THIRD TURKISH ASPECT-ORIENTED SOFTWARE DEVELOPMENT WORKSHOP PROCEEDINGS

December 26, 2008

Ankara, Turkey

Bedir Tekinerdoğan (ed.)

Department of Computer Engineering

Bilkent University

06800 Bilkent, Ankara, Turkey

Table of Contents

Preface

Aspect-oriented software development (AOSD) is an advanced technology for separation of concerns, which provides explicit concepts to modularize the crosscutting concerns and compose these with the system components. During the last years several international conferences and workshops have been organized around this important topic. In Turkey, the First National workshop on Aspect-Oriented Software Development Workshop (TAOSD) has been organized in June 2003. The second TAOSD workshop has been organized in September 2007. This is the report of the Third TAOSD workshop which has been organized at Bilkent University in December 2008.

This third workshop has been organized within the AOSD (graduate) course that is given at Bilkent University. The course provides an in-depth analysis of AOSD and teaches the state-of-the-art AOSD techniques. One of the basic requirements for fulfilling the requirements of the course was an aspect-oriented software development project. Students had to form groups of three and select complex cases from an industrial project or ongoing academic projects. These cases were analyzed for crosscutting concerns and refactored to an aspect-oriented implementation. This resulted in a unique collection of valuable aspect-oriented case studies which shows the strengths and weaknesses of AOP. Because of this unique experience and the interesting results that came out of these projects we decided to organize this workshop to make these valuable practices public for a broad audience. The project results have been presented as workshop papers in this proceedings of the TAOSD workshop. Because most students in the AOSD course wrote a workshop paper for the first time they got a course on how to write workshop papers and got extensive feedback on their papers. We think that the results of these projects are of interest to a broader public and as such would like to share our experiences in this workshop. The primary goals of this workshop were (1) sharing knowledge and improve consciousness on AOSD, and (2) Stimulate research and education on AOSD in Turkey.

About 60 participants both from industry and academic institutes have registered for this workshop. The program included four presentations sessions and one invited talk that was presented by Dr. Semih Çetin from Cybersoft. In total we had 9 presentations, each of them took 30 minutes including a demonstration of the aspect-oriented implemented systems, and questions from the audience. The workshop was concluded with a plenary session in which we summarized the workshop results.

We think that the papers in this workshop proceeding are both useful for novice software developers and researchers who wish to do research on AOSD topics. The papers present a broad range of aspects in different applications, discuss the crosscutting problems and as such provide a unique set of aspect examples. In addition, the papers also highlight the experienced obstacles in AOSD and provide some triggers for new research directions in AOSD.

The workshop had the same format as the workshop in 2003. Again, the unique character of this workshop was based on two factors. First of all, the presenters of the workshop were students and not, as in a usual workshop setting, experienced researchers. Second, the audience consisted mainly of persons who were interested in AOSD but had not used it before.

This event would not have been possible without the help and support of many people. First of all I would like to thank the Department of Computer Engineering, Bilkent University for actively supporting this event. I would also like to thank Semih Çetin for his active involvement during the workshop and giving an interesting invited talk about their industrial project in Turkey which has applied an aspect-oriented software development approach. Further, I would like to thank all the participants to the workshop for sharing this unique event with us. Of course, the workshop would not have been successful without the enthusiasm and the hard work of the CS586 Aspect-Oriented Software Development Course students. I would like to thank them one by one: Cem Aksoy, Hidayet Aksu, M. Uğur Aksu, Eren Algan, Doğan Altunbay, Damla Arifoğlu, Özgür Bağlıoğlu, Faruk Belet, Mehmet Esat Belviranlı, Doğan Kaya Berktaş, Mümin Cebe, Duygu Ceylan, Celal Çığır, Alptuğ Dilek, Ramazan Bertan Gündoğdu, Gizem Gürcüoğlu, Murat Kurtcephe, Mucahid Kutlu, Oğuzcan Oğuz, Kaan Onarlıoğlu, Mehmet Özçelik, Bahadır Özdemir, Sevil Sare Gül, Rıdvan Tekdoğan, Bahri Türel, Volkan Yazıcı, and Hilal Zitouni.

The workshop has been organized as en event of Bilsen – Bilkent Software Engineering Group, which has been recently founded at Bilkent University. The main goal of the group is to foster research and education on software engineering and support ongoing activities in these directions. This was our first event. More events will follow in the future.

Bedir Tekinerdogan
Bilkent University, Ankara, Turkey
January 2, 2009

Analyzing Aspects of a Gaming Application

Cem AKSOY, Mücahid KUTLU, Bahri TÜREL
Department of Computer Engineering, Bilkent University
Bilkent 06800, Ankara, Turkey
{cem, mucahid, bahri}@cs.bilkent.edu.tr

Abstract

Gaming applications can be considered as one of the complex applications in software development. This is because for developing gaming applications we have to deal with many different concerns. Traditionally, gaming applications are developed using general purpose or specific languages that cannot modularize these concerns with their proposed abstraction mechanisms. In this paper, we propose an aspect-oriented approach to modularize these crosscutting concerns by using the explicit language abstractions defined by aspect-oriented programming (AOP). We identify the key concerns in an object-oriented (OO) game, Chicken Invaders, and model these as aspects. We report our observations on aspect-oriented (AO) game programming and discuss the lessons learned.

1. Introduction

Gaming applications have a wide range of usage area. PC and mobile games are used very frequently. Technological developments bring improvements to game properties day by day. Developing graphical engines and artificial intelligence components cause the games to become more complex. These bear design issues in gaming applications [2].

The significant improvement in computer games started after usage of high level languages for game implementations. High level languages helped the programmers to modularize the concerns in the games, which causes more reusable and maintainable codes to occur in programs. In video gaming applications OpenGL also is a good example of modularization of concepts. Object-oriented programming (OOP) is the one of the most important steps in these improvements. Abstraction components called, classes are now used to separate concerns specific to games such as collision detection and resource management [2].

Separation of concerns as pointed by Dijkstra in [3] is separating one aspect from the others and seeing the other concerns as irrelevant from the separated concern's point of view. In programming, this is applied by modularization. Each concern should be mapped to a single module and each module should be coherent in itself. In gaming applications, the complexity rises from the big number of objects and the interactions among them. Current abstraction mechanisms like OO cannot fully modularize the concerns that crosscut over multiple modules. As an example, collision detection usually requires some operations to follow it which crosscut over all the objects which may collide.

In OOP, trying to accomplish all gaming tasks ends up in scattered concerns and tangling code. Also, game engine usually becomes a large object which is responsible for many different tasks. This decreases the coherency in the modules. We aim to overcome these deficiencies by applying AOP to specified game, named "Chicken Invaders". We define the concerns and modularize those using aspects and decrease the load of game engine.

The paper is organized as follows: Section 2 shows the case study for application of AOP in detail. Section 3 follows with the specification of the problem and identification of the game specific aspects. Section 4 describes and illustrates the aspects implemented with AspectJ for solution of the problem. Section 5 explains aspect-oriented solution with JBoss as an alternative approach for AOP and Section 6 briefly indicates the related work in the literature. Finally, Section 7 concludes by discussing the effects of AOP on the case study.

2. A Case Study: Chicken Invaders

In this section, we describe a gaming application named Chicken Invaders that we use as the case study for our work. A sample screen view of the game can be seen in Figure 1.

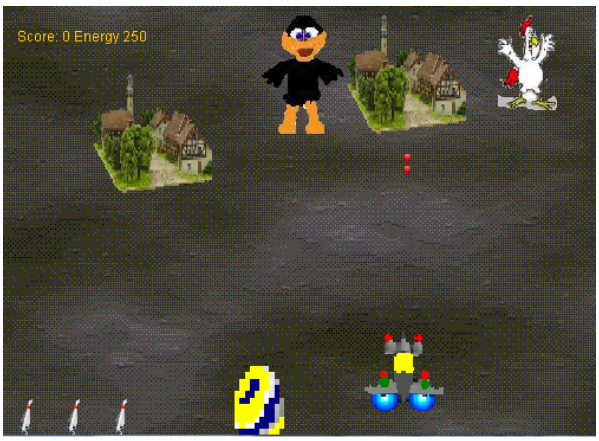


Figure 1. A snapshot from the game Chicken Invaders.

Chicken Invaders is a single player arcade game in which the player aims to complete the levels by destroying the enemy objects (chickens). The player uses an aircraft. The aircraft can use several weapons which have different destructive power and range. The enemy objects are classified into different categories such as ground, air and bonus objects. All these objects have different abilities, properties and vulnerabilities. Chickens come from above of the screen and move to down. The player tries not to have a crash with air objects and destroy as much as enemy objects in order to get a higher score. As the player shoots the enemy objects by using its bullets or bomb, objects may leave a bonus object for the aircraft which may heal the aircraft or give the ability to use a different type of weapon. The player also must be aware of the eggs of chickens which reduces aircraft's health. At the end of each stage, player has to defeat the boss enemy.

2.1. Object-Oriented Design

Chicken Invaders is a rich game in terms of objects so it is suitable for observing most of the properties of OOP paradigm.

In the proposed OO design of the game, there are two packages: GUI and GameComponents. GUI classes are responsible for user interface and GameComponents classes are GUI-independent representations of the game objects.

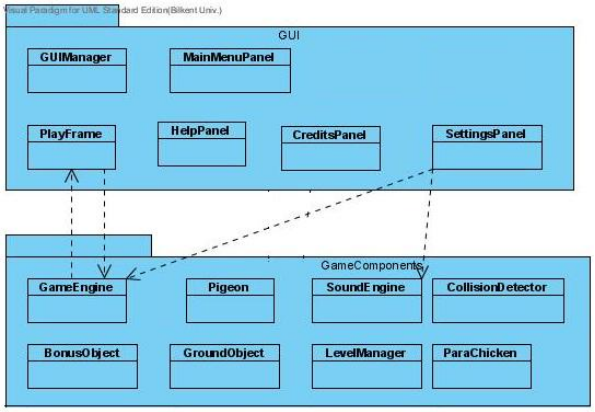


Figure 2. Package Dependency Diagram of Chicken Invaders, not including all classes.

As dependency details are shown in Figure 2, only a few classes from different packages depend on each other. During the game, main interaction between packages occurs between PlayFrame, GUI class of game play, and GameEngine. In each interval, GameEngine triggers PlayFrame to draw its objects. SettingsPanel changes static music/sound level attributes of SoundEngine and static game speed attribute of GameEngine. Thus, SettingPanel depends on those two objects.

As can be seen in class diagram of GameComponent package, Figure 3, GameEngine class is considered as the controller class. It has to control flow of the game and be connected to every class in order to co-ordinate them and provide connection between them. That is to say, GameEngine class has many responsibilities during the game which makes it the most complicated class. Responsibilities of GameEngine can be listed as the following:

- Getting new game objects from GameLevel class.
- Updating game objects positions.
- Checking if there is any crash between aircraft and chickens/bonus objects or between bullets and "ShootableObjects" by using CollisionDetector class.
- Running SoundEngine in proper times like after firing or any crash or sending rocket.
- Removing destroyed game objects and changing the score of the aircraft
- Removing game objects that are out of frame.

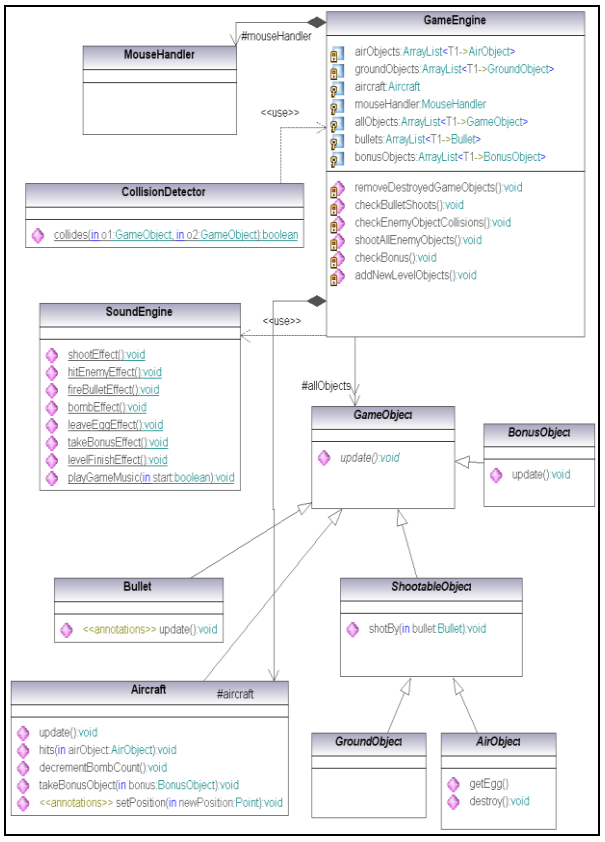


Figure 3. Class Diagram of Game Component Package. including important classes

- Setting bombs of enemy objects (eggs)
- Getting inputs from user via mouse.
- Setting speed of game and pausing the game.

All drawn objects on the GUI are part of a big hierarchy in which they all extend from a main class, GameObject. The intermediate level in this hierarchy consists of two classes, BonusObject and ShootableObject. These two are needed because of the different natures of two categories. BonusObjects cannot be shot by the aircraft, they disappear only when they move out of the frame or they collide with the aircraft. ShootableObject is also divided into two categories which are GroundObject and AirObject. As can be understood from the category names, the main distinction between these two categories is AirObject may collide with the aircraft where GroundObject cannot collide. They both can be shot by aircraft.

2.2. Applied Patterns

In OO design patterns play an important role to maintain easy development of software. In this section we describe design patterns which are used in Chicken Invaders.

2.2.1. Singleton Pattern. Singleton pattern is used to restrict instantiation of an object so that only one active instance of that class can be found on the system.

In Chicken Invaders, GameEngine is defined as a singleton object because it is only needed to be instantiated once. It has a lot of responsibilities over many classes, in other words it assures the game to flow. So as a coordinator, one GameEngine object is necessary and sufficient.

2.2.2. Visitor Pattern. The visitor pattern is used for separating operations from object structures. We can easily add new operations to those object structures without any modification.

In Chicken Invaders, we have applied visitor pattern to GameObject structure. In Figure 4, illustration of visitor pattern in Chicken Invaders is given. GameObjectUpdateVisitor and GameObjectShootVisitor are two visitor classes each of which implement GameObjectVisitor interface. GameObjectUpdateVisitor updates objects locations and GameObjectShootVisitor decreases health of visited object by a certain amount. By having visitor pattern, we can easily add new operations to GameObjects.

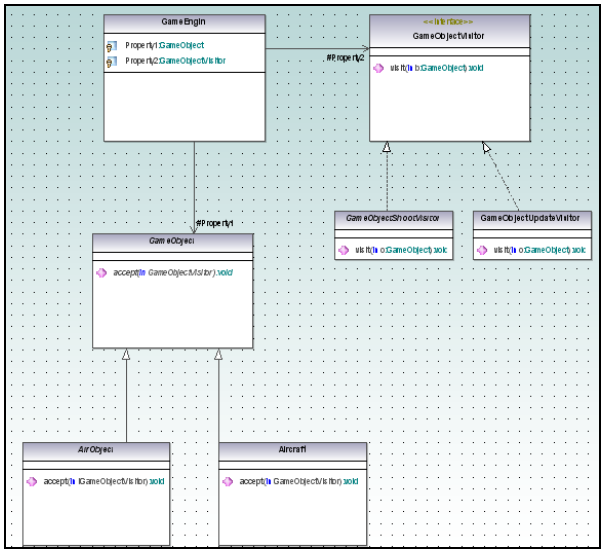


Figure 4. Illustration of visitor pattern in Chicken Invaders.

3. Identifying Crosscutting Concerns

In Section 1, we explained that in gaming applications, some concerns cannot be modularized and

they crosscut over multiple modules. Also, some objects may become non-coherent because of the big responsibility load. This results in a tangled code. In this section, we explore these problems encountered in OO design of Chicken Invaders.

3.1. Detecting Collisions

In Chicken Invaders, multiple objects may collide with each other, such as, aircraft and an air object or a bullet with a "ShootableObject". Detection of these collisions is significant for flow of the game.

These collisions are regularly checked by GameEngine after updating positions of all game objects. GameEngine uses `checkBulletShoots()` to detect collision of bullets and "ShootableObjects", `checkBonus()` to detect collision of aircraft and bonus objects and uses `checkEnemyObjectCollisions()` method to detect collision of aircraft and air objects. All these three methods use CollisionDetector class's `collides(GameObject,GameObject)` method. Handling concern of detection of collisions in GameEngine class causes low coherency for GameEngine which results tangled code.

Low coherency causes some maintenance issues such as finding the relevant part of code in a large code. For example, if all checks are completed by GameEngine it would be hard to find the relevant part in case of a change.

3.2. Removing Out-of-Frame Objects

Enemy objects, bonus packages move from up to down and bullets move from down to up. Whenever they get out of frame, we do not need to hold them in memory since we won't use them anymore. Therefore, after updating positions of game objects, except aircraft, we have to check whether they are out of frame or not. And the ones out of frame should be removed from memory.

GameEngine performs this action in its method `RemovedDestroyedGameObjects()`. This concern increases load of GameEngine.

3.3. Playing Sound

In Chicken Invaders, different sound effects are used for some specific actions like a crash, shooting an enemy object with a bullet or using a bomb etc. In addition, background music starts just after the game starts and continues to play during the execution time of the game.

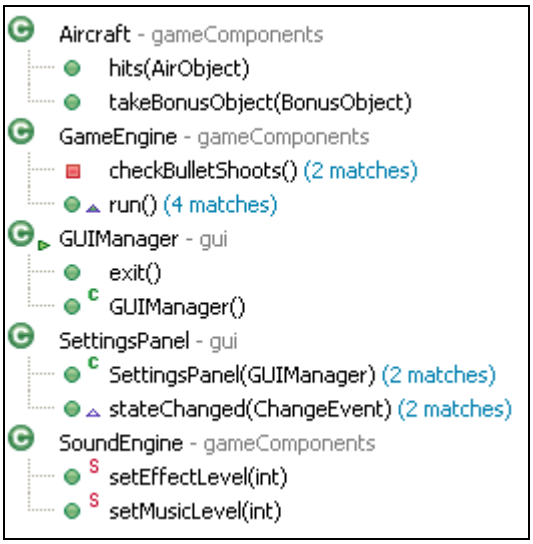


Figure 5. References to SoundEngine.

SoundEngine class coordinates all sound effects by using different methods for each type of sound. These methods are called from many different classes of the game which can be seen in Figure 5. Although, playing sound concern is modularized into a class, SoundEngine, because of these multiple calls from different classes, it becomes a scattering concern. These calls cause maintenance problems because in order to change a SoundEngine procedure, we need to visit all the modules that refer to that procedure. As a concrete example, to change `hitEnemyEffects()`, we may need to change `hits(AirObject)` method of Aircraft class. In simple changes it may not be required but for example if we want to completely remove sounds played after shooting enemy objects, we must change the corresponding code segment in `hits(AirObject)` method.

3.4 Throwing Bomb

In Chicken Invaders, aircraft throws a bomb which decreases health of all game objects that are currently in frame by a certain amount.

This concern is handled in GameEngine class with `shootAllEnemyObjects()` and in Aircraft class with `decrementBombCount()`. It is the responsibility of GameEngine to decrease health of all enemy objects following a bomb. All these decrements, like other tangling modules, reduces the coherency of GameEngine by increases its work load.

4. Aspect-Oriented Programming with AspectJ

In this section, we describe our aspect oriented solutions with AspectJ to crosscutting concerns which is mentioned in Section 3. In addition, two development aspects are described.

4.1 Detecting Collision Concern

To detect collision we need two advices, one for collisions of aircraft and air/bonus objects and other one is for collision of bullets and air/ground objects.

4.1.1 Collision of Aircraft and Air/Bonus Objects. The pointcut and advice for detecting collision of aircraft and air/bonus objects are shown in Figure 6. Pointcut selects joinpoints where position of aircraft is changed. By using `target()` and `this()` methods, we are able to get properties and functions of Aircraft and GameEngine classes. o In advice part, for each air object that is hold in GameEngine, we perform collision check(line 5). If there exists a collision, `hits(AirObject)` method of Aircraft is called (line 6) for decreasing health of Aircraft. Air object is destroyed (line 7). Same process is performed for bonus objects. Here, we call `takebonusObject(BonusObject)` method of Aircraft class (line 13) and remove the collided bonus object from memory(line 14-15).

```
1  pointcut updateInPositions( Aircraft a, GameEngine g):
2          target(a) && this(g) && call( void Aircraft.setPosition(..));
3  after(Aircraft a,GameEngine g):updateInPositions(a, g){
4      for (AirObject ao : g.airObjects){
5          if (CollisionDetector.collides(a, ao)){
6              a.hits(ao);
7              ao.destroy();
8          }
9      }
10     for (int bi = 0; bi < g.bonusObjects.size(); bi++){
11         BonusObject bo = g.bonusObjects.get(bi);
12         if (CollisionDetector.collides(bo, a)){
13             a.takeBonusObject(bo);
14             g.bonusObjects.remove(bo);
15             g.allObjects.remove(bo);
16         }
17     }
18 }
```

Figure 6. Pointcut and advice for collision of Aircraft and Air/Bonus Objects.

4.1.2 Collision of Bullet and Game Objects. The pointcut and advice for detecting collision of aircraft and air/bonus objects are shown in Figure 7. Pointcut selects joinpoints where position of bullet is changed. In advice, we check collision of bullet and for air and ground objects (line 21 and 26). If collision is found, we call `visit(GameObject)` method of GameObjectShootVisitor class(line 23 and 29).

```
17 pointcut bulletCollision(Bullet b, GameEngine g):
18     args(b) && this(g) && call(void GameObjectUpdateVisitor.visit(GameObject));
19
20 after( Bullet b, GameEngine g ):bulletCollision(b, g){
21     for (int i = 0; i < g.airObjects.size(); i++) {
22             if (CollisionDetector.collides(g.airObjects.get(i), b)) {
23                 visitor.visit(g.airObjects.get(i));
24
25             }
26     }
27     for (int i = 0; i < g.groundObjects.size(); i++) {
28             if (CollisionDetector.collides(g.groundObjects.get(i), b)) {
29                 visitor.visit(g.groundObjects.get(i));
30             }
31     }
32 }
```

Figure 7 Pointcut and advice for collision of Bullet and Game Objects.

By having these aspects, load of GameEngine is decreased. GameEngine does not deal with methods of Bullet objects and Aircraft. Interaction between CollusionDetector ends and so a better modularization is provided.

4.2 Removing Out-of-Frame Objects

The pointcut and advice for removing out-of-frame objects are shown in Figure 8. The pointcut selects joinpoints where position of any GameObject is changed. In advice, we get boundary of frame (line 6-

```
1  pointcut RemoveOutOfFrameObjects(GameEngine g,GameObject o) :
2      args(o) && this(g) && call ( void GameObjectUpdateVisitor.visit(..) );
3
4  after ( GameEngine g,GameObject o ) : RemoveOutOfFrameObjects(g,o){
5
6      double screenWidth = g.playFrame.getBounds().getWidth();
7      double screenHeight = g.playFrame.getBounds().getHeight();
8
9          Rectangle oRect = o.getRectangle();
10         if ((oRect.x + (oRect.width / 2)) < -10
11                 || (oRect.y + (oRect.height / 2)) < -10
12                 || ((oRect.y) - screenHeight) > 5
13                 || (oRect.x - screenWidth) > 5) {
14
15             g.allObjects.remove(o);
16             g.bonusObjects.remove(o);
17             g.airObjects.remove(o);
18             g.groundObjects.remove(o);
19         }
20 }
```

Figure 8. Pointcut and advice for Removing-Out-Of-Frame Objects.

7) and check if is out of frame (line 10) and remove the object from proper lists.(line 15-18). By having this aspect, we are now able to delete unused objects immediately. In bigger problems, deletion of these objects has higher priority because of filling memory space.

