with finer grained join points can better maintain the use case modularity. The finer gained joinpoints capable of intercepting statements like TypeCasting, Control Statements and Assignments can modify specific code lines that need modifications and avoid around advice to the entire method.

## 7 Acknowledgements

I thank Prof. Karl Lieberherr for the support, direction and resources for this project. Special thanks to Therapon Skotiniotis for his valuable comments and suggestions through out this project and Pengcheng Wu for the implementation foundation of the Library System.

## 8 References

1. Grady Booch, James Rumbaugh, Ivar Jacobson, “The Unified Modeling Language User Guide”. Addison-Wesley 1998.
2. Ivar Jacobson, “Use Cases and Aspects – working together”. Invited Talk at AOSD 2003, Boston, USA.
3. Karl J. Lieberherr, “Adaptive Object Oriented Software The Demeter Method with Propagation Pattern”. PWS Publishing Company, Boston, 1996.
4. DAJ Homepage. URL: <http://daj.sourceforge.net/>
5. A. Rashid and R. Chitchyan, “Persistence as an aspect”, AOSD, 2003, Boston, USA.
6. Elrad, T.; Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H.: “Discussing Aspects of AOP”. *Communications of the ACM* **44(10)**, pp. 33–38, October 2001.