4.3 Playing Sound

There are several advices for playing sound aspect. The pointcut and advice for playing sound are shown in Figure 9. Advices that play sounds for firing a bullet, shooting an object, sending an egg, throwing a bomb, taking a bonus object are not shown for simplicity.

```
7   pointcut soundAircraft() : call (void Aircraft.hits(..));
8
9   pointcut aspectGUIManager() :  call (public gui.GUIManager.new() );
10
11  after():soundAircraft()
12  {
13      SoundEngine.hitEnemyEffect();
14  }
15
16  after() : aspectGUIManager()
17  {
18      SoundEngine.loadSounds();
19  }
```

Figure 9 Pointcut and advice for Playing Sound.

By having this aspect, we can easily manage playing sound concern like adding new sounds or canceling a sound without searching in codes. In addition, development gets easier because of better modularization.

4.4 Throwing Bomb

The pointcut and advice for throwing bomb are shown in Figure 10. The pointcut selects joinpoints where number of bombs of aircraft is decremented by method.

```
14  pointcut throwBomb(GameEngine e) :
15      this(e) && call ( void Aircraft.decrementBombCount() );
16
17  after(GameEngine e) : throwBomb(e)
18  {
19      for (int i = 0; i < e.airObjects.size(); i++) {
20          e.airObjects.get(i).shotBy(new Bomb());
21      }
22      for (int i = 0; i < e.groundObjects.size(); i++) {
23          e.groundObjects.get(i).shotBy(new Bomb());
24      }
25  }
```

Figure 10 Pointcut and advice for Throwing Bomb.

`decrementBomb()`. In advice part, we call `shotBy(Bomb)` method of each object in game(line 20-23). By having this aspect, load of GameEngine is decreased.

4.5 Development Aspects

Development aspects are used to help developers for understanding flow of software, such as tracing or logging. In this section, we describe two development aspects which we use to control flow of the program and detect the mistakes.

4.5.1 Checking Creation of Game Objects. Controlling creation of objects is significant for the developer since number of objects is high. The pointcut and advice for this concern are shown in Figure 11. Pointcut selects constructor joinpoints of each class which is a sub-class of GameObject. Advice method prints message for creation of the new object by using `thisjoinpoint`.

```
4  pointcut objectIsCreated(): call ( public GameObject+.new(..) );
5  after():objectIsCreated()
6  {
7      System.out.println(thisJoinPoint.toShortString()+" is created");
8  }
```

Figure 11. Pointcut and advice for Checking Creation of Game Objects

4.5.2 Checking Deletion of Game Objects. Controlling deletion of object is important in order to understand usage of memory. The pointcut and advice for this concern are shown in Figure 12. Pointcut selects joinpoints in calling `remove(GameObject)` methods in GameEngine class and RemoveOutOfframe aspect. In advice part, we give proper message about which object is deleted.

```
1  pointcut objectIsRemoved(GameObject g) :
2          args(g) && call(* *.remove(..)) &&
3          (within(gameComponents.GameEngine) ||
4           within(gameComponents.GamaAspect));
5  after(GameObject g): objectIsRemoved(g)
6  {
7      System.out.println(g.getClass() + " is removed");
8  }
```

Figure 12. Pointcut and advice for Checking Deletion of Game Objects.

5. Alternative Aspect Oriented Approach: JBoss

JBoss is a Java Aspect-oriented framework. JBoss and AspectJ both share similar capabilities and semantics. They only vary in syntax and implementation approaches. [5] There is our aspect-oriented solution code with JBoss for playing sound concern in Figure 13. Since only syntax for pointcuts is a little different than AspectJ, we do not mention

again all our aspects. In XML files we define our pointcuts. In Figure 13.a, we get joinpoints where `hits(GameObject)` method of Aircraft is called. Program calls AircraftCrash method of JBossSound class. In Figure 13.b, program calls `hitEnemyEffect()` method of SoundEngine class and returns.

a)

```
<bind pointcut="call(void gameComponents.Aircraft->;hits(..) )">
    <after aspect="src.gameComponents.JBossSound" name="AircraftCrash"/>
</bind>
```

b)

```
public Object AircraftCrash( Invocation invocation  ) throws Throwable
{
    SoundEngine.hitEnemyEffect();
    return invocation.invokeNext();
}
```

Figure 13. Pointcut and Advice for playing sound after crash of aircraft. a, pointcut in XML file, b, advice code in Java.

We can change our program without recompilation by using JBoss since pointcuts are defined in XML files. In case of having a bug in a game, we can debug by changing our advices or at least deactivate bugged module by deleting proper advice from XML.

6. Related Work

There exist few studies on the effects of AOP on gaming applications. In [2], general problems of game developments are mentioned. It explains the crosscutting concerns encountered while building a game engine (See Figure 14. for a sample crosscutting problem in a gaming application). As we mentioned in problem statement part, trying to insert many tasks into a single class as a harmony is not possible. Sound management and Collision Detection are also classifies as crosscutting as an out study. They also point to new paradigms like AOP are trying to solve these problems but there is not an explicit example.

The author in [3] applies AOP to an OO game called Invaders. Aspect is applied to a single concern and its effect on the speed of the game is evaluated. No significant overhead on the frame per second of the game is reported.

Alves et al.[1] discuss AOP's effect on mobile games which are wanted to be system independent. They are trying to reduce power cost of mobile phones applying some different methodologies.

As a result of these related works, we see that AOP's effects on gaming applications are not fully explored yet.

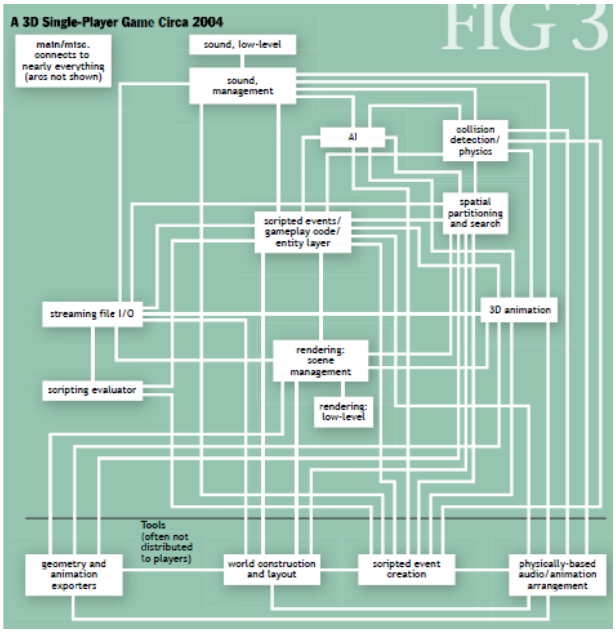


Figure 14. Dependency of Modules in a 3D Single-Player Game Circa 2004. [1]

7. Conclusion

In this paper, we first analyze the main concerns of gaming applications and claim that current abstraction mechanisms like classes are not sufficient to modularize these. We then conduct a case study on an OO game, Chicken Invaders, to identify the crosscutting concerns of the game and then create an AO design by modularizing these concerns into aspects.

We identify the crosscutting concerns of the game as detecting collision, removing out-of-frame objects, playing sound and throwing bomb. The concerns except throwing bomb are common in most of the games. Collision detection and the updates following detected collisions are very frequent in games with object interactions. Sound effects are also a part of almost all of the games. Finally, handling inactive objects is also a general concern of most of the games with moving objects. So, we can say that this case study is illustrative in terms of all games.

We show that all these concerns can be implemented by using aspects to reduce problems rising from crosscutting concerns. We use change scenarios for scattered concerns like SoundEngine to prove it decreases maintainability of the code. Tangling

[1] Adapted from [2]

ones are also shown to decrease coherency of GameEngine.

We implement aspects to handle these concerns by eliminating their actual implementations in OOP. So we modularize these concerns into aspects. This causes increase in coherency of GameEngine and fully modularize SoundEngine. SoundEngine is now easily maintainable because changes in one aspect would be enough to change, remove or add new sound operations. Additionally, for removing out-of-frame objects, AOP style is more realistic because it removes each object just after their position is updated instead of updating all objects and then removing the out-of-frame ones. This may cause some memory save especially in big games.

Development aspects' effects to game development are also examined. Especially, in big games development aspects are proved to be effective for tracing operations like object creation. Usage of these aspects eases the development process.

We demonstrate that some crosscutting concerns specific to most of the games may be eliminated by using AOP. By using AOP, both maintainability and modularity of the code increases. Since most of the concerns are common to all games, we think that AOP usage in gaming applications should be explored further. Some comprehensive study on problems of gaming application development may be conducted.

We believe that AOP should be involved in game programming more and hope that our work may be an inspiration for other studies.

7. Acknowledgments

We want to thank Asst. Prof. Bedir Tekinerdogan for his precious advices and contributions during this study and also his contributions to this workshop.

8. References

[1] Vander Alves, P.M.J., Paulo Borba, *An Incremental Aspect-Oriented Product Line Method for J2ME Game Development*, in *OOPSLA '04: Workshop on Managing Variability Consistently in Design and Code*. 2004: Vancouver, Canada

[2] Blow, J. 2004. Game Development: Harder Than You Think. *Queue* 1, 10 (Feb. 2004), 28-37.

[3] Ernest Criss, "Aspect Oriented Programming and Game Development", *Technical Report at Department of Information and Computer Sciences University of Hawai at Manao*, Honolulu, USA, pp. 1-5.

[4] Dijkstra, E.W.: On the role of scientific thought. In: Edsger W. Dijkstra: *Selected Writings on Computing: A Personal Perspective*. (Springer-Verlag 1982).

[5] Cristiano Breuel, Francisco Reverbel. *Join point selectors. In Proceedings of the 5th Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT 2007), pages 14-21. ACM Press, 2007.*

Aspect-Oriented Refactoring of Graph Visualization Tool: CHISIO

Esat Belviranlı, Celal Çığır, Alptuğ Dilek

Department of Computer Engineering, Bilkent University
06580, Bilkent, Ankara

{belviran, cigir, alptug}@cs.bilkent.edu.tr

Abstract

Graphs are data models and widely used in many areas from networking to data flow charts, from biology to computer science. Visualization, interactive editing ability and layout of graphs are critical issues when analyzing the underlying relational information. There are many commercial and non-commercial graph visualization tools. However, overall support for compound or hierarchically organized graph representations is very limited. CHISIO is an open-source editing and layout framework for compound graphs. CHISIO is developed as a free, easy-to-use and powerful academic graph visualization tool, supporting various automatic layout algorithms. However, since CHISIO was designed and implemented with respect to the object-oriented software approach, cross-cutting concerns were not considered during any phases of the development of the software. In this paper, we propose an aspect-oriented refactoring of CHISIO in which cross-cutting concerns are implemented as aspects.

Keywords
Information visualization, graph editing, software system, graph editor, compound graphs, Aspect-oriented programming, separation of concerns.

1. Introduction

Graphs are simply a set of vertices or nodes plus a set of edges that connect these nodes to each other. They are useful data structures to capture the relations (represented by edges) between the objects (represented by nodes). From network to data flow charts, from bioinformatics to computer science, many real world problems can be modeled and simulated by using this simple but powerful data structure.

Compound graphs are child graphs which can contain inner nodes. They are special type of nodes meaning that: a node can contain nested child nodes inside and also a node can have incident edges on it. By embedding a compound node into another compound node, multiple nested compound graphs can be constructed.

Graph visualization is a research area for producing visual representations of any data using graphs by deriving (or using) the topological information lying under the model. Understandability of these visual drawings must be high for easy interpretation and analysis. Geometrical information of the produced graph holds important measures for better graph visualization. Locations, sizes and relative positions of the nodes are part of the geometrical information and they should be adjusted either manually or automatically in order to produce an understandable and clear graph. This operation is called „graph layout". Many complex graphs can be laid out in seconds using automatic graph layouts.

Each application has its own type of data representation style. Therefore, different layout styles should be applied for different applications. For instance, hierarchical layout produces best result for family tree applications. This is due to the fact that ancestry information can easily be represented by the levels generated by the hierarchical layout.

Many different graph visualization tools are available either commercially or freely and they present a variety of features. Among one of these, CHISIO is a general purpose graph visualization and editing tool for proper creation, layout and modification of graphs. It was developed by Bilkent Information Visualization Team (IVIS) [17] using object oriented software development approaches. In addition to existing capabilities of existing graph visualization software, CHISIO includes compound graph visualization support among with different styles of layouts that can also work on compound graphs.

Although CHISIO is a mature software that is developed by strictly following commonly accepted OOSD approaches, we have faced with extensibility, reusability and maintainability problems while adding domain specific extensions to it. For instance, in trying to extend CHISIO to model biological pathways, we needed to extend not only the graph model in CHISIO; but also many other classes in other packages. Moreover, some of the new features (such as Layout Profiling) we plan to add to CHISIO consist of inherently crosscutting-concerns. We discovered that most of these problems can be solved with aspect oriented approach. As a result, we decided to

refactor CHISIO using aspect oriented programming techniques.

In the rest of this paper, we first explain the object oriented design of CHISIO. Then we address the cross cutting concerns and identify the aspects for these concerns. Later on, we give more details on how these aspects are implemented by including point-cuts and advices. We finally discuss the issues we had and what we have learned during our experience with AOP.

The remainder of this paper is organized as follows: Section 2 describes the OO design of CHISIO problem case. Section 3 identifies cross cutting concerns in the software and mentions corresponding aspects. Section 4 deals with Aspect-Oriented design and implementation issues we had during development. Related work can be found in Section 5. Further discussion is placed in section 6. We finally provide our conclusion in section 7.

2. Object-Oriented Design Overview of CHISIO

CHISIO is one of the major graph visualization and editing tools which supports compound graphs and various layouts. It is a software tool which is developed strictly following object oriented software development criteria. Figure 1, illustrates a running screenshot of CHISIO where compound graphs and supported layout styles can be observed.

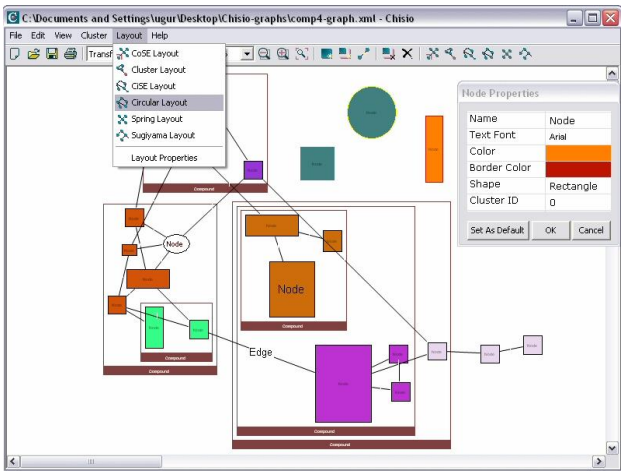


Figure 1. Overview of CHISIO

CHISIO is a comprehensive software which reuses industry-grade frameworks and libraries. It is mainly built on Model-View-Controller architecture which is a very well known structure especially used by visualization software. The design of CHISIO is depends on an extendible model that is developed using several OO design patterns. This section will give information about the architecture and the design of CHISIO in order to give a more clear idea about the software before discussing about the aspect oriented refactoring we have performed.

2.1. The Architecture of CHISIO

CHISIO, in general, is based on the Model-View-Controller architecture. This architecture is required by GEF [14] on which CHISIO is mainly dependent for graph editing and drawing operations. MVC architecture, which is illustrated in Figure 2, suggests separation of visualization and the data model and the interaction between these components is handled by a third module named controller. This ternary relation provides modification of the data model without any need for change in the user interface. The synchronization between model and view is carried out by controller.

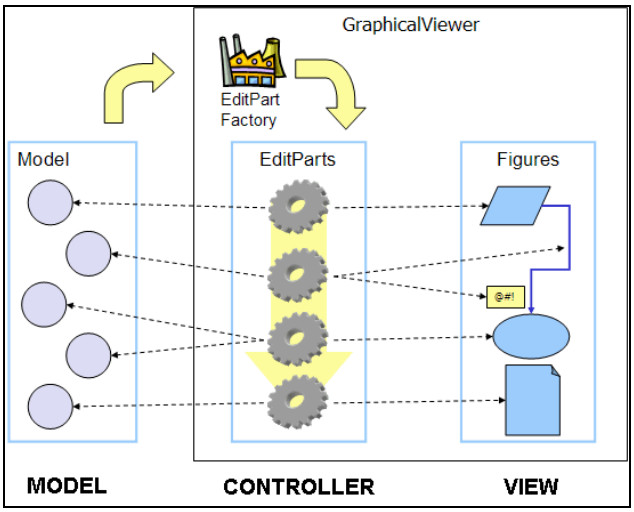


Figure 2. Model View Controller pattern applied to GEF.

In MVC, *model* is the concept to be represented; visualization of the model is the *view*. *Controller* is the bridge reflecting *model* changes to the *view*.

- **Model:** Model holds the data that is to be displayed. It also assesses the structure of the data and provides update mechanisms for modifications to data. Model does neither know how the data is visually represented nor responsible for updating the view. In CHISIO, model part is the graph structure. CHISIO model will be described in detail in Section 2.2.
- **View:** The view is the user interface which is used for visually presenting the model. Similar to the model, view does not also know about the underlying data structure and logic. Changes to the view are consequences of model updates. *Figures* in GEF are the correspondents of the view in CHISIO.
- **Controller:** Controller is responsible for creating and updating the view according to model changes. Whenever a change in the data occurs, controllers are

notified in order to redraw the view accordingly. *EditPart*s are the controllers in GEF.

Figure 2 illustrates how MVC pattern is applied in GEF.

2.2. Design Details of CHISIO

In CHISIO, graphs are composed of nodes, edges and compound nodes. Each of these three types extends from the type graph object.

Nodes are customizable building blocks of the graph where a user can change several properties such as background color, border color and font size. Users also are able to assign a cluster number to each node for further use in clustered layouts.

Edges have source and target nodes and also each node holds a list of incident edges on it. Edges also have customizable properties.

Compound nodes are special type of nodes which can hold a list of child nodes. Since they can act as simple nodes, they can be nested into each other resulting in a composition relation in the model hierarchy as the situation can be observed in Figure 3.

Common properties such as text and color are held in the graph object. Each graph object is responsible for informing the changes to its associated controller via a property change listener interface. When the controller receives the property change event, it notifies the view to update itself.

Besides figure updating, controller has other responsibilities such as determining capabilities of the associated model. For example, installing a XYLayoutPolicy to the EditPart (controller) corresponding to a compound node will enable insertion of child nodes into specific x and y coordinates.

Whenever a user wants to add a custom graph object where some restrictions should exist on editing, the appropriate policy should be created and installed to the corresponding controller.

CHISIO is able to load graphs in several widely known graph formats such as GraphML[10] and also save edited graphs in the same format.

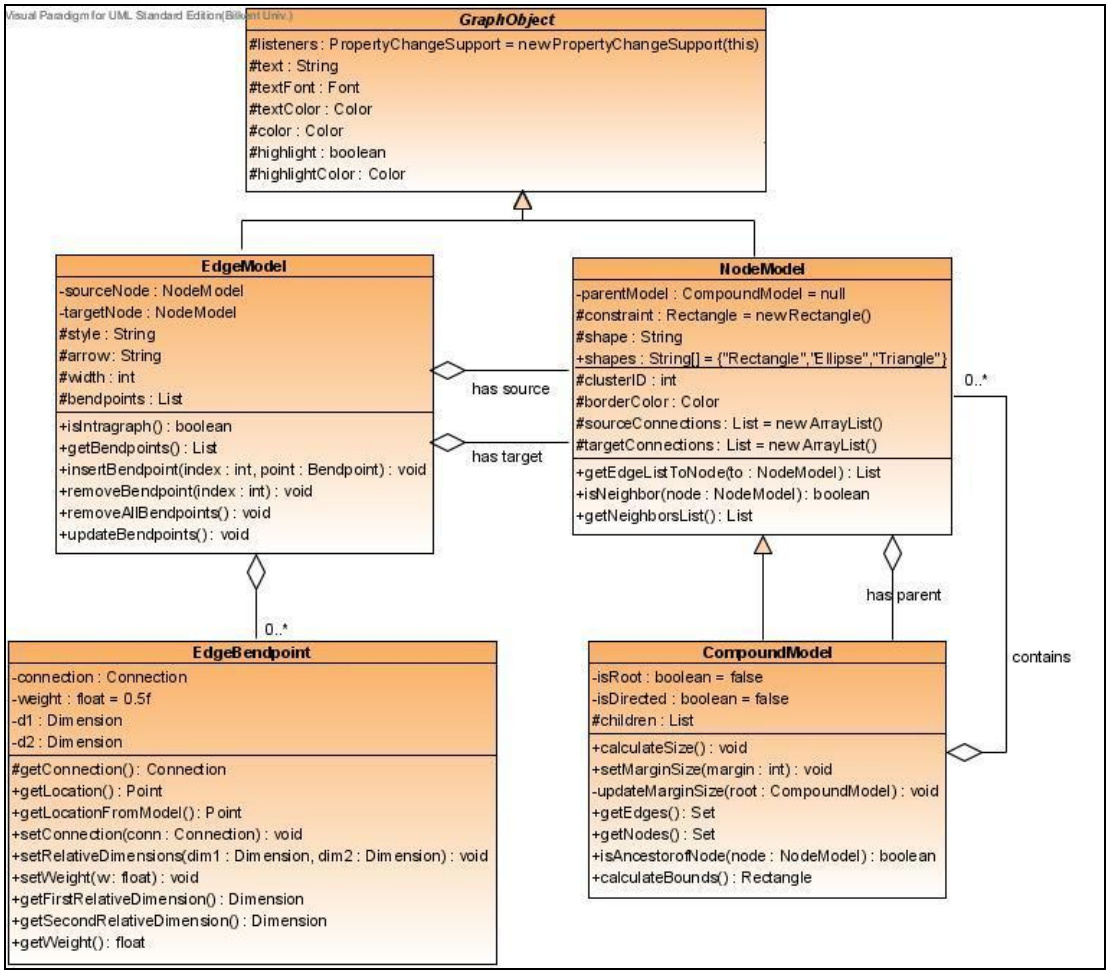


Figure 3. A UML Class Diagram illustrating the compound graph model used in CHISIO.

2.3. Applied Design Patterns

In order to simplify most operations and reduce coupling among software components, some design patterns were applied during the design of CHISIO. The new code introduced during this aspect oriented review is also designed and implemented in the same manner. These are as follows:

2.3.1. The Composite Pattern

"Composite pattern allows a group of objects to be treated in the same way as a single instance of an object. The intent of composite pattern is to compose objects in to tree structures to represent part-whole hierarchies. Composite pattern lets clients treat individual objects and compositions uniformly" [15].

CHISIO model is inherently based on a composite structure. Compound nodes, hold nodes as their children, which ends up the model to have nested compound nodes when they are used recursively. This structure can easily be observed in Figure 3.

2.3.2. Command Pattern

"The Command design pattern encapsulates the concept of the command into an object. The issuer holds a reference to the command object rather than to the recipient. The issuer sends the command to the command object by executing a specific method on it. The command object is then responsible for dispatching the command to a specific recipient to get the job done." [16].

Commands are one of the core structures of GEF. All operations on *model* are performed via commands. They are executed on a command stack which provides the ability for undo and redo. GEF provides hooks for creating and attaching required commands when some predefined cases occur. In CHISIO such commands are implemented according to custom needs of the application.

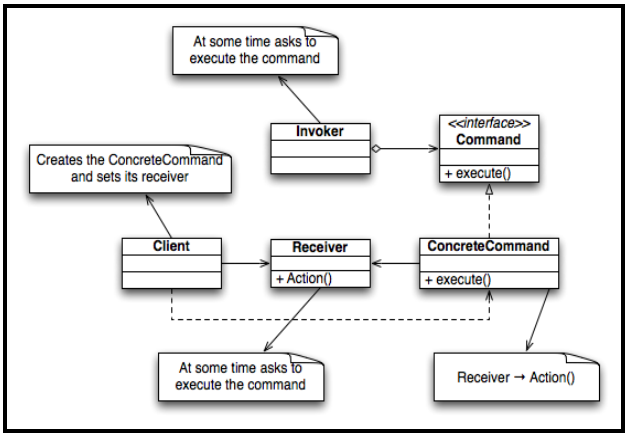


Figure 4. Command Design Pattern.

Once all undo and redo methods of the commands are implemented properly, undoing last operation is simply popping the latest command on the command stack and vice versa for redo.

2.3.3. Singleton Pattern

Singleton pattern is used where only one single instance of a class is needed or required during execution of a program. Constructor of the class should be public and the single instance of the class might be accessed via a static method which performs a lazy initialization.

Many dialogs in CHISIO can be grouped according to their similarity where the only difference is the texts they display. There is no need to create separate instances for such groups of dialogs. Changing related text is enough for expressing the intended message and getting the response.

For example, the new dialog introduced for setting command execution constraints is a singleton object and the same dialog instance is used during the execution of the program.

3. Aspect Identification

When we analyzed the OO design of the system, we have realized that some concerns were crosscutting through different modules. We were not able to localize them in a single component even by means of applying design patterns. This is because, these concerns were inherently crosscutting and scattered among several modules. Moreover, we realized that we need to introduce new cross-cutting concerns when we try to model specific domains such as a networking tool, in which not every node type can be incident to every other node type, if we stick to OO design. Yet, as mentioned in the introduction part, layout profiling feature that is thought to be integrated into the system is inherently crosscutting.

Crosscutting concerns which we have recognized in our system can be addressed with the following aspects:

As production aspects, we have identified the following:

- Command Execution Constraints
- Model Property Settings

Whereas development aspects we have identified are as listed below:

- Layout Profiling
- Action Tracking

3.1. Production Aspects

The production aspects we determined are thought to solve some of the existing understandability and extendibility problems CHISIO contains.

3.1.1. Command Execution Constraints

In OO implementation of CHISIO, there are generic node, edge and compound types. However, when a developer decides to extend CHISIO, specific graph object types must be introduced. It is generally the case that, a specific edge type can only attach to some specific node types or a specific node can only be child of some specific compound types. For instance, in order to model a biological pathway as a graph, new node types (such as Protein, DNA, RNA, etc.), new edge types (such as Activation, Protein-Protein-Interaction, Inhibition, etc.) and new compound types (such as Abstraction, Compartment, etc.) must be introduced via extending the current graph model. However, you also need to extend from other classes of CHISIO other than the model package.

The connection of a node with an edge or an insertion of a node to a compound in CHISIO is achieved via executing instances of the class called "*Command*". Since CHISIO is a general graph visualization tool, those commands do not consider any domain specific constraints. In order to introduce those constraints many commands in the software must be extended and their execution methods must be overridden. However, this situation results in scattering of those constraints into many classes. In order to tackle this problem, we introduce a new aspect called CommandConstraints. So, when new graph object types with different constraints (node-edge attachment constraint, or compound-node containment constraint) are to be introduced, the Command classes are no longer needed to be extended. The only change will occur in the specified aspect.

Figure 5 illustrates the class hierarchy that should have been adapted if an only OO approach was chosen. Thanks to the newly introduced aspect, there is no need to override execute methods of different methods. Thus there will be no change in the existing class hierarchy of the commands.

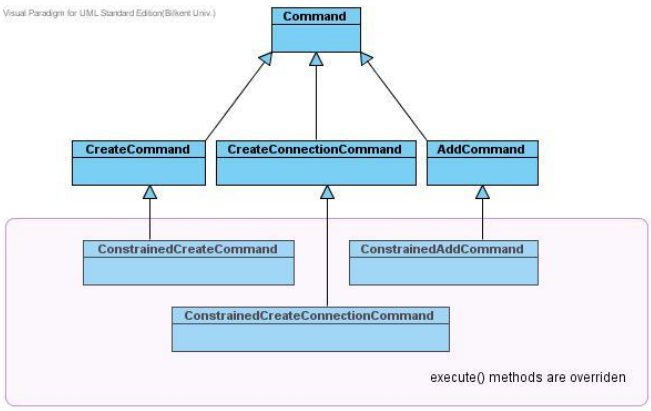


Figure 5. Prevented OO Command Hierarchy.

3.1.2. Model Property Settings

As mentioned above, CHISIO is built upon GEF, which is a framework based on MVC pattern. Any changes in any properties of the model object must be propagated to the controller. However, this results in scattering of the controller notification concern in the entire graph model classes. When a new property is to be introduced to a graph object class, the notification code must also be written for the new field. In order to tackle this problem, we decided to introduce a new aspect called *ModelPropertySettings*. In fact, this is not just a solution for CHISIO, but also a general aspect oriented solution for the separation of notification concern from the actual modeling concern. The graph model classes in CHISIO now do not include code related with the notification of controller part. If a new field, for which the controllers must be notified when a change in the field occurs, a new point-cut must be included to the aspect if it is not already captured.

3.2. Development Aspects

Development aspects are useful to provide more information the programmer about the internals of the software to during the development phase. These aspects are not included in the release. Development aspects address many possible cross cutting concerns that might be faced when the same functionality would be implemented with OO approach. By the help of AO programming, developers can easily debug and reveal internals of the software.

We have implemented two development aspects that will help us with the further improvement and extension of CHISIO.

3.2.1. Layout Profiling

CHISIO provides an attachable layout hook structure which enables developers to easily implement and adapt new layout algorithms. Layout algorithm development is a long and difficult process which requires comprehensive profiling in order to increase performance of the layout. The developer needs to fulfill many requirements such as decreasing method call count and detecting and solving method hotspots. Thus layout profiling should be included in the developer version of this tool. Unfortunately, current implementation of CHISIO lacks this capability. Implementing the profiling will give us the chance to detect and correct bottlenecks of the layout algorithms.

3.2.2. Action Tracking

Nearly, all graph model changes in CHISIO occur as results of user interactions. Those interactions are called „actions" in GEF terminology. After an action is triggered, the control flows through internals of the framework. As a result, it is really difficult for a newbie developer to understand and follow the mechanism of GEF. Therefore,

we decided to implement tracking functionality for the actions to ease the debugging and testing of the system.

4. Aspect-Oriented Programming

For implementing the identified aspects, we used Eclipse plug-in of AspectJ for most of the aspects. However, we implemented one of the aspects by using JBoss AOP library in order to benefit from the dynamic AOP feature (addition and removal of aspects in the runtime dynamically).

4.1. Command Execution Constraints Aspect

In order to add domain specific constraints to CHISIO Commands, we need to capture the execution of the commands for which constraint checks must be performed. Thanks to the OO design of the command package in CHISIO, we can capture the execution of the methods by writing a somewhat general point-cut. We use around advice for that point-cut, in order not to execute the command in case a possible constraint check fails. In order to test the usefulness of this aspect, we introduced new edge, node and compound node types, which extend from the base graph model of CHISIO. As explained in Command Execution Constraints section, there is no need to introduce new command types. We implemented a singleton dialog as depicted in Figure 6 to add/remove constraints (rules) and to enable/disable the constraint checking in the runtime via using hot deployment of JBoss.

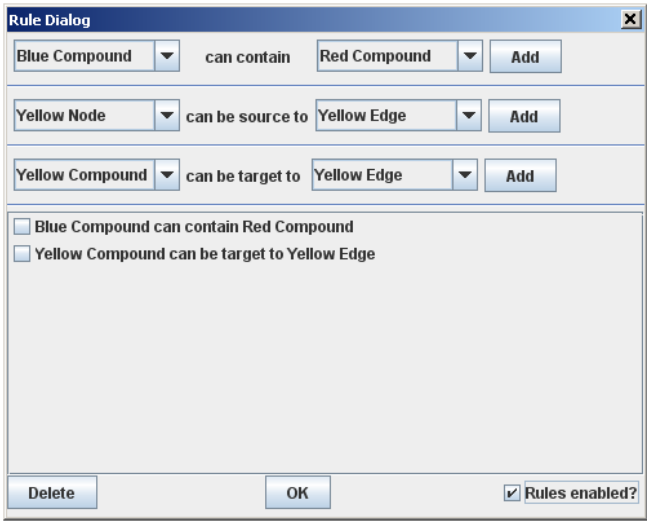


Figure 6. Rule Creation Dialog

Here is the point-cut for this aspect written in AspectJ and JBoss respectively:

```
pointcut commandConst(Command command):
   target(command) &&
   (execution(void ChsCompoundCommand.execute()) ||
   execution(void CreateCommand.execute()) ||
   execution(void CreateConnectionCommand.execute()));
```

Figure 7. Command constraint pointcut for AspectJ

```
execution(public void *.ChsCompoundCommand->execute())
OR execution(public void *.CreateCommand->execute()) OR
execution(public void *.CreateConnectionCommand-
>execute())
```

Figure 8. Command constraint pointcut for JBoss

The interceptor class called CommandInterceptor, performs the constraint checks with respect to the rules submitted via rules dialog. If the constraint check fails, then the command is simply ignored and not executed. Here is the pseudo-code for invoke method of CommandInterceptor class:

public Object invoke(Invocation invocation)

- Cast invocation to method invocation

- Assign target object of the method invocation to variable command

- If command is related with connection creation, check validity of node-edge relation

- Else check validity of node-compound relation

- If the validity check fails, show error message

- Else, run the command as normal

The code related to hot deployment feature of JBoss (dynamic advice binding) is given below:

```
this.binding = new AdviceBinding(
        "execution(public void *.ChsCompoundCommand->execute())" +
        " OR execution(public void *.CreateCommand->execute())" +
        " OR execution(public void *." +
            "CreateConnectionCommand->execute())",
        null);
this.binding.addInterceptor(CommandInterceptor.class);
```

Figure 9. Advice Definition and binding via JBoss

In Figure 9, the advice inside CommandInterceptor is bound to the point-cut expression written via *addInterceptor()* method. To enable/disable the binding dynamically in run time, addBinding()/removeBinding() methods must be called as seen in Figure 10.

```
if (this.constraintEnabler.isSelected())
{
    AspectManager.instance().addBinding(this.binding);
}
else
{
    AspectManager.instance().removeBinding(this.binding.getName());
}
```

Figure 10. Adding/ Removing of advices dynamically

4.2. Model Property Settings Aspect

Writing the model property setting as an aspect was not as trivial as the other aspects. The reason is that, there needs to be a mechanism to determine when to notify the controllers when a property is set in the model part of the software. Simply, capturing the set methods of all the fields does not solve the problem, because there might be set methods for which the controllers must not be notified. Moreover, there are some methods where we need to notify the controllers even if they are not set methods for any property. For instance, the method to add a child node to the children list of a compound is not a set method.

However, the most general case a controller should be notified is within a set method for a field. Thus we determined a general point-cut to capture the set methods. The point-cut is given in Figure 11.

```
pointcut propertySet(GraphObject graphObject, Object object):
    target(graphObject) &&
    set(* GraphObject+.*) &&
    args(object)&&
    withincode(void GraphObject+.set*(*));
```

Figure 11. The propertySet point-cut to capture the setting of the fields of any Graph Object type inside the set methods.

The notification mechanism in CHISIO requires an identifier string for the notification of different properties. As a result, we need to use different notification strings for each different property. However finding that notification string for different properties is not that trivial. We developed a mechanism relying on Java"s reflection API. If when a property named "sample" is set within the "setSample" method, there must be a final static String named "P_SAMPLE" in the class containing the "sample" property. If the programmers follow this pattern in adding new fields, the controllers will automatically be notified when the field is set in the set method. In order to find the "P_SAMPLE" notification string, when "sample" field is set; we decided to create a hash-map between each field and the corresponding notification string. To create this hash-map, we determined yet another point-cut named notificationChecker(), given in Figure 12.

```
pointcut notificationChecker():
    staticinitialization(GraphObject+);
```

Figure 12. The point-cut to be called, when a GraphObject type class is initialized.

Both of the point-cuts given in Figures 11 & 12 are referred by after advices. The advice given in Figure 13 iterates through fields of the GraphObject type class and creates a mapping between the fields and notification strings as explained above via using Java"s reflection API. The advice given in Figure 14 makes use of the hash-map constructed by the advice in Figure 13. It finds the corresponding notification string for a property that is just set and performs the actual notification of the controllers.

```
after():notificationChecker()
```

Figure 13. The after advice using notificationChecker point-cut

```
after(GraphObject graphObject, Object object):propertySet(graphObject, object)
```

Figure 14. The after advice using propertySet point-cut

Other than the setting of the fields within set methods, there are some cases where the controllers must be notified as explained. There are specific point-cuts written for that purpose. Figure 15 lists those point-cuts.

```
pointcut childNodeAdditionRemoval(CompoundModel compoundModel, Object child):
    target(compoundModel) &&
    args(child) &&
    execution(void CompoundModel.*Child(Object));

pointcut edgeAdditionRemoval(NodeModel nodeModel):
    target(nodeModel) &&
    (execution(void NodeModel.add*Connection(EdgeModel)) ||
        execution(void NodeModel.remove*Connection(EdgeModel)));

pointcut bendPointAdditionRemoval(EdgeModel edgeModel):
    target(edgeModel) &&
    execution(void EdgeModel.*Bendpoint*(..));
```

Figure 15. The reamining point-cuts in ModelPropertySettings aspect.

Those point-cuts capture the methods where addition/removal of a node to/from a parent node, addition/ removal of an edge to/from a node and addition/removal of a bend point to/from an edge occur. There are separate after advices written for those point-cuts. The tasks performed inside the advices using those point-cuts are pretty much similar. All the advices notify the controller part with specific notification strings. An overview of the advices can be seen in Figure 16.

```
after(CompoundModel compoundModel, Object child):childNodeAdditionRemoval(
        compoundModel, child)

after(NodeModel nodeModel):edgeAdditionRemoval(nodeModel)

after(EdgeModel edgeModel):bendPointAdditionRemoval(edgeModel)
```

Figure 16. The reamining advices in ModelPropertySettings aspect.

4.3. Layout Profiling Aspect

As stated previously, all layout algorithms in CHISIO extend from an abstract class named "AbstractLayout" and there is the method "layout()" which is the start point for all layout styles. Thus in profiling, we needed to capture the control flow of this method in order to calculate the total time spent during execution of each method executed under this control flow and the number of times this method is executed

For that purpose, we have used two pointcuts, allLayoutMethodCalls() and runLayoutMethod() which are depicted in Figure 17.

```
 * Pointcut for catching runLayout method executions.
pointcut runLayoutMethod():
    execution(* org.gvt.layout.AbstractLayout.runLayout(..));

 * Pointcut for catching all method executions in the control flow of layout()
pointcut allLayoutMethodCalls():
    execution(* org.gvt.layout.*.*(..)) &&
    !runLayoutMethod() &&
    cflow(execution(* org.gvt.layout.AbstractLayout+.layout(..)));
```

Figure 17. Point-cuts for layout profiling

The first point-cut, runLayoutMethod() is used for detecting the entry and exit points of the layout operation. We have two advices for this point-cut which are illustrated in Figure 18.

```
before(): runLayoutMethod()

after(): runLayoutMethod()
```

Figure 18. Advices for runLayoutMethod point-cut

In the before advice for runLayoutMethod() point-cut, we launch the profiling dialog which displays a tree for the call hierarchy of the methods that will be executed in the control flow of layout() method. This tree is updated in constant time intervals during profiling in order to display the real-time call hierarchy, call counts and execution times of the methods. After advice for runLayoutMethod() simply stops automatic refreshing of the tree.

The second point-cut, allLayoutMethodCalls(),uses cflow keyword in order to capture all method executions under execution flow of AbstractLayout.layout() method. We exclude all of the methods outside the layout package because they are out of scope for profiling.

In order to measure elapsed time properly, we have used both before and after advices for this allLayoutMethodCalls() point-cut. After a call of a method, we finalize the timer for the method and save its duration. Headers for these two advices are given in Figure 19.

```
before(): allLayoutMethodCalls()

after(): allLayoutMethodCalls()
```

Figure 19. Before and after advices headers for layout profiling.

In the before advice for allLayoutMethodCalls() point-cut, we first create a tree node for the method to be executed and set the current time as the method"s start time. In order to prevent redundancy, the nodes corresponding to same methods which have exactly the same hierarchies are represented by the same node, hence making the profiling tree more meaningful. Additionally we put the created (or reused) tree node and executed method data into a stack which will be further used by the after() advice.

In the after advice for allLayoutMethodCalls() point-cut, we retrieve the node and the method data corresponding to already executed method from the stack. Since the method execution is finished, we save the elapsed time and reset related fields.

Figure 20 represents a screenshot taken during profiling.

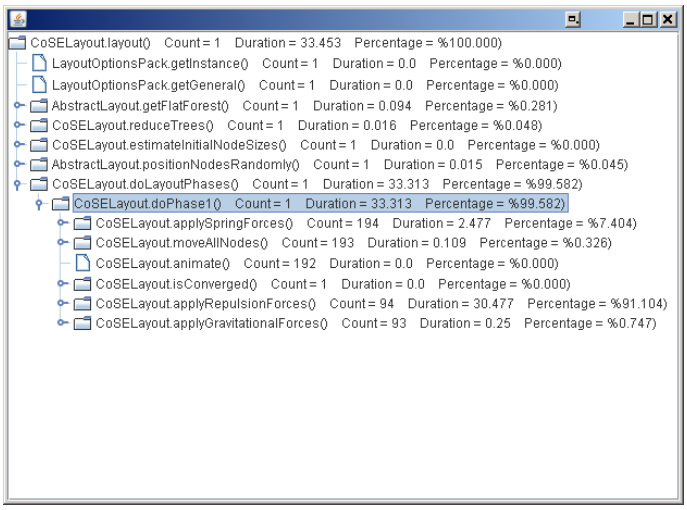


Figure 20. Layout Profiling Screenshot

4.4. Action Tracking Aspect

Action tracking aspect is more like a specialized logger which keeps track of the executed methods of two parent classes, Command and EditPolicy, and also their child classes. Logs are generated and printed on a separate console in debug mode. Whenever user performs an interactive operation, related method calls in the associated Command and EditPolicy derivative classes are displayed.

The point-cuts used for this aspect is given in Figure 21. Besides the point-cuts used for catching all the methods in child classes of EditPolicy and Command, we have also included a method for catching constructor calls of Command classes in order to provide some more useful logging information that might be derived during Command creation.

```
pointcut editPolicyMethods() :
   execution (* org.eclipse.gef.EditPolicy+.*(..) );
pointcut commandMethods() :
   execution (* org.eclipse.gef.commands.Command+.*(..) );
pointcut commandConstructionMethods() :
    execution (org.eclipse.gef.commands.Command+.new(..));
```

Figure 21. Action tracking aspect point-cuts.

Advices for these pointcuts, which are depicted in Figure 22, are simple before advices, since they only invoke the method of the logger dialog class which is used for displaying given method signatures in a simple JTextArea.

```
/**
 * Advice for logging edit policy related methods.
 */
before() : editPolicyMethods()
{
    this.log(thisJoinPoint.getSignature());
}

/**
 * Advice for logging command related methods.
 */
before() : commandMethods()
{
    this.log(thisJoinPoint.getSignature());
}

/**
 * Advice for logging command construction methods.
 */
before() : commandConstructionMethods()
{
    this.log(thisJoinPoint.getSignature());
}
```

Figure 22. Action tracking aspect point-cuts.

Figure 23 illustrates a sample screenshot of the log produce after inserting a new node into the graph:

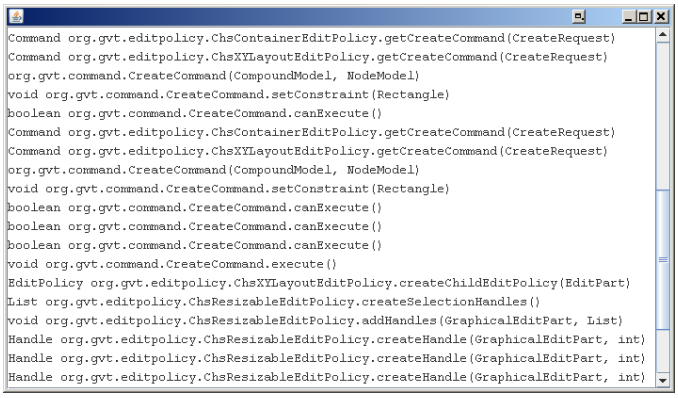


Figure 23. Action Tracking Screenshot

5. The Related Work

There are many different commercial and non-commercial graph visualization tools. Some of them, generally the ones with better capabilities and user interfaces are commercial. There are also good ones even though they are non-commercial. Tom Sawyer's TSV [1], yWorks' yFiles [2] and ILOG's JViews Diagrammer [3] are well-known examples for commercial tools. Prefuse[10], uDraw [4], Graphviz [5], VGJ [6] and Graphlet [8] are some of many non-commercial tools.

In order for a graph visualization tool to be considered as useful, it has to support some basic features. Some of them are ability to create, delete, move, resize graph objects, zooming, highlighting, scaling functionalities, save/load capability, clustering of nodes and user interface to view properties of graph objects. Although CHISIO is an academic tool, its features and capabilities are comparable to most non-commercial and some commercial tools. Figure 24, shows a comparison of CHISIO with other visualization tools including both the commercial and non-commercial ones.

	Move/Resize/ Create/Delete	Zoom/ Scale	Compound Support	Highlighting	Save/ Load	Clustering/ Grouping	Object Inspector
TSV	+	+	+	+	+	+	+
JViews	+	+	+	+	+	+	+
yEd	+	+	+	-	+	+	+
aiSee	+	+	+	-	+	-	+
Graphviz	+	+	-	-	+	-	+
Prefuse	+	+	+	-	+	-	-
Gravisto	+	+	-	-	+	-	+
JGraphEd	+	+	-	-	+	-	+
VGJ	+	+	-	-	+	+	+
GDToolkit	+	+	-	-	+	-	+
uDraw	+	+	-	-	+	-	+
VCG	-	+	-	-	+	-	-
CHISIO	+	+	+	+	+	+	+

Figure 24. Comparison Table

However, none of these tools used AOP techniques as far as we know. CHISIO may differ from these tools since AOP technique is applied in the development cycle.

Besides, UML class drawing tools can be considered as a similar study since UML can be considered as graph visualization tool [9]. In this study, AOP technique is applied and some improvements made as an outcome of AOP.

However, the concerns in developing a UML drawing tool and a general graph drawing and layout tool seem to differ in many ways. Thus there is not an aspect oriented design of a general graph drawing tool to which we can make comparisons.

6. Discussions & Lessons Learned

During the aspect oriented refactoring of CHISIO, we gained experience about the benefits of aspect oriented software engineering. However, we also tackled with problems resulting from the fact that, CHISIO was not originally designed with the cross-cutting concerns in mind. For instance, aspectizing the cross-cutting concern of notification of the controllers by the model classes was really painful and it is not as clean and solid as the other aspects. If we were to design CHISIO from scratch with an aspect oriented approach, we might have ended up with a simpler and easy to implement aspect. Moreover, we realized the fact that, there are inherently cross-cutting concerns in a general graph visualization tool, such as checking the node-edge attachment and compound-node containment constraints or profiling the layout.

In improving CHISIO via aspect oriented software approach, we had the opportunity to use and compare two different libraries in terms of usability and functionality. We implemented most of the aspects mentioned in section

3 with AspectJ, whereas we preferred using JBoss AOP for the command execution constraints aspect.

We integrated AspectJ plug-in of Eclipse to the development environment, which provided ease of use when compared to JBoss AOP. The cross-cut references and the advice markers, which are useful in checking the code segments captured by a certain advice, provided by the plug-in increase the testing and debugging opportunity for aspect oriented programming. Since AspectJ treats the point-cuts and advices as first level abstractions, you get any possible syntax error at the time of compilation. This functionality is sure to shorten the implementation phase of the software development cycle especially for the big projects. On the other hand, for JBoss AOP you have to execute the code in order to get any possible syntax error related with your point-cuts, since you write them either in a separate XML file or as string parameters within Java code.

The biggest advantage of using JBoss AOP is the ability to add/remove advices dynamically in the runtime. For integrating command execution constraints to CHISIO, JBoss AOP seemed as a better candidate thanks to having this feature. If the constraints are to be removed dynamically at any time, one can just disable the advice via the provided user interface. After that, the interceptor code is no longer called and this might become an important performance gain over AspectJ. This is due to the fact that, there is no way to prevent AspectJ to run/avoid certain aspect with respect to some property. The advice code is run every time regardless of anything as long as the point-cut is captured.

7. Conclusion and Future Work

We have reviewed CHISIO, a compound graph visualization and layout tool and framework, with respect to the aspect oriented approach. Since cross-cutting concerns were not explicitly considered during the development phase, we have separated the cross-cutting concerns in CHISIO as much as we can and we integrated the layout profiling feature as an aspect. By the end of this study, CHISIO has become easier to extend, maintain and reuse.

We used AspectJ as the aspect oriented programming language. Modeling and implementation of cross-cutting concerns as aspects in AspectJ is easy. This way, we have seen that AOSD reduces complexity of the software system by modularizing the cross-cutting concerns. The resulting system is less coupled and highly cohesive. This positive improvement makes the software system easily understandable, adaptable and manageable. Therefore, the overall quality of the software has been increased.

Due to the time constraints, we ended up with just two development and two production aspects. Thus, possible future work should be identification of further cross-cutting concerns existing in the software and handling those concerns in the same manner. One such concern is the enabling/disabling of "Highlighting" mechanism. Highlighting is not supported internally by GEF framework and it was implemented as a separate layer on top of the graph layer. Since it is not handled internally by the framework, it must be disabled before or enabled after some operations such as running the layout or performing animation. Thus the code related with highlight layer handling is scattered in different classes of different packages. Further cross-cutting concerns should be found with further study of the existing system as stated.

8. References

[1] Graph Layout Toolkit and Graph Editor Toolkit User's Guide and Reference Manual. Tom Sawyer Software, Oakland, CA, USA, 1992-2002.
http://www.tomsawyer.com.

[2] yFiles User's Guide. yWorks GmbH, D-72076 Tbingen, Germany, 2002. http://www.yworks.com.

[3] JViews User's Guide. ILOG SA, 94253 Gentilly Cedex, France, 2002.
http://www.ilog.com.

[4] uDraw(Graph), 2005.
http://www.informatik.uni-bremen.de/uDrawGraph.

[5] Graphviz - Graph Visualization Software.
http://www.graphviz.org.

[6] VGJ, Visualizing Graphs with Java. Auburn University, Auburn, Alabama.
http://www.infosun.fmi.uni-passau.de/Graphlet

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides.
Design Patterns Elements of Reusable Object Oriented
Software. Addison-Wesley, 1995..

[8] Graphlet, A toolkit for graph editors and graph algorithms. University of
Passau, Passau, Germany.
http://www.infosun.fmi.uni-passau.de/Graphlet

[9] Y. Dedeoğlu, H. Sözer, M. Tekkalmaz, A. Uluçınar. *Aspect-Oriented UML Class Diagram Drawing Tool*,1st Turkish AOSD Workshop Preceedings, TAOSD 2003.

[10] The GraphML File Format
http://graphml.graphdrawing.org/

[11] The Prefuse Visualization Toolkit.
http://prefuse.org/

[12] The AspectJ(TM) Programming Guide.
http://www.eclipse.org/aspectj/doc/released/progguide/

[13] JBoss AOP
http://www.jboss.org/jbossaop/

[14] Eclipse Graphical Editing Framework (GEF)
http://www.eclipse.org/gef/

[15] Composite Pattern
http://en.wikipedia.org/wiki/Composite_pattern

[16] Command Design Pattern
http://www.jboss.org/jbossaop/

[17] Bilkent I-Vis Information Visualization Research Group
http://www.cs.bilkent.edu.tr/~ivis/

Aspect-Oriented Multi-Client Chat Application

Duygu Ceylan, Gizem Gürcüoğlu, Sare G. Sevil
Department of Computer Engineering, Bilkent University
Ankara, Turkey 06800
{dceylan, gizem, sareg} @cs.bilkent.edu.tr

Abstract

As the demand for network applications increases and the necessity of developing more sophisticated systems arises, distributed computing such as Client-Server applications receives more attention. Due to the distributed nature of these applications, many concerns like synchronization, security, and performance issues become crucial and the ability to modularize these concerns becomes more valuable. This paper focuses on using Aspect-Oriented Software Development approaches for separating such concerns. An example case, namely a chat application, constitutes the basis of this study and several concerns specific to this application are illustrated. Considered and handled concerns are types of concerns that occur both in development and production phases of the application. Implementations using different environments are also discussed.

Keywords
aspect-oriented software architecture design, multi-client chat application, design patterns.

1. Introduction

From publishing and sharing of information through software, to the execution of complex processes within several computers, distributed systems are widely used among all kinds of applications in software development. These systems are mainly composed of multiple processes that generally work in parallel in some predefined structural design. Among these structural designs, client-server architecture models are some of the most commonly applied software architecture models where their application range varies from simple business applications to standardized internet protocols including HTTP, DNS and SMTP [3].

In client-server applications, processes are classified into client and server systems. These systems communicate over a network connection through which client systems make connections and request for operations while server systems wait for requests from clients and serve them accordingly.

Throughout the years, providers such as Microsoft, Yahoo and Google, and many open-source software developers have adapted client-server model to a popular user application: the multi-client chat. These chat applications generally appear in the form of *chatrooms* where synchronous, and sometimes asynchronous, conferencing takes place; or they can be in the form of instant messengers where users communicate by exchanging textual, visual or audio data over a connection. While in chatrooms all users are allowed to communicate with each other, in IMs only users who accept to communicate with each other are allowed to exchange messages.

In this paper we go through the design and implementation process of an instant messaging chat application program. We analyze the components of a standard chat application and the concerns that might be encountered during development and production stages of the application. In order to facilitate the handling of concerns we propose the application of aspect oriented approaches and discuss why these aspects are more convenient for usage.

The paper is organized as follows: in section II the initial object oriented design of the application is explained. Section III describes the problems that might be encountered with this design. In section IV a new design with aspects, applied in AspectJ, is presented. Section V discusses the implications of aspect usage. Section VI describes possible implementation differences with JBoss usage in implementing aspects. In section VII related work is discussed and section VIII concludes the paper.

2. Multi-Client Chat Application: Object-Oriented Design

2.1 Description of the Application

The multi-client chat application focused in this paper is a server-client chat application where multiple clients

communicate with each other by sending text messages over a connection provided by a single server. For this reason, it would be convenient to analyze this system in two main components: server and client.

Each client component found in the system represents a user who wants to communicate. A client is identified by his/her username. All users have their own friend lists which are simply lists of other clients (i.e. contacts) using the chat application. Each user is allowed to communicate only with his/her own contacts.

Each contact can be in one of two possible states, “Online” or “Offline”, depending whether they are currently connected to the server or not. When an offline contact connects to the server, his/her status changes from offline to online. Similarly, when an online contact disconnects from the server, his/her status becomes offline.

Users are allowed to have private conversations (private chats) with their online friends. During a private conversation, the contact being talked to might disconnect from the server without notifying the client. If the client continues to send messages when this happens, an automatic notification is send to the client informing him/her of the situation. Finally, a client can add a new contact as his/her friend and remove an existing friend.

The main function of the server component is to wait for connection requests from the clients and authenticate these requests by checking the usernames and passwords provided in requests. The server thus stores a list of valid username-password pairs which are used in the authentication process. In our current implementation, when a username which is not found in the list is provided by a connection request, the server accepts this request as a new client and adds the username-password pair to the existing list. In addition to authentication information, the server also keeps data related to the friend lists of the clients. For each client, a list of friends is kept as well as a list of other clients that has this client as friend. Upon confirming a connection request from a client, the server sends usernames of the friends of the client and their connection status. Finally, the server processes the requests from a client such as adding or removing a friend by updating lists accordingly and it provides necessary communication base for private conversations with friends. Use case diagrams shown in Figures 1 and 2 summarize these components and their features.

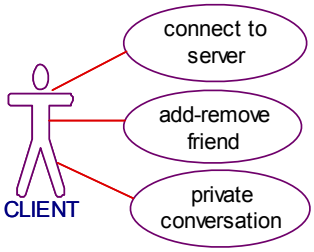


Figure 1. Use case diagram: Client

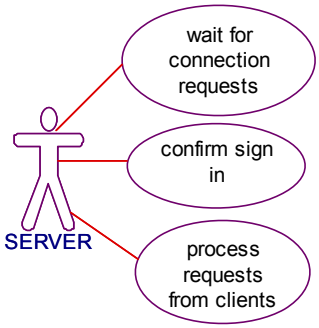


Figure 2. Use case diagram: Server

2.2 Object-Oriented Design

When a closer look is taken at the design of the multi-client chat application system, we can see that it is possible to represent the system with two main classes, corresponding to the two main components of the system, and some other helper classes.

In our design we have chosen to implement the client component as the *Client* class that has associations with user interface classes *MainWindow* and *PrivateWindow*. The *MainWindow* can be defined as the class responsible for the main user interface window of a client. This class is used in displaying the list of contacts, providing necessary interface for adding and removing contacts and for signing in and out of the chat application. Figure 3 shows an example of a main window for an online client.

The *PrivateWindow* is the class responsible for the user interface for a private conversation between two clients. An example of a private window is shown in Figure 4.

Figure 3. Main Window for signed in Client

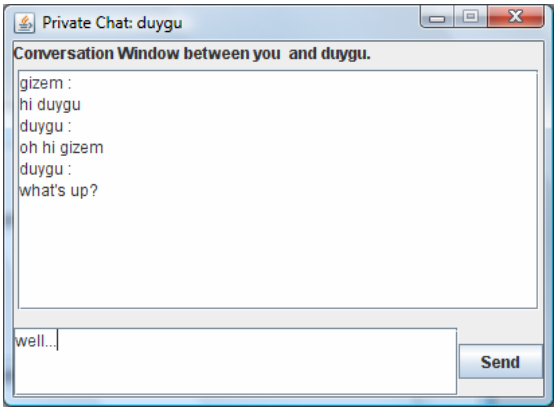


Figure 4. Private Chat Window of Client Gizem with Client Duygu

In addition to the user interface classes, *Client* also has an association with the class *ContactList* which contains a list of the friends of a given *Client*. Contacts in the *ContactList* are represented as *Contact* objects that consists of a username and connection status. *Client* objects also include a *MessageFormatter* object which is responsible for formatting the messages sent by the *Client* in the correct way. These relationships are summarized in the class diagram shown in Figure 5.

For the server component, the main class is the *Server* class. As a socket connection is received, the *Server* creates a new thread, namely a *ServerThread* object, to send and receive messages from the corresponding socket. *ServerThread* class is also responsible for parsing the messages coming from clients. In order to be able to hold information related to clients such as username, password and friend list, *Server* class creates *ServerClient* objects which are a simpler version of *Client* objects containing only relevant information and excluding everything else. Finally, the status of *ServerClient* objects is implemented through a *ConnectionStatus* interface. Currently, there are two concrete implementations of this interface which are *OnlineConnectionState* and *OfflineConnectionState* respectively. Operations for sending messages to the clients are defined in the *ConnectionStatus* interface and implemented differently according to the status of the client. The relations of the server component are summarized in the class diagram shown in Figure 6.

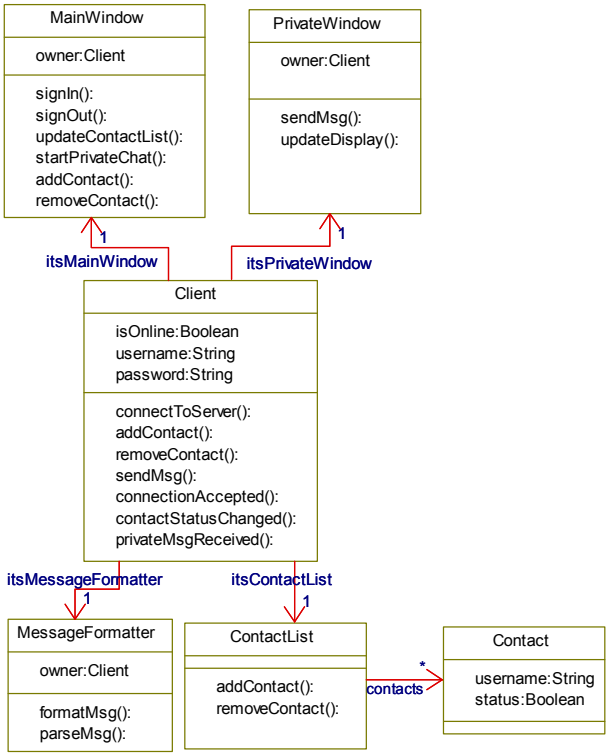


Figure 5. Class Diagram of the Client component.

2.3 Implementation of Design Patterns

In order to enhance the object-oriented design described in the previous subsection and ease some operations of the application, two design patterns have been implemented. The first design pattern that has been considered is the Observer pattern. In the system it is important to notify clients when another client that appears within their contact lists changes his/her status (online/offline). For this purpose, subscription-notification structure is used. In this structure, each client has a list of other clients that contain this client as friend. We call this list as the observer list. When client A adds client B as a friend, A is automatically attached to the observer list of B. Similarly, when client A removes client B from its friend list, it is automatically detached from the observer list.

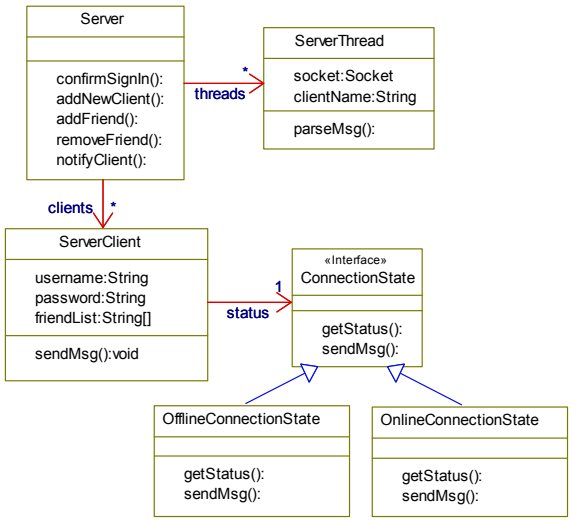


Figure 6. Class Diagram of the Server component.

Thus, when B changes status, a notification message is sent to the clients contained in the observer list of client B such as client A. Upon receiving a notification, other clients update their contact list views. This way, the need for checking whether each client contains B as a friend or not is eliminated. The structure of this pattern can be shown as in Figure 7.

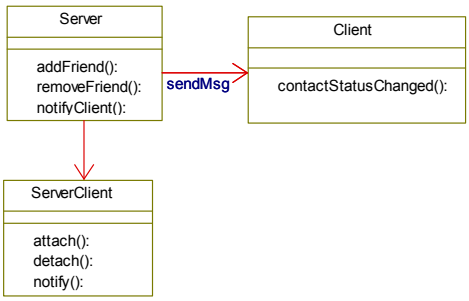


Figure 7. Class Diagram: Observer Pattern implementation.

The other design pattern included in the architecture is the State pattern. In our system clients may be in one of many states and the server needs to check the status of a client when a need of sending a message to that client arises. Currently our system supports two different states for its clients: online or offline. But in the future, other Client states may be added to the system such as Busy and/or Away where in each state, the client may perform different actions for sending messages. For example for the Away state, the client can send automatic replies to incoming private chat messages. But defining new states may result in the need of heavy code maintenance. In order to ease these operations, State pattern is used to implement the connection state of the client. A *ConnectionState* interface is defined which currently contains methods like *getStatus()* and *sendMsg()*. The concrete implementation of this interface is defined in *OnlineConnectionState* and *OfflineConnectionState* classes. For example, in the *OfflineConnectionState* class, *sendMsg()* function simply returns without sending any messages. This structure also eases the operation of adding new methods related to the connection state of the client in the future. This structure is shown in Figure 8.

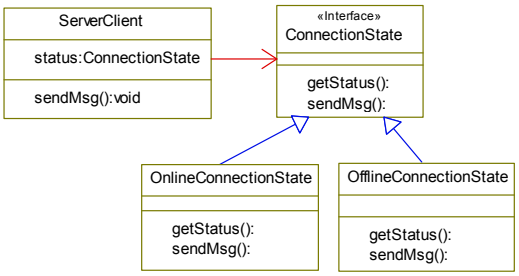


Figure 8 - Class Diagram: State Pattern implementation.

3. Aspect Oriented Programming

An important milestone for software development is to be able to define the object-oriented architecture of the system being developed. After this milestone is accomplished, concerns that seem to be scattered over several modules are identified as crosscutting concerns. Modularizing these concerns not only improves the functionality of the system but also eases the maintenance of the software. For this respect, after completing the object-oriented design of our chat application, we defined the possible crosscutting concerns that may arise during the implementation phase. We adopted the aspect-oriented approach to modularize these crosscutting concerns and developed corresponding aspects. These aspects can be categorized in two main groups which are production and development aspects respectively. Production aspects are those that add functionality to our system where as development aspects tend to ease the development phase of the system. The aspects have been implemented in AspectJ framework and code segments showing implementation details are included. Each of these aspects will be now considered in detail.

3.1 Production Aspects

Production aspects add functionality to a software application by facilitating the implementation of some features resulting in crosscutting concerns. In our system, there were some such features which we decided to implement with aspects. These features can be named as providing notifications and handling exceptions related to message sending. We have defined production aspects

for both of these features which will be described in further detail.

3.1.1 Notification Aspect:

One of the most important features of a chat application is to play several notification sounds to provide a friendlier user interface. Alerts can be used when a client signs in or receives a private message for example. However, the places where alerts are added can be increased. This means that code becomes scattered when an object oriented approach is used. Moreover each time a new notification sound is defined, both the Client code where the new notification will be added, and the class responsible for playing the sound should be changed. In order to overcome these obstacles, we have defined a notification aspect that collects all operations related to alert sounds in one place.

Currently, the notification aspect creates alerts when a client signs in and receives a private message. Therefore two pointcuts and related advices have been defined as shown in Code Fragment 1.

```
pointcut signedIn() : execution(void
      Client.connectionAccepted(..));

void around() : signedIn()
{
   new AudioWavPlayer(signedInAlertFile)
        .start();
    proceed();
}
```

Code Fragment 1: *signedIn* pointcut captures the execution of the *connectionAccepted* method of the *Client* class and the *around* advice plays an alert.

```
pointcut receivedMsg() : call(void
      Client.privateMessageReceived(..));

void around() : receivedMsg()
{
  New AudioWavPlayer(receivedMsgAlertFile)
        .start();
  proceed();
}
```

Code Fragment 2: *receivedMsg* pointcut captures the calls to the *privateMessageReceived* method of the *Client* class and the *around* advice plays an alert.

An additional *AudioWavPlayer* class has been defined to a play a wav file in a separate thread. In the notification aspect, when either of *signedIn* and *receivedMsg* pointcuts is reached, the play method of the *AudioWavPlayer* is called with the corresponding wav file name.

3.1.2 Message Send Failure Handling:

As chatting among clients is the main process of a multi-client chat application, it needs to be done properly. In our systems, private chat windows have been defined to control the chatting operation between two users. When users send messages to each other, necessary methods of the Client, ServerClient and Server classes are called and messages are sent from one user to another in pre-defined formats over a proper network connection. But what if one of the users looses this connection and is abruptly out of the chat? Obviously, due to the usage of the observer pattern, all contact lists containing the client that lost connection are updated but it is necessary for the clients that were chatting with the client during the status change to be explicitly notified.

Although this notification operation can be implemented using standard OO techniques, an implementation at the object level will increase the number of dependencies among classes and thus will result in scattered concerns. Also, at maintenance phases it would be possible to change this notification structure by reacting each time a message could not be sent due to networking problems. So this problem can be seen as a crosscutting concern and thus we have used aspects to enhance our application.

To do this, we defined a point cut that catches all calls made to the *sendPrivateMsg* method of the server class. When this pointcut is captured, the corresponding advice sends a warning message to the client whose message could not been sent. This scenario is implemented in Code Fragment 3.

```
  pointcut msgSendFailedNotification(Server s,
String name, String receiver, String msg) :
     call(* Server.sendPrivateMsg(..)) &&
     target(s) &&
     args(uname, receiver, msg);

  after(Server s, String uname, String
receiver, String msg) throwing():
  msgSendFailedNotification(s, username,
                            receiver, msg)
  {
    System.out.println("exception aspect");
    DataOutputStream dout = s.getDout(uname);
    try
    {
    dout.writeUTF(header + receiver + separator
                + notification);
    }
    catch(IOException ex)
    {}
  }
```

Code Fragment 3: *msgSendFailedNotification* pointcut captures the *sendPrivateMsg* method call in the *Server* class. The advice sends a warning message to the client when the operation returns throws an exception.

Once we have identified calls to our target method, we need to identify the times when this method throws an exception for not being able to send the message

properly through the network. This is done by using the **after throwing** aspect. Using the **after throwing** aspect we can perform necessary actions right after an exception has been thrown by the server. Of course necessary arguments and target objects need to be identified in the aspect in order to force the system to send an automated message to the client sending the message.

3.2 Development Aspects

Developments aspects have the characteristics of facilitating debugging, testing, defining enforcement policies, and performance tuning work. In our project, we had similar needs like profiling system requirements in terms of increase in the number of clients connected to the server and enforcing policies during the implementation phase. For each of these needs, we have defined two corresponding development aspects which are described below.

3.2.1 Profiling System Requirements:

In server-client applications, one of the most crucial issues is how the server behaves when the number of clients connected to the server is above a certain number. Although, we could test our system only with a limited number of clients, in the future it could be used in more sophisticated environments. In such a case, we thought it would be beneficial to have some statistical information such as when a client logs in to the system, how long s/he remains logged in, and when s/he logs out. This information can be used to determine the times of a day when the number of clients increase abruptly so that special care is taken if needed. Moreover, the type of information logged can be extended as new features are added to the system. For example, if a file transferring feature is added, the size of the files being transferred can be logged to determine how the server behaves if a large file is being transferred.

The requirements explained above however cannot be included among the functionalities of the system. They are rather used for development purposes. In addition, these requirements are scattered among the code. To begin with, in order to log information in a file, a file must be created and opened when the server starts and closed when the server stops. Secondly, code must be added to places where the information to be collected is found. For example, to collect information about when a client logs in and out of the system, code must be added to both log in and log out functions. Moreover, if the type of information collected is varied in the future, related code must be added to corresponding places as well. Finally, the option for turning collecting statistical information on and off should always exist. All these issues make the concern of profiling system requirements a good candidate to be implemented with aspects. When an aspect oriented implementation is chosen, the code will not be scattered but collected in the aspect body. Additionally, it will be easy to add and remove this concern to the system by only adding and removing the aspect itself.

As a result, a profiling aspect is added to the system. This aspect creates a log file each time the server is run. In order to create the file appropriately, the aspect defines pointcuts to catch when the server is started and stopped. Then, advices are defined corresponding to these pointcuts to open and close a file. The following code fragments show the described pointcuts and advices:

```
pointcut startServer() : execution(*
                    Server.listen(..));

before() : startServer()
{
  try
  {
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf = new
SimpleDateFormat("yyyyMMdd_hhmm");
    File f = new File("bin\\log" +
sdf.format(cal.getTime()) + ".txt");
    wr = new BufferedWriter(new
FileWriter(f));
    signInTimes = new
Hashtable<String,String>();
  }
  catch(IOException ex)
  {}
}
```

Code Fragment 4: *startServer* pointcut captures the execution of the *listen* method of the *Server* class and the *before* advice creates and opens a log file.

```
pointcut stopServer() : execution(*
                  Server.stopServer(..));

after(): stopServer()
{
  try
  {
    wr.close();
  }
  catch(IOException ex)
  {}
}
```

Code Fragment 5: *stopServer* pointcut captures the execution of the *stopServer* method of the *Server* class and the *after* advice closes the log file.

Currently, the system collects information about the log in and out times of clients. Therefore, two additional pointcut-advice pairs are defined which correspond to places where a client signs in and out. These operations are implemented by the Code Fragments 6 and 7.

```
  pointcut signIn(String username, Socket
socket) :
   args(username, socket) && execution(*
   Server.clientConnected(..));

  after(String username, Socket socket)
returning() : signIn(username, socket)
  {
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf = new
           SimpleDateFormat("hh:mm");
  signInTimes.put(username,sdf.format(
                        cal.getTime()));
  }
```

Code Fragment 6: *signIn* pointcut catches the *clientConnected* method execution in and the advice logs the current time as the sign in time of the client.

```
  pointcut signOut(String username, Socket
socket) :
   args(username, socket) &&
  execution(* Server.removeConnection(..));

  after(String username, Socket socket)
returning() : signOut(username, socket)
  {
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf = new
        SimpleDateFormat("hh:mm");
    try
    {
      wr.write(username + "\t\tsigned in at " +
        signInTimes.get(username) + ", signed
        out at " + sdf.format(cal.getTime()) +
        "\n");
    }
    catch(IOException ex)
    {}
  }
```

Code Fragment 7: *signOut* pointcut captures the execution of the *removeConnection* method and the *after* advice writes the previously logged time and the current sign out time to the log file.

The aspect works as follows. After the startServer pointcut is reached a new file whose name includes the current date and time is created. Each time signIn pointcut is reached, the log in time of the newly connected client is stored in a buffer. When the signOut pointcut is reached, the buffered log in time of the client and the log out time are written to the file. Finally, when the server is closed, the file is also closed. Figure 9 shows a view of a sample log file created.

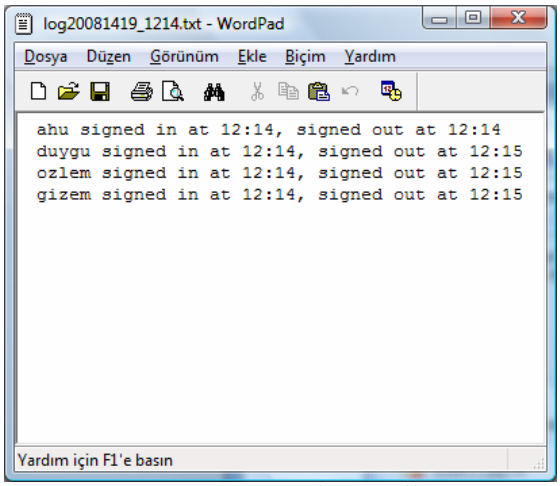


Figure 9. Sample Log File

3.2.2 Access Controlling for Client Class Aspect:

In our implementation of the multi-client chat application our Client class was a very extensive class that had some attributes and methods that by default were meant to be used by classes other than the Server class. But, as explained in the previous section, the Server class is required to store some of the information provided by the Client class.

In order to handle this access problem we had, we decided to add a separate class, *ServerClient*, that only stored Server related attributes and methods. But for a developer, two classes representing the same component might be confusing and it is possible for a developer to confuse these two classes in the development stage. Thus the access to the Client class from the Server class needs to be restricted.

As object oriented approaches have no concepts to handle this issue, we have used aspects. Due to the level of abstraction provided by aspects we are able to catch any illegal access to the Client class made by the Server at compile time and declare it as an error. In order to do this, we have used the *declare error* aspect as shown in Code Fragment 8.

```
declare error : call(Client.new(..))&&
  within(Server)
: "Client cannot be created in Server.";
```

Code Fragment 8: Calls of the *Client* constructor by a *Server* object are captured and considered as an error.

This way, whenever a Client object or any of its static methods is trying to be used by the Server class, a compile time error is produced.

4. Alternative Implementations: Jboss

In our current implementation, we have used AspectJ, one of the most popular frameworks of AOP, for applying aspect-oriented structures to our design. However we could also use some other popular framework like JBoss. However, due to the differences between the structures of AspectJ and JBoss, implementation using JBoss would be quite different.

First of all, the pointcut, and aspect declarations in the two frameworks differ from each other. JBoss supports pointcuts defined by XML or Java annotations where as AspectJ also supports language based pointcuts. Secondly, in order to define an aspect using JBoss, one has to encapsulate the aspect in its own Java class that implements the *Interceptor* interface of the JBoss API. All methods and constructors intercepted this way are turned into a generic *invoke* call. As the interceptors are attached to pointcuts, the methods intercepted are invoked at appropriate places. As an illustration, if we had used JBoss to define the pointcut and advice pair listed in Code Fragment 2, the pointcut definition and the corresponding interceptor would be as in Code Fragment 9.

Furthermore, compiling and deploying aspects in JBoss is more complicated than AspectJ. The compilation process includes compile and run steps separately and the deployment process involves adjustments related to XML files. However, AspectJ's compiler *ajc* does not require a second pass [4] and it has an easy to use plugin with Eclipse IDE tool that we have used [5].

```
public class AlertingInterceptor implements
Interceptor
  {
   public String getName() { return
       AlertingInterceptor; }
   public InvocationResponse invoke
       (Invocation invocation)
       throws Throwable
     {
       new AudioWavPlayer
       (receivedMessageAlertFile).start();
       return invocation.invokeNext();
     }
}

  <bind pointcut="public void Client->
   privateMessageReceived(String name,
   String msg)">
  <interceptor class="AlertingInterceptor"/>
</bind>
```

Code Fragment 9: Pointcut definition and interceptor attachment in JBoss

When we considered the above differences, AspectJ appeared as a more suitable solution for our case. It provided the necessary components to define both dynamic and static crosscutting through an interface we were more familiar with. Due to its popularity, we could collect more technical support for AspectJ than any other framework. JBoss had advantages over AspectJ when working with the JBoss Application Server but we did not need this feature anyway.

5. Related Work

Because Aspect Oriented Programming (AOP) is an immerging technology, it is being started to be used in several different areas of software development. Obviously one of these areas is development of distributed applications. Developing distributed applications involves many different concerns like synchronization, security, fault tolerance etc. Several approaches have been presented for using AOP to deploy these concerns. One such approach is illustrated in [1] where advantages and disadvantages of using AOP in a distributed environment are discussed through a case study. The case study focuses on a framework for investigating and exchanging of algorithms in distributed systems called ALGON. The study concludes that using AOP has many advantages when dealing with complex systems but it should be used with caution. AOP provides good solutions especially for monitoring performance and fault tolerance.

Another approach for using AOP in distributed systems, especially in client-server applications, is illustrated in [2]. This work focuses on security concerns in network-enabled server-client architecture. Security problems may arise when a malicious user has complete access to system resources over a networked computer base. The paper presents a solution by starting with an existing chat application and developing aspects on top of that application with PROSE, a dynamic AOP platform.

Our approach is similar to that of [2], in the manner that we too are developing aspects on top of an existing chat application. However, we tried to develop aspects corresponding to different concerns instead of focusing on only one issue.

6. Conclusion

In this paper, we tried to investigate our work of building a multi-client chat application with AOP approach. The experience we have gained proves the fact that aspect-oriented programming enhances the object oriented design by modularizing crosscutting concerns. Although object-oriented approach helps to modularize system functionalities into different objects, there still remain some concepts that result in scattering. This was also the case in our situation. After designing our system with an object-oriented approach and improving this

architecture with design patterns, there were still some concerns that were scattered in several different places of the code and could be scattered more as the system functionalities increased. Handling exceptions related to sending messages and playing notification sounds were among such example concerns. AOP helped us to implement these concepts as separate aspects without changing the original code.

Besides modularizing crosscutting concerns, AOP also helped us to improve the development phase of our system. As an illustration, through the use of development aspects, we could easily define some enforcement policies to check the correct usage of classes. Furthermore, performance profiling is an important concern for distributed applications and AOP allows this concern to be implemented easily via defining logging and tracing aspects. For example, we implemented a simple aspect for collecting information about the times when clients connected to and disconnected from the server in order to determine when the server becomes highly populated. Similar information can easily be logged to improve the performance of the system.

On the whole, the importance of aspect-oriented approach for client-server applications became solid in our experiences. Even though we have focused only on a simple case of client-server applications, there were several concerns that had a high possibility of resulting in scattered code. These concerns could be expanded if more functionality was aimed. For example, the type of information logged to analyze the performance of the system could be varied as features like file transferring, video conference, and multi-user conversations were added. In addition, the alerts provided could be increased by notifying users when they add or remove friends, send a message to an offline friend, or have been added as a friend. Moreover, if the application were analyzed in a wider perspective, more concerns would be considered as crosscutting. As an illustration, several concerns like security and error handling become crucial for client-server applications, like any other distributed application. To be more specific, both of these concerns affect both the server and the client components of the system. Implementation of these concerns from an aspect-oriented perspective could be interesting. These mentioned points can be considered as feature work and implementation of these points can lead to more interesting points about using AOP in distributed applications.

Acknowledgements

We would like to thank Assist. Prof. Dr. Bedir Tekinerdogan for his comments and suggestions. This work has been carried out in the Aspect-Oriented Software Development class given by Mr. Tekinerdogan.

References

[1] S. Subotic, J. Bishop, and S. Gruner, "Aspect-Oriented Programming for a Distributed Framework", in the *Proceedings of SACJ*, 2006.

[2] P. Falcarin, R. Scandariato, and M. Baldi, "Remote Trust with Aspect-Oriented Programming", in the *Proceedings of the 20th International Conference on Advanced Information Networking and Applications*, 2006.

[3] http://www.damnhandy.com/jboss-aop-vs-aspectj-5-pt-2/

[4] http://www.eclipse.org/aspectj/

[5] Ramnivas Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Manning Publications Co., Greenwich, CT, 2003

Aspectual Development of a P2P Secure File Synchronization Application

R. Bertan Gündoğdu, Kaan Onarlıoğlu, Volkan Yazıcı
Bilkent University Department of Computer Engineering
06800, Ankara, Turkey
{*gundogdu,onarliog,vyazici*}*@cs.bilkent.edu.tr*

Abstract

Development of a file synchronization system requires careful handling of several concerns such as transactions, fault-tolerance, and security. Very often these concerns are hard to modularize into first class elements using traditional software development paradigms. Such cross-cutting concerns decrease the cohesion and increase coupling of the components of the file synchronization system. Aspect-Oriented Software Development (AOSD) provides explicit abstractions to address these cross-cutting concerns. In this paper, we go through the development process of a secure peer-to-peer file synchronization application, FSync, and AOSD techniques to enhance separation of concerns.

1 Introduction

Peer-to-peer (P2P) architecture has emerged as a strong alternative to centralized network architectures that provides scalability, fault-tolerance, and high adaptability to dynamic changes. Contrary to a client-server system where communication is established between a central server and numerous clients, in P2P systems, there is no clear distinction between a client and a server. In other words, all the nodes in the system are considered ”peers” that can act both as clients and servers.

File synchronization process studies the concept of synchronous distribution of a data state between multiple peers. It is essential that all peers subscribed to a certain network have the same data state of the related network at a given time – excluding peers that could not finish their synchronization process for some reason and waiting in an inconsistent state.

One of the key points that must be accomplished during a synchronization process is guaranteeing the reliability of the committed state after data transfer that the state is transfered and committed on the peer side as is. Another one is atomicity of the process[1] – that after any failure a peer should not get stuck in an inconsistent state. It must also be noted that, encryption of the data traffic and authentication of the peers are essential in modern file synchronization systems.

The concerns considered above cannot easily be represented as first class elements by conventional software development paradigms. These concerns are inherently cross-cutting and are scattered among the components of the system, causing tangled code in the implementation. The reason behind this problem is that conventional development paradigms, such as Object-Oriented Software Development (OOSD), do not provide explicit abstractions to address and modularize cross-cutting concerns[9].

The recognition of such cross-cutting concerns that adversely affect the software quality lead to the emergence of a new programming paradigm: Aspect Oriented Programming (AOP). AOP aims to allow separation of cross-cutting concerns by identifying and expressing cross-cutting concerns as first-class elements in a software system. To this end, it provides additional language abstractions, called ”aspects” that explicitly encapsulate cross-cutting concerns.

In this paper, we are going to study the development of a secure peer-to-peer file synchronization application, FSync, by providing design and implementation details. We are first going to present an object-oriented solution for the design task and identify cross-cutting concerns and their implicit impacts that adversely affect the maintainability, understandability and complexity of the system. Next, we are going to apply AOSD techniques to concentrate on modularizing the identified cross-cutting concerns to enhance the overall quality of the previous design and discuss our experiences.

In the rest of this paper we first provide the preliminaries required to understand the concept of the peer-to-peer architecture and of a file synchronization system in section 2. Next, we present a high level object oriented design of the system together with explanations of core components in section 3. Then, we discuss each of the concerns ad-

[1] We assume that the transactions are not network wide, but peer wide.

dressed in the design in detail and identify the cross-cutting concerns and their impacts on the design in section 4. We move on to describe specific implementation issues and provide aspect oriented code samples in section 5 and finally, conclude with the discussions in section 6.

2 P2P File Synchronization

File synchronization studies the methodology of data state propagation among multiple endpoints. It ensures the synchronized identity of data state between distributed peers at a given point in time. Modifications applied to a peer in a synchronization network gets propagated to other peers subscribed to the same network automatically[12].

P2P communication is an popular architecture in modern network topology setups. P2P architectures introduce scalability and fault-tolerance concepts into the system built upon this architecture.

With the above considered architectures and methodologies in mind, file synchronization infrastructures implemented over P2P concepts serve as a successful platform for many real-world IT scenarios. Below you can find concrete usage examples in the field.

1. Mobilized data can be accessed from any peer where the data state synchronization consistency is supplied by the underlying mechanisms[11]. (E.g. Consider distinct employees working on the same project; each can access data from his/her own repository and propagate local changes to other peers in the network.)

2. It enables request load-balancing possible for the related data state. (E.g. It doesn't matter which peer serves an incoming request in a cloud of file servers where data states between servers are synchronized.)

3. It enables fault-tolerant architectures by making it possible to build high-available network of peers. (E.g. In above file server example, failure of a peer doesn't affect the general responsivity of the whole infrastructure.)

4. Synchronized state axiom can be used for backup purposes of sensitive data.

2.1 Implementation Approaches for P2P File Synchronization

In a P2P network, synchronization of the distinct peers implicitly turn out into an unexpectedly complex problem to be solved because of the unreliable nature of the communication environment and peers. There are various software implementation solutions to this problem each having its own advantages and disadvantages. Some of the major solutions that we want to underline are discussed in the rest of this section.

2.1.1 Master-to-Master Database Replication

In this model, multiple (R)DBMSes (Oracle[6], PostgreSQL[13], IBM DB2[8], Microsoft SQL Server, etc.) are configured to propagate each data modification query to other RDBMSes in the network. Such a design should cope with

1. Advanced locking issues (optimistic/pessimistic locking) where every node is accessed and modified simultaneously,

2. Complicated transaction atomicity methodologies. (E.g. 2-Phase commit.)

3. While this implementation makes a perfect fit for the problem, it achieves this goal by issuing complexity in the code and management sides.

2.1.2 Revision Control

Revision control models, keep the track of data state modifications in a structured manner; and to access to a data state in a point in time, it applies the modifications done till that time in an incremental order. A master peer keeps the main repository (with all of its underlying modification history) and slave peers can receive their copy of the data state of a specific point in time.

In this revision control scheme, any change must be reflected on the *master* peer and than manually propagated to slave peers.

2.1.3 Distributed Revision Control

In distributed revision control systems (git[4], darcs[3], bazaar[1], mercurial[2], etc.), every user of keeps a local copy of the repository. One can propagate changes belonging to an other repository by his/her own prompt.

While this scheme supplies a perfect environment for working with groups of individuals involved in distinct code parts (e.g. every user can have his/her own branching, tagging state), it makes it nearly impossible (and infeasible) to have each node at the same data state by design. To summarize, in this implementation it requires manual prompt of peers to receive modifications from a *specific* peer.

2.1.4 Distributed File System

While so far mentioned models run in the user level, distributed file systems[2] (e.g. OpenAFS[14], etc.) sit between

[2] Assuming file system peers are synchronized, not owners of distinct data parts.

user and kernel level[3], providing an easy to use traditional file system to the user.

This model has the same advantage, disadvantage and implementation complexities as master-to-master database replication. One more advantage of distributed file systems over master-to-master database replication is that the user isn't restricted to use an (R)DBMS, instead, he/she can benefit from the functionality of a traditional file system interface.

3 FSync: Secure P2P File Synchronization

3.1 Overview

FSync is a secure peer-to-peer file synchronization application that provides synchronization of data files across a virtual file sharing network, called a "sync network". Computers join sync networks to become "nodes" in that network and synchronize with the shared content on the network. A node can be in a single sync network or can share different content on several different sync networks; which means overlapping sync networks are possible. The system adopts a peer-to-peer architecture; in other words there is no master coordinator controlling the synchronization of the nodes; instead each node issues updates to the shared content and these local changes are propagated to the peer nodes in that sync network. Note that FSync is very different from a "versioning system" such as CVS in that it altogether avoids doing versioning, file locking through check in/out, etc. Moreover, in a CVS structure updates are pulled by clients from the server, whereas in FSync, updates are pushed onto the network by the peers. It is a lightweight application to synchronize personal content on a network in a secure and fault tolerant way.

3.2 Requirements Analysis

We have identified the core requirements of FSync as follows:

Sync Network Management: Application allows users to create sync networks, specify their names and set up passwords to control access to the network.

Joining/Leaving a Sync Network: Users can become nodes in a sync network by joining that network. Since access to the sync networks are controlled, a user must specify a valid password to be eligible to share content on that network. Users that have become nodes in network can leave that network.

File Synchronization: Users issue updates to the shared content and the rest of the nodes belonging to the corresponding sync network synchronize with the updated content.

Transactional File Transfer: During a synchronization process, file transfers are handled as a single transaction; that is to say, the local content on a node is updated only if all of the updated content is retrieved and verified through an integrity check.

Secure Network Association: Joining a network is performed through a cryptography based authentication protocol. The protocol does two-sided authentication; that is both the joining node and the node that accepts the new node into the sync network are authenticated to prevent malicious connections.

Secure File Transfer: Files are encrypted before being transmitted over a sync network. Encryption could be performed by a selection of popular symmetric encryption algorithms, which is specified while creating the sync network. The symmetric key to be used for encrypting the files is exchanged securely during the secure network association protocol.

Persistence: The application offers persistent storage of sync network management data together with the state of the synchronized folder content.

3.3 Object-Oriented Design

The class diagram of the final system, expressed in UML, is provided in Figure 1. Note that, this diagram is simplified to only include the essential details for the sake of clarity.

The core components of the system are briefly explained below:

FSync: The top-level component of the system. It manages the rest of components, acts as a bridge over several components of the system and provides an interface for the graphical user interface to interact with the application.

SyncNetwork: Represents a sync network and the local machine that acts as a node in that network. It stores all the essential data to identify a sync network, performs management of network nodes and issues update notifications to nodes during a synchronization request by the user.

Node: Represents a remote node in a sync network. It contains all the information to identify and access a node on a physical network connection. Note that the local machine is not represented as a Node object but as a SyncNetwork object, as explained above.

Encryption: The abstract parent class that represents an encryption box. It stores the encryption parameters such as the symmetric keys. This is class is sub-classed to implement encryption and decryption facilities using different cryptography algorithms.

ConnectionManager: Handles transmission of control messages. It is responsible for handling and issuing con-

[3] Actually it's also possible to implement file systems in user level using FUSE (Filesystem in Userspace) like APIs.

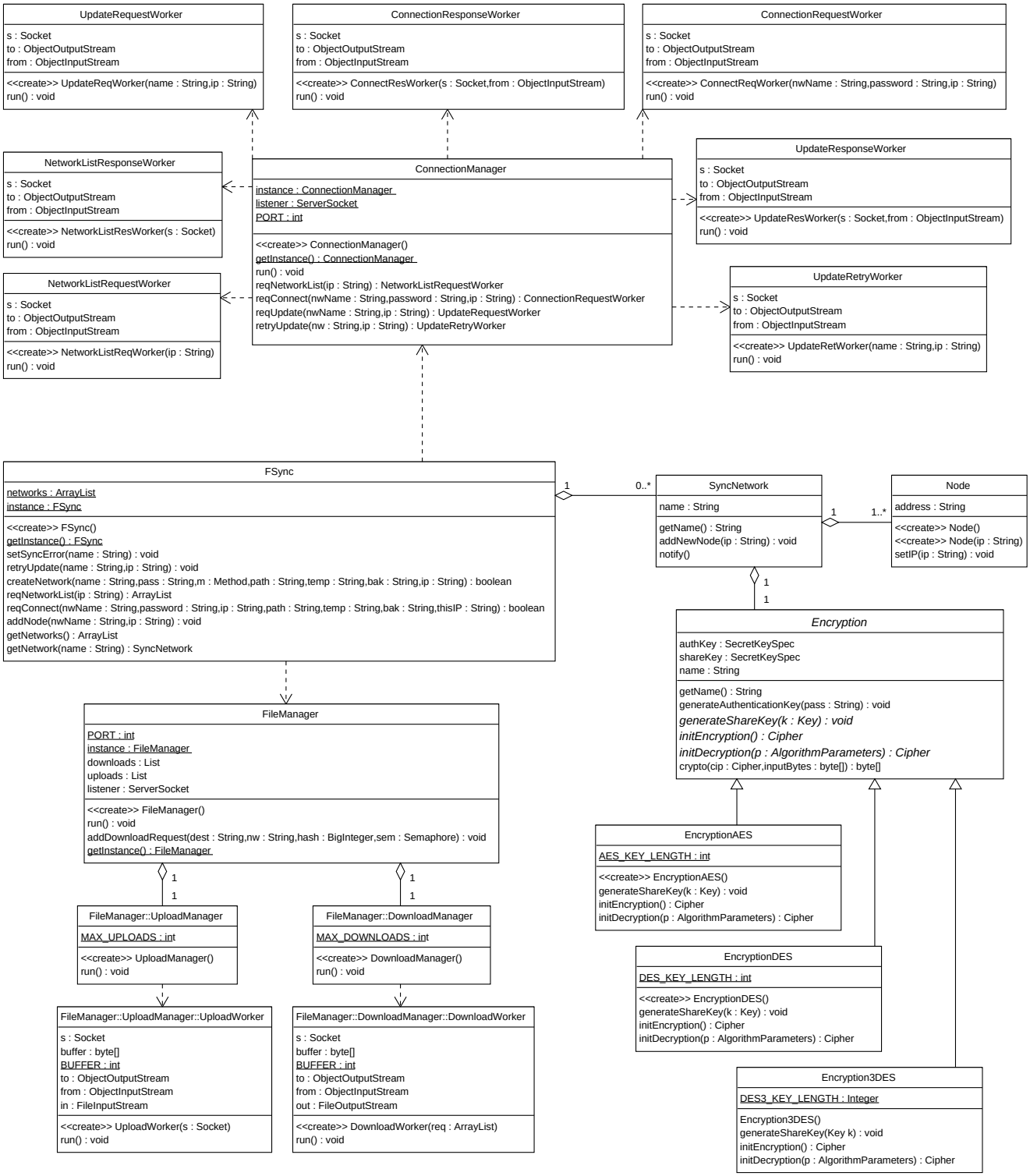


Figure 1. UML Class Diagram Depicting the Object-Oriented Design of FSync

nection requests, update notifications and retry requests. All the functionality provided by this class is performed by separate worker threads, as illustrated in the UML diagram.

FileManager: Handles transmission of files. It is respon-

sible for issuing download requests to remote FileManagers and serving uploads requests arriving from them. All the functionality provided by this class is performed by separate worker threads, as illustrated in the UML. FileManager is capable of downloading and uploading in parallel connections, number of which could be specified by the user.

The object oriented design benefits from a number of well-known design patterns.

Observer Pattern

An observer pattern is used for coupling SyncNetwork and Node objects in an abstract and oblivious fashion. Whenever the synchronized content is changed in a sync network, the notify() method of the corresponding SyncNetwork object is invoked, which then invokes the update() method of each of its Node objects. (See *Figure 2. Application of Observer Pattern in FSync*)

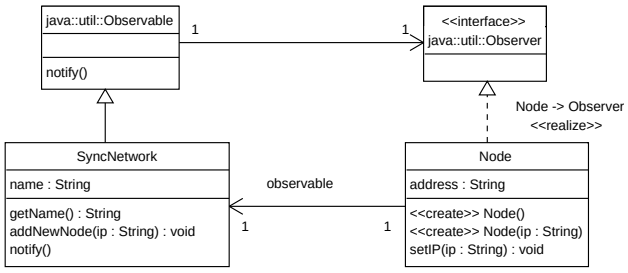


Figure 2. Application of Observer Pattern in FSync

It should be noted that the pattern applied in our case differs from the general case in that, the Node object represents a remote machine and thus it forwards the "Update" notification to the corresponding machine with the help of ConnectionManager. Then the following "Get State" message will be received again by the ConnectionManager and relayed to the SyncNetwork. In short, our case applies the observer pattern over a TCP connection, similar to Java Remote Method Invocation.

Strategy Pattern

The strategy pattern is used for providing several variant encryption algorithms with a uniform interface without affecting the rest of the system. It is used for providing DES, AES and 3DES encryption algorithms to be used in file transfer. (See *Figure 3. Application of Strategy Pattern in FSync*)

Singleton Pattern

FSync, ConnectionManager and FileManager classes are defined to be singleton classes; in other words only one instance of each can be constructed. This is essential since they use specific network and system resources that must only be accessed by a single thread of execution at a time[7].

4 Aspect Oriented Design

4.1 Problems with the Object-Oriented Approach

In the previous section we have presented an object oriented design of FSync that aims to support the quality of the system by applying the well-known object-oriented design principles: abstraction, decomposition, encapsulation, information hiding and separation of concerns. We have strived to minimize the coupling among the components of the system and provide high cohesion within. To this end, we have applied several object-oriented design patterns to support the quality factors affecting FSync.

At this point, we can safely assume that we have achieved a strong object-oriented design based on well-established object-oriented design techniques; however despite our best effort it is still possible to identify weaknesses in the system design. These are primarily caused by concerns that cannot be expressed as first class elements, in other words cross-cutting concerns, due to the inherent inadequacy of the object-oriented approach.

In the light of the requirements analysis we have provided in section 3.2, we can identify the concerns that must be expressed in the system as follows: Sync network management, file transfer & synchronization, authentication, encryption, transactions & data integrity, persistence and finally, graphical user interface updates and notifications. We shall now discuss whether each of these concerns can be expressed as first class elements in our design and demonstrate some of the problems that cannot be resolved even by high-quality object-oriented design through change scenarios on the system.

4.2 Concerns Expressed as First-Class Elements

4.2.1 Sync Network Management

Sync network management involves creating new sync networks and joining/leaving sync networks.

A sync network is expressed as a single component in the system by the SyncNetwork class; similarly nodes that belong to the networks are represented by the Node class.

Connection responsibilities are performed by ConnectionRequestWorker and ConnectionResponseWorker classes which are coordinated by the ConnectionManager. Each of these components only handles a single step of the connection process and they do not have interactions with each other.

We can conclude that, each physical computer using the system can be represented in the software by highly cohesive modules with a one way coupling link. Moreover, each

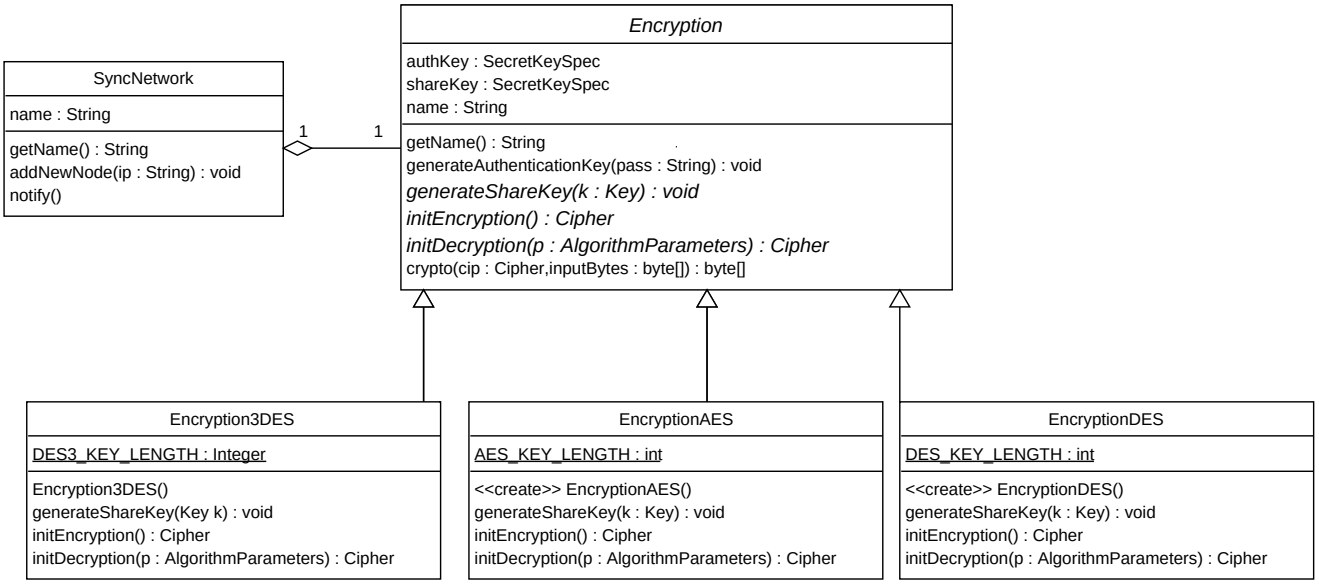


Figure 3. Application of Strategy Pattern in FSync

separate step of the connection protocol is handled by separate modules with well-defined responsibilities. Therefore, OOD can decompose this concern into sub-steps and can encapsulate each of them in separate modules with sufficient quality.

4.2.2 Synchronization & File Transfer

Synchronization & file transfer concern involves notifying nodes in a sync network of updates in the shared content and propagating the updated content to those nodes.

The notification mechanism is very well handled with the aid of the observer pattern, as discussed in section 3.3, so that it is already expected to be a strong part of the design. The communication of the modified or new content is performed by the UpdateRequestWorker and UpdateResponseWorker classes; which then issue download/upload requests to the FileManager.

File transfers are performed by the UploadManager and DownloadManager classes, which use a number of workers to transfer data in parallel. These are also coordinated by the FileManager class, so that they are lowly coupled components with a single specific task to perform.

Therefore, synchronization & file transfer is another concern that can be realized by using the OOD paradigm without violating the good design principles.

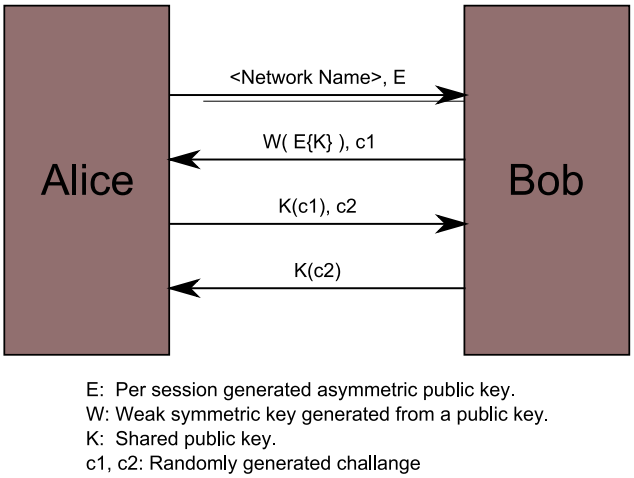


Figure 4. EKE (Encrypted Key Exchange) Protocol

4.3 Cross-cutting Concerns

4.3.1 Authentication

Authentication concern involves running a cryptography based protocol to ensure a node is eligible to join a remote sync network, and in turn ensuring that the remote point is a valid sync network. During this protocol the symmetric key to use for file encryption is also exchanged between the nodes. We have used an authentication protocol based on the well-establish Encrypted Key Exchange (EKE) protocol[5]. (See *Figure 4. EKE (Encrypted Key Exchange) Protocol*)

As demonstrated in *Figure 4* above, the EKE protocol requires two rounds of data exchange between the authenticating nodes. Since this is also a part of the sync network connection protocol, it naturally is implemented within the ConnectionRequestWorker and ConnectionResponseWorker classes, which means that the authentication concern is scattered among two components in the system. Additionally, since the original responsibilities of these workers was to handle connection requests but not to perform complex cryptographic protocols, introducing the authentication concern into their implementation causes tangled code in those modules. We shall demonstrate the effects of this problem with a change scenario:

Scenario: Replace the EKE protocol with a simple password based login protocol to improve performance in a trusted zone.

Impacts: run() methods in the ConnectionRequestWorker and ConnectionResponseWorker need to be changed dramatically. If we want to provide both protocols as options, then we need to clutter these methods with many conditional statements. The changes on the two modules must be strictly symmetric, otherwise we may break the connection protocol altogether by sending messages different from what the receiver expects next; which may give rise to security flaws such as unauthorized access as well.

4.3.2 Encryption

Encryption concern involves encrypting the files to be transfer over a physical network to maintain secrecy. It could be performed using an algorithm specified by the user among provided algorithms.

The Encryption algorithm is represented by its own class Encryption, and thanks to the strategy pattern switching between different algorithms is not an issue. However, this abstraction and the strategy pattern do not help prevent this concern from scattering. Since the files are stored in clear in disk, they must be encrypted just before the upload process. In a similar way, received files must be decrypted before being written to the disk. Consequently, UploadWorker and DownloadWorker must access the Encryption object and make use of the cryptography services it provides. Moreover, the upload worker must send a set of algorithm specific encryption parameters that are randomly generated each time a file is to be encrypted before transferring the file, and the corresponding DownloadWorker must receive this data and set up its Encryption object for proper decryption. Finally, since each sync network can possibly use a different encryption algorithm with different parameters, a SyncNetwork object must properly initialize its Encryption object upon creation. As a result, this concern is scattered among Encryption, SyncNetwork, UploadWorker and DownloadWorker classes, leading to tangled code in each of them. The following change scenario expresses the effects of this problem:

Scenario: Offer the option to bypass encryption for significantly faster synchronization of insensitive data.

Impacts: In order to support this feature we have to modify the run() methods of DownloadWorker and UploadWorker classes, to bypass invocations to the Encryption object and prevent transferring the encryption parameters. Again the changes should be strictly symmetric in order to keep the protocol running correctly. Another approach would be to introduce a dummy Encryption sub-class NoEncryption, but creating an Encryption object which actually does no Encryption would be a poor design choice. Moreover we may need to modify the constructor of the SyncNetwork class to properly initialize this new subclass.

4.3.3 Transactions & Data Integrity

This concern involves making the synchronization operations atomic and prevent going to an inconsistent state by first checking the integrity of the files received and then updating the local share.

Transaction operations are performed inside the UpdateResponseWorker since this is the worker class that checks for updated content and requests download of the modified or new files from the FileManager. After all downloads are finished, the integrity check is performed. If an error is detected, the downloaded content is discarded and the UpdateRetryWorker requests a new update by restarting the process. Not only is this concern scattered among these two components, it introduces heavy tangling code into the UpdateResponseWorker whose original task is to simply issue download requests to the FileManager. We shall demonstrate the problem with a change scenario:

Scenario: To support efficient transfer of large amounts of files, offer the option to adopt a best-effort download approach and in case of error, instead of retrying the whole update, only request download of erroneous files.

Impacts: Not only does this change require modification of the run() methods of UpdateResponseWorker and UpdateRetryWorker, but it also amplifies the problem of scattering and tangling by increasing the scope of cross-cutting: now UpdateRequestWorker should also be modified to support partial update requests.

4.3.4 Persistence

Persistence concern involves persistently storing the latest state of the synchronized folders and the list of sync networks when the system is shut down. Then, when the application is run again, these data are loaded.

The loading operation obviously must be the first task to perform once the system components are initialized and it

suffices to perform this task only once each time the application is launched. However, the saving operation is much harder to address, it must be performed each time a state change occurs. These include creating a new sync network, joining a new network, welcoming another node to a network, and committing or receiving synchronization updates. The components where these changes are observed greatly vary; FSync, SyncNetwork, UpdateRequestWorker, UpdateResponseWorker and Encryption are among the components where the persistence task is scattered among. The below change scenario expresses this problem:

Scenario: Add new fields to the SyncNetwork and make them persistent: last update date, last update issuer, version, etc.

Impacts: Apart from the task of adding these fields and methods to set/get their values, making these persistent requires populating these methods with persistence code and tangling the methods. Moreover, the load operation must also be modified to read and load the new data accordingly. The scope of cross-cutting in similar scenarios is not easy to guess, since any of the components could possibly have persistent data.

4.3.5 GUI Updates and Notifications

This concern involves updating the GUI elements according to the state of the application and display notification indicators when necessary.

This concern has a vast scope since there are numerous possible things that can be expressed to the user of the application. In our implementation, we chose to display the list of current sync networks the application is associated with, the details of these networks and details of the nodes in them. We also display notifications whenever synchronization starts, completes successfully or terminates with an error. It is easy to see that the components that are aware of these states vary greatly; for instance, only FSync knows when a new sync network is created, and only the UpdateResponseWorker can determine the end of a synchronization period. Consequently, in our implementation the task of updating the GUI needs to be addressed in almost all of the worker classes and additionally in FSync. As a result, this is probably the most scattered cross-cutting concern which causes serious tangling in each of these modules. The problem is demonstrated in the following change scenario:

Scenario: Add progress indicator bars to display the synchronization progress in percentages.

Impacts: This requires adding progress bar update calls in each of the UpdateResponseWorker, UploadWorker and DownloadWorker classes. As seen, adding a new GUI notification feature requires modification of multiple components and the scopes of the changes depend on what feature is added.

4.4 Evaluating the Object-Oriented Design and Identifying Aspects

The detailed examination of the core concerns handled in the system, in the previous part, shows that the object oriented approach alone is incapable of addressing cross-cutting concerns. This results in overall low quality and hurts complexity maintainability, understandability and re-usability. Tackling the problem of cross-cutting concerns is not possible using OOD techniques; because OOP paradigm fails to express cross-cutting concerns as first-class elements. Thus, making use of the AOP paradigm, which provides explicit abstractions to handle cross-cutting concerns, is necessary.

Taking into account the cross-cutting concerns identified in 4.3, we have defined an aspect for each of them: Authentication Aspect, Encryption Aspect, Transaction Aspect, Persistence Aspect and UIAspect. Specific implementation details of these aspects will be presented in the next section.

5 Aspect-Oriented Programming

We have refined our object oriented design to address the issues discussed in the previous section and implemented the system using aspect oriented programming techniques. Our choice of implementation language was AspectJ; thanks to its popularity, good support by development environments and simplicity as a tool for introduction to the concept of aspect orientation[10].

In this section we will briefly discuss the implementation details of two of the aspects identified above: Authentication Aspect and Encryption Aspect, together with simplified AspectJ code samples. Note that we will use `Java♯` syntax; which is an imaginary syntax specification that allows English statements preceded by ♯, where details of implementation need be avoided.

5.1 Authentication Aspect

This aspect implements the routines required to realize a successful authentication protocol during the sync network connection process. As we have already pointed out, the connection protocol is performed between a ConnectionRequestWorker and a ConnectionResponseWorker object, in the corresponding run() methods. Therefore, we need to specify pointcuts to capture the appropriate join points in those functions.

Listing 1. Authentication Aspect Pointcuts

```
public aspect AuthenticationAspect {
    // variable declarations
    ...
    // pointcut definitions
    pointcut newOInputStream(ConnectionRequestWorker w):
        set(InputStream ConnectionRequestWorker.from)
        && this( w);
    pointcut newOutputStream(ConnectionResponseWorker w):
        set(OutputStream ConnectionResponseWorker.to)
        && this( w);
    // advice definitions
    ...
}
```

Pointcuts in the *Listing 1. Authentication Aspect Pointcuts* capture establishment of communication pipes that are going to be used during the connection protocol. Then, before the rest of the connection protocol could be carried out between the ConnectRequestWorker and ConnectResponseWorker, the authentication protocol is run as an advice as in *Listing 2 Authentication Aspect Join Points*.

Listing 2. Authenticatio Aspect Join Points

```
public aspect AuthenticationAspect {
    // variable declarations
    ...
    // pointcut definitions
    ...
    // advice definitions
    after( ConnectionRequestWorker worker)
        : newInputStream( worker) {
            # send E;
            # receive W( E{K}) and  c1;
            # extract K, send K(c1);
            # receive and verify K(c2);
    }

    after( ConnectionResponseWorker worker)
        : newOutputStream( worker) {
            # receive E;
            # send W( E{K}) and  c1;
            # receive and verify K(c1), send K(c2);
    }

    // support methods
    // performs encryption operations
    private byte[] crypto( ... ){...}

    // encrypts a given key
    private byte[] keyWrap( ... ) {...}

    // decrypts a given key
    private Key keyUnwrap( ... ) {...}

    // computes a key from a password
    private SecretKeySpec deriveKey( ... ) {...}
}
```

Note that the advice contains several regular methods that are used in various stages of the protocols. Details of the advice and method contents are not provided since a cryptography background is beyond the scope of our intents.

This implementation localizes all the authentication operations inside the aspect, and results in the connection workers being unaware of the fact that an authentication protocol is running. This is a useful feature that helps us change the authentication protocol or remove it altogether without disrupting the intended connection operation.

5.2 Encryption Aspect

This aspect provides secrecy during file transfers by encrypting the transferred data. As we have discussed above, the task of sending and receiving files is performed by UploadWorker and DownloadWorker objects. Therefore, we have to define pointcuts to capture appropriate joinpoints during the file transfer operations.

The first task is to agree on the encryption parameters to be used. In our example, we will use a single parameter for this purpose, the Initialization Vector (IV). IV is a large random integer value that is roughly used for randomizing the encryption process to make it more secure. It is determined or generated by the encrypting side, and must be transferred to the decrypting side for proper decryption. The IV itself is not secret and could be transferred in plain text without affecting the security of the cryptosystem.

As a result, we need to initialize our cryptosystem, generate an IV on the encrypting side and then send it to the receiving side so that it can use the IV to initialize its own cryptosystem for decryption. This is done as demonstrated in *Listing 3. IV Initialization in Encryption Aspect.*

Listing 3. IV Initialization in Encryption Aspect

```
public aspect EncryptionAspect {
    // variable declarations
    private Cipher DownloadWorker.cipher;
    private Cipher UploadWorker.cipher;
    // pointcut definitions
    pointcut sendIV( UploadWorker w):
        set( ObjectOutputStream UploadWorker.to)
        && this( w);

    pointcut getIV( DownloadWorker w):
        set( ObjectInputStream DownloadWorker.from)
        && this( w);
    ...

    // advice definitions
    // initializes the encryption and sends IV
    after( UploadWorker w): sendIV( w) {
        w.cipher = getNetwork( w.name)
            .encyption
            .initEncryption();
        Parameters params = w.cipher.getParameters();
        w.to.writeObject( params.getEncoded());
    }

    // receives IV and initializes decryption
    after( DownloadWorker w): getIV( w) {
        byte[] param = (byte[])w.from.readObject();
        Encryption encryption = getNetwork( w.name)
            .encryption;
        Parameters ap = Parameters.getInstance();
        ap.init( param);
        w.cipher = encryption.initDecryption( ap);
    }
}
```

The pointcuts defined above take over the control as soon as the communication pipes between the workers have been established and realize the IV exchange. Note that the aspect also addresses static cross-cutting by declaring a member Cipher in each of the workers, which store the current encryption status and parameters.

The next task is to actually encrypt and decrypt the data. This is performed as in *Listing 4. Forward/Backward Encryption Methods in Encryption Aspect*.

Listing 4. Forward/Backward Encryption Methods in Encryption Aspect

```
1  public aspect EncryptionAspect {
2      // variable declarations
3      ...
4      // pointcut definitions
5      pointcut encryptBuffer( UploadWorker w):
6          call( * FileInputStream.read(..))
7          && this( w);
8
9      pointcut decryptBuffer( DownloadWorker w):
10         call( * ObjectInputStream.readFully(..))
11         && this( w);
12     ...
13     // advice definitions
14     // encrypt data buffer
15     after( UploadWorker w): encryptBuffer( w) {
16         SyncNetwork sn= getNetwork( w.name);
17         w.buffer = sn
18             .encryption
19             .crypto( w.cipher, w.buffer);
20     }
21
22     // decrypt data buffer
23     after( DownloadWorker w): decryptBuffer(w) {
24         SyncNetwork sn= getNetwork( w.name);
25     }
26 }
```

The above code sample captures every read file statement on the uploader and every receive data statement on the downloader. Then the filled data buffer is retrieved from the workers, contents are encrypted or decrypted, and the buffer is filled with the processed content. In this way, the file transfer operation is performed with absolutely no knowledge of the encryption and the aspect could be removed entirely without disrupting the normal file transfer. Moreover, all the encryption task is localized in the aspect.

Finally, we have said that each SyncNetwork object stores its corresponding Encryption object. We would also like this coupling and initialization of the Encryption object to be localized in this aspect, as demonstrated in *Listing 5. Introducing SyncNetwork in Encryption Aspect*.

Listing 5. Introducing SyncNetwork in Encryption Aspect

```
1  public aspect EncryptionAspect {
2      // variable declarations
3      private Encryption SyncNetwork.enc;
4
5      // pointcut definitions
6      pointcut newSyncNetwork( String pass,
7                               Encryption.Method m,
8                               SyncNetwork nw):
9          execution( SyncNetwork.new(..))
10         && this( nw)
11         && args( .., pass, m);
12     ...
13     // advice definitions
14     // instantiate an encryption object
15     after( String pass,
16            Encryption.Method m,
17            SyncNetwork nw):
18         newSyncNetwork( pass, m, nw) {
19
20         if( m == Encryption.Method.DES)
21             nw.enc = new EncryptionDES();
22         else if( m == Encryption.Method.AES)
23             nw.enc = new EncryptionAES();
24         # else if...
25
26         nw.enc.generateShareKey();
27         nw.enc.generateAuthenticationKey( pass);
28     }
29 }
```

This code sample demonstrates introduction of a new member variable Encryption to the SyncNetwork class through static cross-cutting and initializing it with the help of an advice that executes after a new SyncNetwork has been instantiated.

6 Discussion

- In this paper we have studied Aspect-Oriented software development on a real-world application and experienced its uses. While AOSD certainly contributed advantages to the separation of cross-cutting concerns into suitable modules, it also extended the flexibility of the application. For instance, it is sufficient to exclude the encryption aspect from source code to remove the encryption feature from the system and the application will still compile and run. Moreover, this flexibility applies to almost every aspect (e.g. Authentication, Transaction, User Interface, etc.) we used.

- It must also be noted that, one needs to be aware of the detailed uses of every aspect in the system to be able to not to conflict with pointcut definitions and to conform with aspect conventions. This obligation requires a highly strict and clear conventions for coding.

- During development, we experienced that pointcuts relying on the code fragments strictly dependent to the local conventions used at that point are more vulnerable to the programmer mistakes and result in hard to spot bugs. This issue can be solved by improving the functionality of the existing development tools (e.g. more verbose display of pointcut relations) and/or reconsidering the design of the pointcuts so that the lexical dependency is relaxed.

- User interface design and implementation requires relatively huge amount of work in the code place, and generally a majority of the graphical component actions and concerns tend to overlap. During development of FSync, we saw that such cases can be

addressed and implemented aspect-oriented programming with ease, resulting in cleaner code. For instance, close and cancel operations of dialogs can be shared, hence can be implemented within suitable aspects. Moreover, business logic does not have to contain any user interface related code. Consequently, gluing of the business logic and user interface can be implemented in aspects, instead of tangling the code throughout the system.

- Finally, we would like discuss a new alternative technology to AspectJ, that is JBoss. JBoss is a full-featured Java Application Server with enterprise quality support for different programming paradigms, one of which is AOP. Unlike AspectJ, JBoss does not introduce any new keywords to the language, but uses native Java syntax. Pointcuts are defined in XML and the weaving is dynamic and performed at run time. As an example, the pointcuts in our Authentication aspect would look like as in *Listing 6. JBoss Alternative* in JBoss.

Listing 6. JBoss Alternative

```
<?xml version="1" encoding="UTF–8"?>
<aop>
  <bind
    pointcut="set(ObjectInputStream
                  ConnectionRequestWorker->from)">
     <interceptor
       class="ConnectionRequestWorker->doAuthentication" />
  </bind>
</aop>
```

Note that, interceptors in JBoss replace the advises in AspectJ, and they are defined similar to regular Java methods. In this case, when pointcut matches the join-point the `doAuthentication` method in `ConnectionRequestWorker` class gets executed.

7 Acknowledgements

We would like to thank Asst. Prof. Dr. Bedir Tekinerdoğan, for his efforts and for organizing the Turkish Aspect-Oriented Software Development Workshop 2008.

8 Conclusion

In this paper, we have developed a peer-to-peer secure file synchronization system and studied the application of aspect-oriented software design. We first provided the object-oriented design and then elaborated on the process of identifying, specifying and implementing aspects to handle cross-cutting concerns and enhance the design quality of FSync.

As a result of our studies, we have observed that aspect-oriented software design paradigm indeed addresses the problem of scattering and tangling in the object oriented design of peerto- peer file synchronization applications by explicitly expressing the cross-cutting key concerns. Despite the discussed advantages, during the development we have found out that AOP requires dedicated awareness of the aspect joinpoints, otherwise pointcuts might easily result in hard to spot bugs in the program.

In conclusion, many core requirements of a peer-to-peer file synchronization system could be addressed by using AOSD to enhance modularization by seperating cross-cutting concerns and to provide a design that supports software quality factors.

References

[1] Bazaar version control. http://bazaar-vcs.org/.

[2] Distributed version control system. http://www.selenic.com/mercurial/.

[3] Free, open source source code management system. http://darcs.net/.

[4] Git - fast version control system. http://git.or.cz/.

[5] S. M. Bellovin and M. Merritt. Encrypted key exchange: password-based protocols secure against dictionary attacks. pages 72–84. IEEE, 1992.

[6] O. Corporation. Oracle real application clusters 11g technical overview. http://www.oracle.com/technology/products/database/clustering/pdf/twp_rac11 2007.

[7] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[8] L. J. Gu, L. Budd, A. Cayci, C. Hendricks, M. Purnell, and C. Rigdon. *A Practical Guide to DB2 UDB Data Replication V8*. IBM Redbooks, 2002.

[9] W. L. Hursch and C. V. Lopes. Separation of concerns. Technical report, 1995.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[11] D. Malkhi, L. Novik, and C. Purcell. P2p replica synchronization with vector sets. *SIGOPS Oper. Syst. Rev.*, 41(2):68–74, 2007.

[12] C. Mastroianni, G. Pirrò, and D. Talia. A hybrid architecture for content consistency and peer synchronization in cooperative p2p environments. In *InfoScale '08: Proceedings of the 3rd international conference on Scalable information systems*, pages 1–9, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[13] G. S. Mullane and E. P. Corporation. Multi-master asynchronous postgresql replication system. http://bucardo.org/.

[14] C. M. University and I. P. Labs. Open source implementation of the andrew distributed file system. http://www.openafs.org/.

Aspect-Oriented Development of a P2P File Sharing Application

Eren Algan, Ridvan Tekdogan, Dogan Kaya Berktas

Department of Computer Engineering,
Bilkent University, Ankara, TURKEY

{erenalgan, ridvantekdogan, dkberktas} @gmail.com

Abstract*—P2P applications become popular since the number of the internet users have increased. There are many concerns in these applications like concurrency, monitoring etc. Most of the concerns can modularized with OO development. However, there are some concerns that crosscut each other and Object Oriented design cannot cope with these crosscutting concerns. In this paper, we propose a new P2P application developed with Aspect Oriented design that copes with these crosscutting concerns.*

***Index terms*—Aspect-Oriented Programming, P2P, Napster**

1. Introduction

File sharing is the public or private sharing of computer data or space in a network with various levels of access privilege. P2P model is one of the models being followed by file sharing applications and yet P2P file sharing applications have been one of the most widely used applications among Internet users [1].

P2P applications run in the background of user's computers and enable individual users to act as downloaders, uploaders, file servers, etc. Designing and implementing P2P applications raise particular requirements. On the one hand, there are aspects of programming, data handling, and intensive computing applications; on the other hand, there are problems of special protocol features and networking, fault tolerance, quality of service, and application adaptability. While handling with these problems, it is important to take crosscutting concerns into account. Object Oriented designs cannot cope with crosscutting concerns. Hence, Aspect Oriented design is the solution of crosscutting concerns and yet we have designed and implemented a P2P File Sharing Application with an Aspect Oriented development.

The main idea of this study is to apply Aspect Oriented design to a P2P file sharing application, which uses Napster protocol, to solve crosscutting concerns.

The paper is organized as follows. Basic information to understand P2P file sharing applications and design patterns will be given in section 2. Example cases will be focused in detail in section 3. The approach of our system will be discussed in section 4. Problem statement will be detailed in section 5. Since there are many examples of our study, related work will be mentioned in section 6. Difficulties encountered during study will be mentioned in section 7. And finally, we will conclude our work in section 8.

2. Case Study: P2P Application

2.1 Example P2P Application

In this section we describe an example P2P application. There are various protocols designed for sharing files over Internet. The protocols can be categorized into three groups according to their network topology. 1) Napster style protocols: Clients are connected to a server for informing about their shares and for searching downloads. 2) Gnutella style protocols: Clients are connected to each other and queries send to peers and peers forward them to their neighbors until the result found. 3) Hybrid approach: Clients are clustered and select a cluster head which has responsible as server in Napster. Since in this paper we propose solution for Napster style P2P sharing application our example will be also about that.

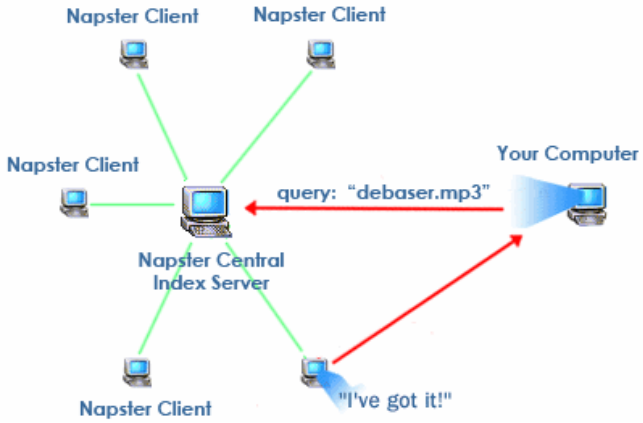


Fig 1 Example Scenario for Napster protocol

As shown in the Fig 1 there are two kinds of participant in Napster style P2P file sharing application. One of them is client and the other server. Server does not have capability of searching and sharing. It only gathers information and performs search operation. Clients connect to a server by entering its address at the

beginning and then shared folder information is sent to server. Server keeps user and his/her shared file information as long as she/he stays connected to server. When client disconnected or a connection problem occurs between client and server; server cleans the information about that client. Clients search shared files through server by sending a command. Server returns search result with owner peers' information. Client can either download from single peer or from multiple peers by directly establishing connection.

A sample OO design of P2P file sharing application is given in Fig 2 for server application and in Fig 3 for client application.

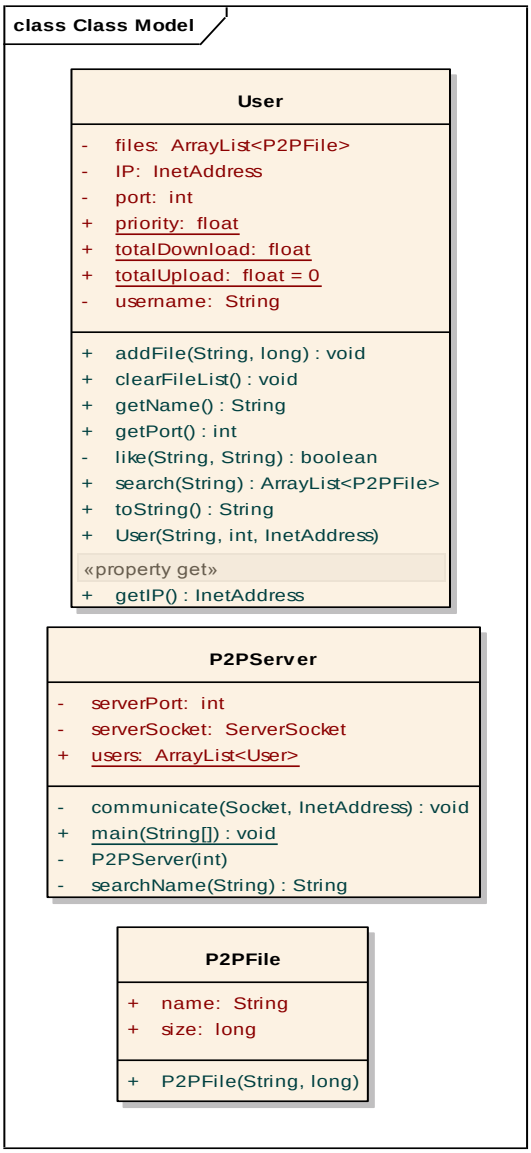


Fig 2 Class Diagram of Client Application

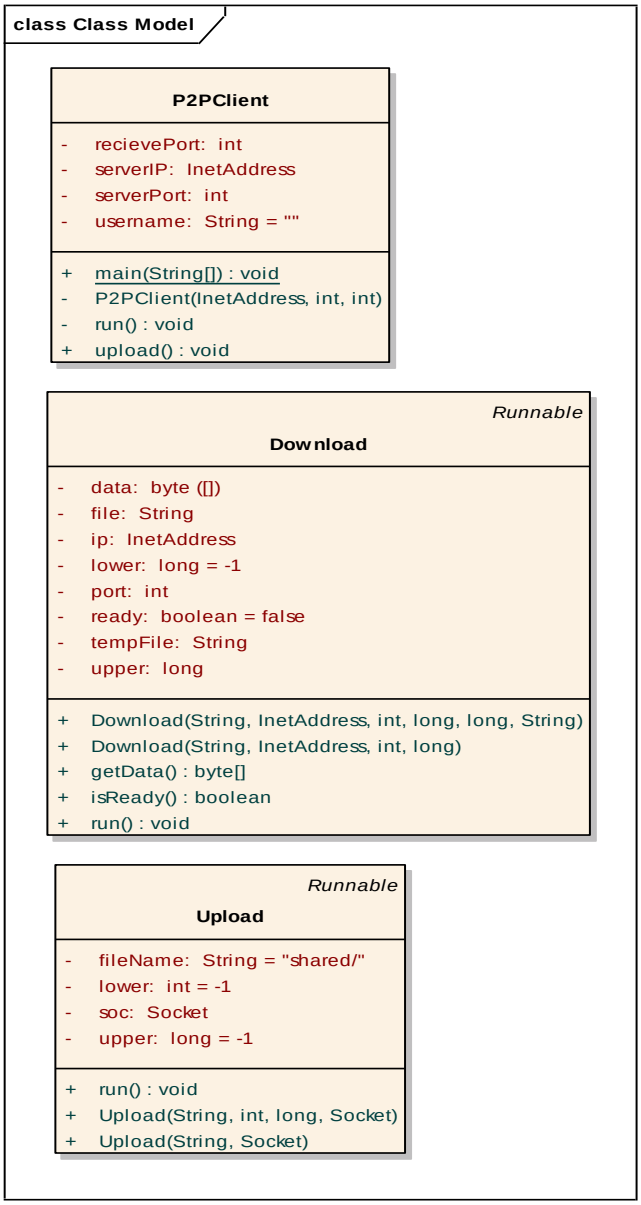


Fig 3 Class Diagram of Server Application

The server application is a batch process and when it runs it opens a server socket and listens for connections. When a client connects to server, server communicate with that client in a separate thread in communicate method.

The client application has a command line menu and when client begin to run, it opens a server socket for answering other peers file request. We have Upload and Download classes for running these operations in thread. For example when an upload request arrives, client created a new Upload instance and transfers a file concurrently with other operations.

These two applications contain the only core logic of the P2P file sharing and at the same starting point of our project. The main concern this system is concurrency: downloading uploading and doing other jobs at the same time. There are many other concerns in today's P2P applications that are not handled here. In this paper we extend these applications to cover more concerns like limiting concurrent transfers, priority based download, user friendly user interface.

2.2 Applying Design Patterns

While providing OO solution to our system, some problems have encountered are the common problems that have solutions. In such cases we make use of applying patterns. For updating UI components and informing server about change in shared folder, we provide a mechanism that monitors specified folders and inform observer classes about changes. This problem is suitable to solve with Observer Pattern. Also for providing concurrent transfer limit, we apply Active Object pattern. To make some operations to run in separate thread we have already used Worker Object pattern. These patterns and how applied will be discussed in this section.

Observer Pattern

The observer pattern (publish/subscribe) is a software design pattern in which an object maintains a list of its dependent objects and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems [5].

The participants of observer pattern are subject and observer.

Subject

This is basically an interface that enables observers to attach and detach themselves to class that implements this interface. This interface holds a list of observers that are subscribed to it. The main functions for desired functionality are as follows:

- Attach - Adds a new observer to the list of observers observing the subject.
- Detach - Removes an existing observer from the list of observers observing the subject
- Notify - Notifies each observer by calling the update function in the observer, when a change occurs.

ConcreteSubject

This class is the actual implementation for the subject. Concrete observers are subscribed to that subject. When an event occurs, the necessary observers are called.

Observer

This class defines an updating interface for all observers, to receive update notification from the subject. The Observer class is used as an abstract class to implement concrete observers.

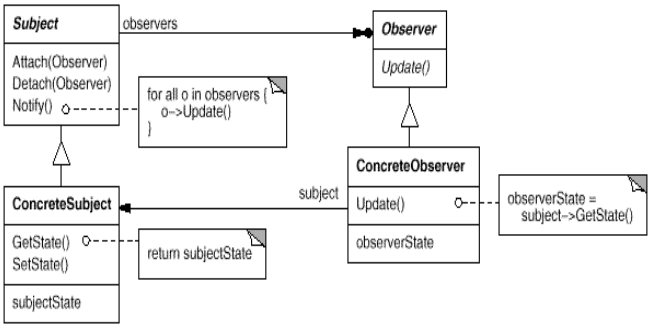


Fig 4 Structure of Observer Pattern

ConcreteObserver

This class is the actual implementation for the observers which are subscribed to ConcreteSubject. When an event occurs, the corresponding method of observers is called.

With the help of Observer pattern, subjects and observers can change independently. Any of them can be reused without having dependent to other parts. For instance, without modifying the subject, the business logic in the observers can be changed. In short, with Observer Pattern, Abstract coupling is achieved. That is subject is not aware of the concrete implementation of concrete observer and vice versa.

The downloading and uploading processes are two major parts of a P2P application since they should be in communication almost every other part of the application such as the download and upload manager, user interface, etc. The files which are downloaded and ready to be uploaded can be moved, deleted, and edited without checking the actions performed on them by the P2P system. That is why; some kind of monitoring system should be responsible for the file structure of interest and P2P system.

Active Object Pattern

Active Object pattern is object behavioral pattern, which decouples method execution from method invocation in order to simplify synchronized access to and object that resides in its own thread of control. The Active Object pattern allows one or more independent threads of execution to interleave their access to data modeled as a single object. A broad class of producer/consumer and reader/writer applications is well suited to this model of concurrency. This pattern is commonly used in distributed systems requiring multi-threaded servers. In addition, client applications, such as windowing systems and network browsers, employ active objects to simplify concurrent, asynchronous network operations [1].

The Structure of pattern is shown in Fig 3. Clients, who aim to call the methods of Servant class, call them through Proxy class. Only the Proxy class is visible to clients. When client calls a method on Proxy, Proxy creates corresponding MethodRequest object and enqueues it on Scheduler's queue. Scheduler contains an ActivatinQueue, which contains the MethodRequest. Scheduler also contains Dispatcher that dequeues MethodRequests that are ready to run. MethodRequest call method is responsible for calling the Servant's original method. When MessageRequest dequeued its call method is invoked.

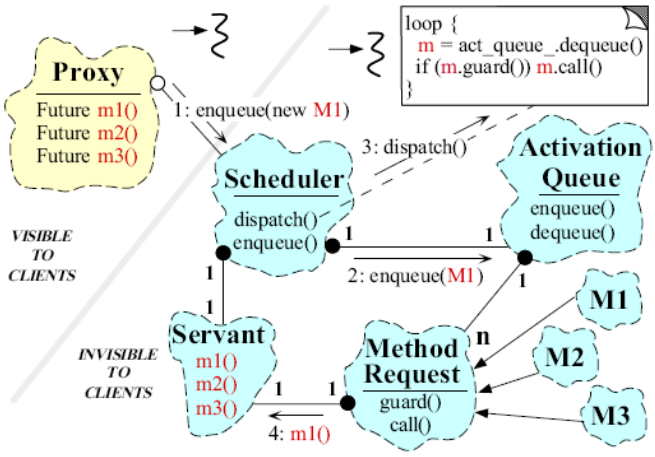


Fig 5 Structure of Active Object Pattern [2]

For decoupling the method invocation and execution to enhance the concurrency, we apply Active Object [1] pattern. Class diagram of the pattern is given in Fig 7. When we map our classes to Active Object pattern, our Servant and Client classes are same which is Client in our solution. Our MethodRequest are uploading and download because these methods have execution precedence according to priority and there is a limit for concurrent download and upload operations. So we take extra parameter in Proxy for upload and download which is the priority and also we add limit as a parameter in Scheduler's constructor. There are three phases in the pattern:

1. **Method Request construction and scheduling:** In this phase, the client invokes a method on the ServantProxy where this might upload and download in our solution. This triggers the creation of corresponding IMethodRequest which maintains the argument bindings to the method, as well as any other bindings required to execute the method. The ServantProxy then passes the IMethodRequest to the Scheduler, which enqueues it on the ActivationQueue. Here the ActivationQueue implemented as PriorityQueue based on the priority of the download and upload operation.

2. **Method execution:** In this phase, the Scheduler runs continuously in a different thread than its clients. Within this thread, the Scheduler monitors the ActivationQueue and determines which IMethodRequest(s) have become runnable, e.g., when their synchronization constraints are met. When a MethodRequest becomes runnable, the Scheduler dequeues it, binds it to the Client, and dispatches the appropriate method on the Client. When this method is called, it can access/update the state of its Client and create its result(s).
3. **Completion:** In the final phase, Scheduler continues to monitor the Activation Queue for runnable IMethodRequests. Since our methods don't return any result this phase is skipped in our solution

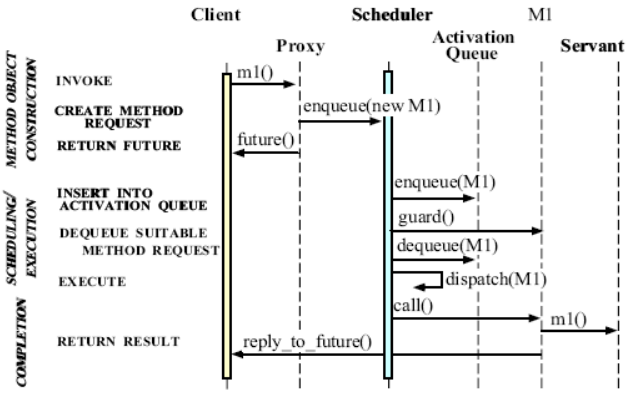


Fig 6 Dynamics of Active Object Pattern [2]

In Fig 6 the dynamics of original pattern is given, different from the original our Proxy does not return Future and in Completion phase our MethodRequest does not reply to Future.

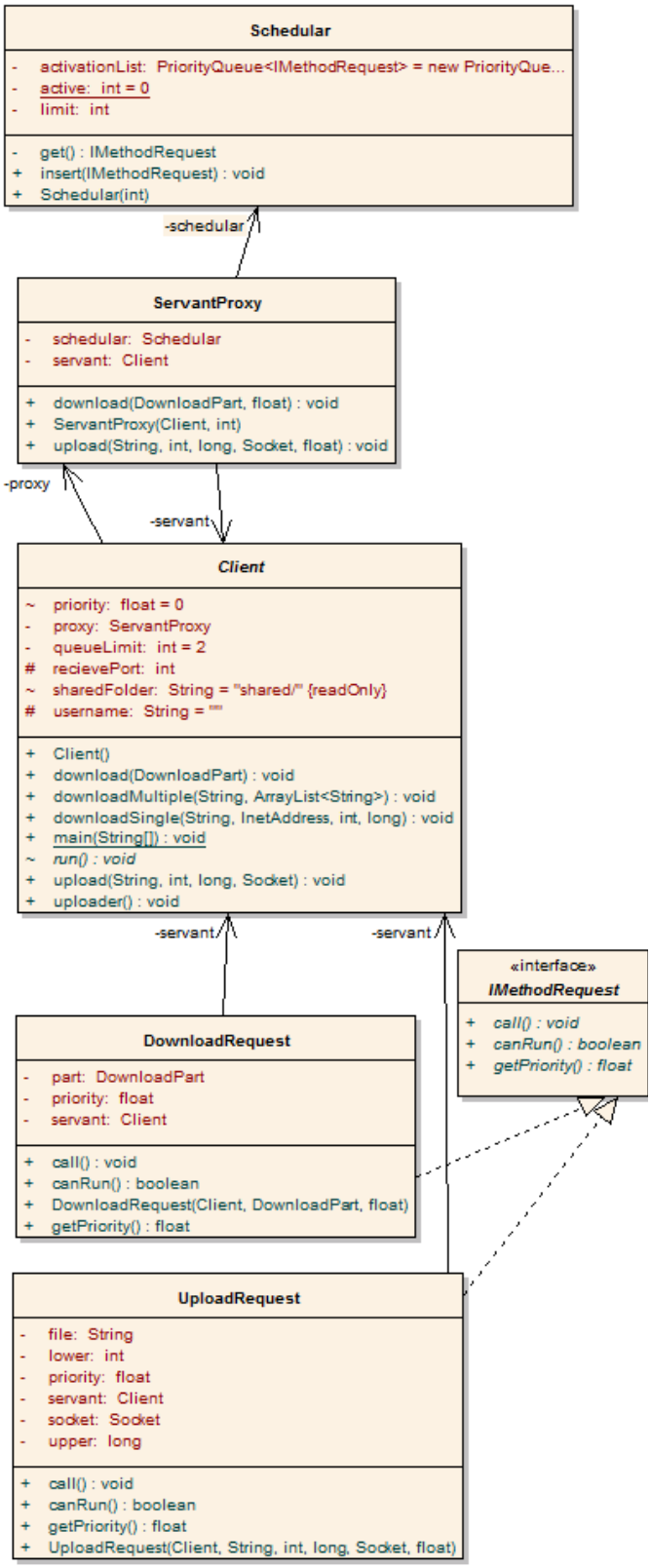


Fig 7 Class diagram of applied Active Object pattern

3. Identifying crosscutting concerns

OOSD is good way separating concerns but some concerns which have crosscutting behavior, cannot handled with OO. These concerns are creating worker objects, observer, monitoring and fault management. In this section we give the details of these concerns and why there are crosscutting.

Observing Object State Changes

The interactions between file system and P2P system are scattered among all parts (modules) of the application. Observer Pattern is the obvious choice for the problem of interaction between modules. It enable the file system watcher to broadcast event happened on the file structure. For instance, when a file is deleted, the folder structure watcher broadcast the modules that are interested with the deletion of a file such as user interface updates the table tree structure to show that the file is deleted and download/upload manager restructure its download/upload schema according to deleted source.

Another issue is to reverse communication as when the user interface wants to change the structure of the file system, and then the same procedure can be called.

Creation of Worker Objects

Since one of the main concern in P2P applications is concurrency, we need a lot of Worker Objects for operations. Some of these operations are upload, download, upload request handler, file merger which are long running operations. Choosing either defining a Worker class for each operation or creating a anonymous threads inside methods, are crosscutting.

```
public class Download extends Runnable
{
  public Download(..)
  {
        ..
  }
  public void run()
  {
        // Core aspect
  }
}
```

If we define a class for Worker Object, the concerns concurrency and download, the core aspect, is tangled.

```
public void download(..)
{
  Thread t = new Thread()
  {
        public void run()
        {
              // Core aspect
        }
  };
  t.start();
}
```

Also creating anonymous threads inside all asynchronous methods tangled with core aspect and scattered among all classes that have such operations.

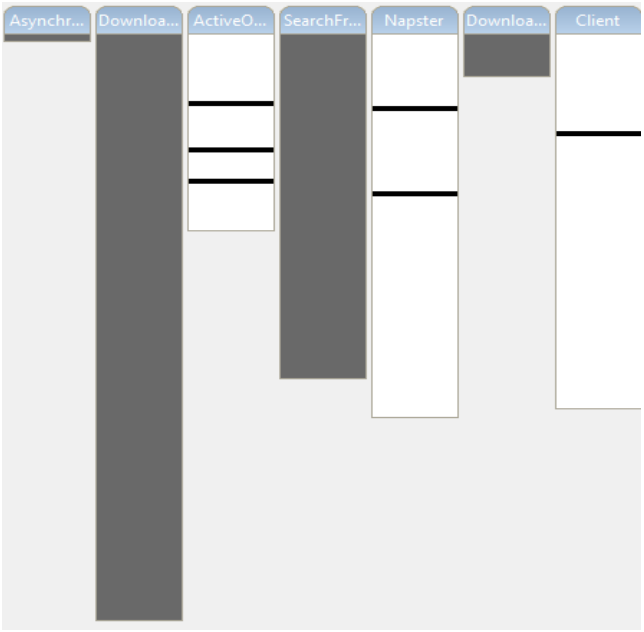


Fig 8 Scattered Creation of Worker Object Concern

As shown in Fig 8 the concurrency aspect is scattered among three classes.

Monitoring

In our concurrent system, some actions trigger some state changes objects. For example when file download completed, GUI must change status of corresponding Download object to complete. Such cases require method calls for informing about the status which is crosscuts with core aspect.

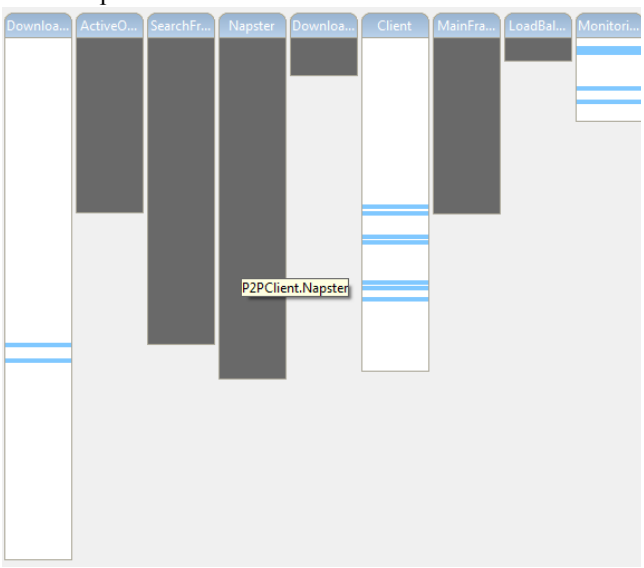


Fig 9 Scattered Monitoring Concern

As shown in the Fig 9 the monitoring concern is scattered into three classes.

4. aspect oriented implementation

Since the one the critical point software development is separating the concerns, we should separate crosscutting concerns stated in previous section. We provide our solution with using AspectJ.

Observing Object State Changes

The main purpose of the Observer Pattern is to construct a one-to-many dependency between objects so that when one object (which is the subject of the pattern) changes state, all its dependents (subscribers) are notified and updated automatically. The important questions related to Observer pattern are as following:

- Which objects are Subjects and which are Observers?
- When should the Subject send a notification to its Observers?
- What should the Observer do when notified?
- When should the observing relationship begin and end?

[4]

Before inspection our solution in terms of these questions, the motivation for using aspect oriented observer pattern is discussed below. The general object oriented implementation of observer pattern requires changes to the domain classes which is not trivial for most of the cases. For the subject class, many methods should be added (addObserver, removeObserver, notifyObservers). Also, notifyObservers method call should be added to all necessary methods to trigger the events. The observer classes should also implement update method which will be called by the subject. To start the relationship between the subject and the observer, some glue code should be implemented. As a result, it can be said that Observer pattern is not a lightweight pattern.

The peer-to-peer (p2p) system had already contains some basic functionality when we started the project. That is why; implementing observer pattern requires writing some parts of the system from scratch and for some other parts requires code modifications. As this is not a desired implementation methodology, we searched through literature and decide to apply an aspect oriented version of observer pattern. As a starting point, we use the abstract aspects of Jan Hanneman and Gregor Kiczales who prepare the 23 GoF patterns as aspect oriented. Their implementation provides an abstract aspect. Instead of the interface mechanism (subject and observer interface which contains the methods required), this approach requires these interfaces without any method in them. The aims of these interfaces are to mark the classes according to their roles. These interfaces are empty and don't require any method.

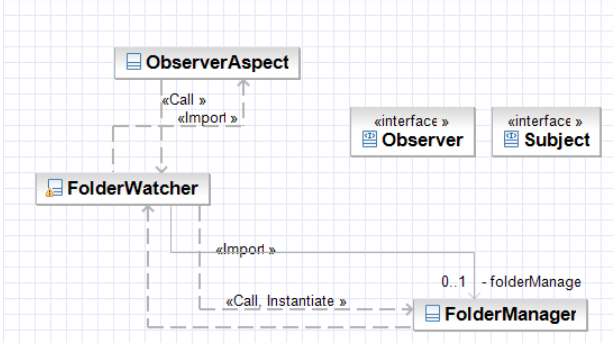


Fig 10 Observer Aspect and FolderWatcher Class Diagram

Another major difference is the tracking of observers. Normally, the participants keeps track of the observers

that listening to a particular subject. However, instead of this distributed approach, in aspect oriented approach, this job is centralized via ObserverProtocol aspect. This aspect will keep track of subjects with their observers (listeners). This is done via a HashMap structure. The key values are the subject objects and the values in the HashSet are LinkedLists that holds the observers that listen that particular subject.

```
  / * Stores the mapping between Subjects and
   * Observers. For each Subject, a LinkedList
   * is of its Observers is stored.
   */
  private WeakHashMap perSubjectObservers;

  /**
   * Returns a Collection of the Observers of
   * a particular subject. Used internally.
   */
 protected List getObservers(Subject subject)
{
    if (perSubjectObservers == null)
   {
        perSubjectObservers = new WeakHashMap();
    }
    List                observers                =
(List)perSubjectObservers.get(subject);
    if ( observers == null )
   {
       observers = new LinkedList();
       perSubjectObservers.put(subject,
observers);
    }
     return observers;
  }

/**
* Adds an Observer to a Subject.
*/
public    void    addObserver(Subject    subject,
Observer observer)
{
    getObservers(subject).add(observer);
}

/**
* Removes an observer from a Subject.
*/
public    void    removeObserver(Subject    subject,
Observer observer)
{
    getObservers(subject).remove(observer);
}
```

//aspect continues...

Another main difference is the notifying the observers. The OO way of doing is via subject. Subject calls every observers update method. On the other side, aspect oriented version is also using a loop to call observers update methods, but this time from the aspect.

```
protected abstract pointcut subjectChange(Subject s);
after(Subject          subject)          returning          :
subjectChange(subject)
{
          for (Observer observer : getObservers(subject))
          {
            updateObserver(subject, observer);
          }
}
```

```
protected    abstract    void    updateObserver(Subject
subject, Observer observer);
```

The updateObserver() method is called when each Observer which changed state. This is implemented in FolderWatchler class which extends ObserverProtocol.

```
public    void    updateObserver(Subject    s,    Observer
o)
{
        MainFrame service = (MainFrame) o;
        service.incomingFileChange((FolderWatcher
        )s);
}
```

The role assignments are also handled via aspect. The important point here is that, all these role assignments are handled without any interference wiht modifying classes directly.

```
declare parents : FolderWatcher extends Subject;
declare parents : MainFrame implements Observer;
```

Starting the observation relationship

The last step of the pattern is to make the relationship between subjects and observers. This is also done via aspect.

```
// used for injection
private      MainFrame      defaultObserver      =
MainFrame.newContentPane;

pointcut folderWatcherCreation(Subject s) :
        execution(public FolderWatcher+.new(..))
        && this(s);

after(Subject        s)        returning        :
folderWatcherCreation(s)
{
        addObserver(s, defaultObserver);
}
```

Finally, concerning the aspect oriented observer pattern the last point is the answer to the questions mentioned above. The answer to the "Which objects are Subjects and which are Observers" is that subject is mainly the FolderWatcher class that we implemented that implements the empty Observer interface. FolderWatcher has basically maintains a thread which in charge of monitoring a given path whether there are any change in the folder structure.

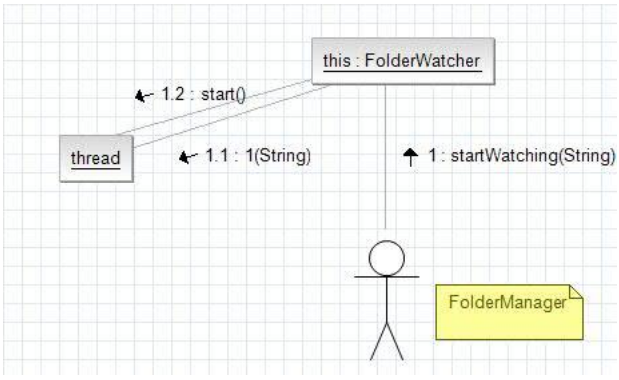


Fig 11 FolderManager thread structure

The observers are so far the user interface and to some

extent, indirectly download and upload manager. Second question is “When should the Subject send a notification to its Observers? “. This is also related to FolderWatcher implementation. Any file adding, deletion or change in the size of file produce notifications so that, for instance, when a file is being downloaded, whenever a new package is arrived, a new notification is broadcast and user interface update its view accordingly. Third question is “What should observers do when notified”. This is a part of the logic which observer has. For instance, for the case of a view in the user interface that lists the downloaded files, when user delete a file outside the system, FolderWatcher broadcasts an event and user interface should update its view via catching that event. And final question is “When should the observing relationship begin and end”. This is also a very visible through a user interface example. Say a view is disposed and no longer needed, and then the corresponding observer in the subject should be eliminated.

Creation of Worker Objects

To handle the creation of Worker Objects we define an annotation to determine which methods should be run asynchronously. Annotation does not have any attributes since only the aim is tagging the methods.

```
public @interface Asynchronous {}
```

The signatures of the asychronous methods and their usage are given below:

```
@Asynchronous public void download(DownloadPart
part): Download part of a file.

@Asynchronous public void upload(String
fileName, int lower, long upper, Socket soc):
Uploads part of a file.

@Asynchronous  private void dispatch(): Get the
next avaliable transfer request from activation
queue and calls in.

@Asynchronous public void uploader(): Handles
the upload request coming from peers.

@Asynchronous  private void merge(int numConn,
String file,ArrayList<DownloadPart> downloaders,
long fileSize): Merge the download parts.
```

A generic solution for creating Worker Object is provided with an abstract AsynchronousExecutionAspect. Aspect has a abstract pointcut which captures the asychronous methods calls. Around advice executes the method inside an anaymous Worker Object. Since In our system there is no WorkerObject returning a value, the advice returns null. It is also possible to handle the methods that return a value by defining a Worker class with a return_value attribute and getter method for that. By using the custom Worker class, the methods that return value can also be executed asynchonously.

```
public abstract aspect
AsynchronousExecutionAspect {
        abstract pointcut asyncOperations();

        Object around():asyncOperations(){
                Runnable worker = new Runnable()
{
                        public void run() {
                                proceed();
                        }
                };

                new Thread(worker).start();
                return null;
        }
}

public aspect Concurrency extends
AsynchronousExecutionAspect{
        pointcut asyncOperations():
        call(@Asynchronous * *(..));
}
```

To use the abstract AsychronousExecutionAspect we define a concrete aspect Concurrency inherits from AsychronousExecutionAspect. asyncOperations pointcut captures all the method calls with any number of parameters and return type.

Monitoring

Monitoring aspect captures all changes in objects states and takes an action according to the case. In this aspect for two cases which are monitoring the how much data download for a specific download to update the download status panel.

```
aspect Monitoring {

        long Client.totalUpload = -1;
        long Client.totalDownload = 1;


        after(Client c,byte[] data, int off, int
len): this(c)&&call(void
java.io.DataOutputStream.write(byte[],int,int))
&& args(data,off,len){
                c.totalUpload += len;

        }


        after(Client p) returning(int retval) :
this(p) && call(int
java.io.DataInputStream.read(..)) {

                p.totalDownload += retval;
        }


        after(Client p):target(p) &&(set(long
Client.totalUpload) || set(long
Client.totalDownload)){
                if (p.totalDownload> 0) {
                        p.priority= p.totalUpload/
p.totalDownload;
                }
        }

        after(Download d) : target(d) && set(long
Download.downloaded)
        {
                d.downloadFlagChanged = true;
        }

}
```

We introduce to attributes into Client class since how much data downloaded and uploaded is not the concern of client. totalUpload incremented with amount of data written on socket, where we can gather this information from the arguments of DataOutSream.write method. After advice is used since if a problem occurs while writing data, totalUpload should not change. A similar advice is

used for updating totalDownload but in this advice the return value gives us how much data is read from the socket.

For determining the priority we monitor the set operations of totalUpload and totalDownload which captures the pointcuts inside the aspect since the totalUpload and totalDownload is introduced in the aspect. Priority is the proportion of totalUpload to totalDownload.

For determining the download completed we monitor set operation of downloaded attribute and we set the downloadedFlag of Download object.

5. Discussion

In this paper we mention utilizing design patterns and aspect oriented development principles for P2P file sharing application. Instead of beginning form the scratch, we modify an existing system. Most of the problems stated in Section III were not covered in the previous system. By using the AOD principles we easily integrate our aspects to system with the minimum modification to the original system. Many of the problems that we handled are similar to existing problems in the literature so we look for the patterns applied for the similar cases. Applying patterns, especially for concurrent systems is not easy, because solutions are generally technology dependent. Implantation provided for Active Object pattern in [2] is designed for C++. At first we tried modify C++ code to get Java but we weren't able to do it. Then we provide another solution with using Proxy class and reflection in Java SDK but when calling our asynchronous methods with reflection we encountered some problems. At last we provide our proxy class implementation and handle the situation.

Since the usages of an already existing project as a starting point, applying certain patterns was hard to implement. Observer pattern is a heavyweight pattern which requires many code recompilations in many point of the code. The subject, the observers and glue code should be implemented into existing code. Instead of this approach, aspect oriented way of implementation of observer pattern is far more easy then the orthodox way. By using aspect to keep track of subject, observers' relation which is a one-to-many relation is no longer have to hard code into existing code. Moreover, aspect is also role assignments such as MainFrame class as observer is done via aspect. Normally, the code inheritance should be changed, but aspect eliminates this need. Another point is aspect catches the update in the subjects and run the desired code again inside aspect which is a perfect solution for an already existing code structure.

The problems stated in Section III are the small portion of the cake. Also access to shared resources, fault management, security and logging can be implemented by using aspect oriented development techniques.

6. Related Work

Peer-to-Peer applications have been popular lately, and therefore many studies have been performed. There are two significant works that we want to talk about in this paper. One is a framework for testing distributed systems designed by Hughes [6], and the other one is about managing concurrent file updates in a P2P system designed by Borowsky [7].

In the first system, paper discusses a Java based testing environment that facilitates the testing of distributed applications with easy publication, monitoring and control of prototype systems on a peer-to-peer test-bed. They state that, manual creation and maintenance of such tests is an extremely time consuming activity, especially where nodes are required to change their behavior dynamically. Such tests may require the creation of specialist control and monitoring tools. Furthermore, monitoring code must often be inserted throughout such prototype systems. The addition and removal of such code is time-consuming and error-prone. Their framework uses a combination of reflection and Aspect Oriented Programming to automatically insert and remove code that is required for applications to interface with the testing framework's central monitoring and control interface [6].

In the second system, they state that they solved the problem of concurrent data updates in a peer-to-peer system by layering a robust, well- tested, client-server application for managing concurrent file access on top of a peer-to-peer distributed hash table. This coupling provides a distributed, scalable, fault-tolerant, service for managing concurrent updates to replicated data. By using well-tested components, they are able to leverage the extensive development efforts put into both CVS and Bamboo to extend the functionality of both services. Moreover, this work gives proof of concept that the client-server and peer-to-peer paradigms, when appropriately layered, can provide functionality beyond the current capabilities of either paradigm alone [7].

7. Conclusion

The internet usage is widespread which leads to Peer-to-Peer (P2P) file sharing systems becoming popular way for sharing files among users on the internet. P2P enables its clients to utilize the infrastructure by downloading and uploading resources whenever available sources are found. Napster is a widely accepted protocol for P2P systems. It is basically composed of a server that holds the users info such as IP addresses and file lists that clients have and clients. Clients connect to server and give the server a list of files it has and in return client ask for other clients which contains a particular file. After the client have the list of other clients, it connects to other clients to get the file. The server doesn't interfere with the file exchanges between clients. The server only responsible for making searches among all users and return a list of clients who has the desired file.

Developing P2P file sharing system includes many complex concerns like concurrency, observing the state change of objects and monitoring status of objects. The architecture of a P2P system, due to its nature, composed of many concerns. Some of these concerns are crosscutting and they are either scattered around many modules of the system or tangling which is one module contains many unrelated concerns.

Object oriented software approach solves many problems easily but it cannot cope with crosscutting

concerns elegantly. Aspect oriented software development (AOSD) aims the separates crosscutting concerns and via that enables modularization and increase in the overall software quality. AOSD in our project act a quality enhancement.

To provide an elegant solution we apply Active Object pattern for decoupling the invocation and execution of methods for better concurrency. Downloading and uploading mechanisms are controlled by an aspect powered mechanism. The number of uploads and download and the threading mechanism is controlled in an aspect oriented fashion. For the methods that should work in separate thread we use Worker Object pattern with aspect oriented implementation. Moreover, the synchronization between file structure and P2P system is done via an observer pattern solution which is enhanced with aspect oriented approach.

Applying patterns and using aspect oriented programming make the system more reliable, easy to understand and maintainable. Also development stage of the application is shorter than traditional ways because of handling crosscutting concerns in separate modules. Another final point is that aspect oriented patterns eliminate the need to change the already existing code.

References

[1] K. G. Morrison, K. Whitehouse, “Top 10 downloads of the past 10 years”, <http://www.cnet.com/1990-11136_1-6257577-1.html>

[2] R. G. Lavender, D. C. Schmidt “Active Object : An Object Behavioral Pattern for Concurrent Programming,” http://www.cs.wustl.edu/~schmidt/PDF/Act-Obj.pdf

[3] R. Laddad, *AspectJ in Action,* Manning:Greenwich, 2003, pp.247-256

[4] N. Lesiechi, “Enhance Desing Patterns”, <http://www.ibm.com/developerworks/java/library/j-aopwork6/index.html#sidebar3>

[5] Wikipedia contributors, 'Observer pattern', Wikipedia, The Free Encyclopedia, 10 December 2008, 15:06 UTC, <http://en.wikipedia.org/w/index.php?title=Observer_pattern&oldid=257061737>

[6] D. Hughes, P. Greenwood, G. Coulson, *Framefork for Testing Distributed Systems*

[7] E. Borowsky, A. Logan, R. Signorile, *Leveraging the Client-Server Model in P2P: Managing Concurrent File Updates in a P2P System,* Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW 2006)

Aspect-Oriented Development of an E-Commerce Application

Doğan Altunbay, Damla Arifoglu, Hilal Zitouni

Department of Computer Engineering, Bilkent University

Bilkent, 06800, Ankara, Turkey

(altunbay, arifoglu,zitouni)@cs.bilkent.edu.tr

Abstract

E-commerce applications which are used for selling and buying products on electronic platforms have become popular with the recent growth in internet usage. Different kinds of applications for e-commerce applications are developed considering the concerns of shipment, financing persistency and ordering systems. However; modularization of these concerns are not usually easy with object-oriented software development. AOSD is a recently developed new software development technology which provides abstraction mechanism for these so-called crosscutting concerns. In this paper, we will be discussing an aspect-oriented software development of an e-commerce bookstore application and we will briefly give the lessons learned throughout the development phase.

Keywords
E-commerce systems, Aspect-oriented Programming, Exceptions, User Authentication, , Design Patterns,

1. INTRODUCTION

Separation of concerns is one of the most important requirements of good system design and implementation, since it leads to maintainability, extendibility, reusability, adaptability and clarity which can all be achieved by the help of Aspect-oriented Systems. AOP (Aspect-oriented Programming) deals with crosscutting concerns in a modular way and this helps modularity, easier developing of the code and increase of reusability.

E-commerce systems are online systems, which sell products or provide services via internet and having an e-commerce website will increase the benefit of the company since web based communication and shipment will be easier for the clients from the other sides of the world. Some of the examples for e-commerce applications are [4, 8, 9] In this paper, we use a case study named "e-commerce bookstore system" in aspect-oriented development in order to show the benefits of aspect-oriented design over object oriented design. Our bookstore application is similar to e-bay which is a worldwide famous e-commerce internet web site.

Our system, which is an e-commerce book store application, originally provides users to search, and order his selected choice of the documents that are retrieved from the database. The paper will more focus on the implementation of the application itself rather than the server – client part since the server – client part of the application is not the basic focus of how usage of aspect-oriented development has improved our previous, original e-commerce bookstore application.

The previous system was actually did not include the authentication for different user types. In other words, all the users who enter the site may search through the document database; however the non-registered users should not be able to order any selected document. On the contrary, the system was not checking whether or not to allow users to give orders. The aspect-oriented development on the other hand will handle to improve the

previous system by providing the users different types of authentications.

Applying the aspect-oriented development on the previously implemented code for improvement has proved its superiority over changing the system manually from the code inside.
The organization of the paper is as follows: in section 1, a brief information on the e-commerce application concerns will be given, then in section 3 our e-commerce case study and its detailed design will be explained in object oriented design, and then the improvements provided via aspect-oriented design will be mentioned. This will be followed by the aspects we have implemented under section 4, *Aspect-oriented Programming*. Afterwards, in section 5 the related works achieved up to now will be given, and finally we will give the discussion and conclusion in section 6 and.

2. OBJECT-ORIENTED DESIGN

In this section an example of the design of e-commerce application system will be introduced. In the subsequent sections the application will be improved with the help of aspect-oriented development.

An e-commerce application is used for buying and selling products over electronic systems like internet. Our book store application is an e-commerce application in which people can buy and search documents from a database of documents over internet. In our application we have not implemented the server client part since it is not really the focus of applying aspect-oriented development on a system. Therefore, we will give brief information on how the bookstore application is implemented without the internet part and the UML modelling of our system.

Here we will introduce a simple book store application in which the users will search through the document database of the Book Store, in order to view and/or order documents. In this system the database has three types of documents which are Book, Magazine and Journal, each of which has soft and hard copy versions.

The program is implemented by using some of the design patterns to overcome the problems that might occur. In order to explain the system better, brief information of the design is given along with the corresponding UML diagrams and screenshots.

Once the user is logged in, the main user interface of the application which is given in Figure 1 will be seen.

At the top of the screen user will search through the database filling specific fields and pressing "Search" button, then the corresponding document list will appear on the list. Once the documents are listed user may either select a document and view its information pressing the "View" button, or select several documents and add them to the list below using the "Add To List" button, which will later be the list of orders once the "Order" button is pressed. Pressing the "Order" button another window specifying the prices, quantity and the total price of the listed orders will come up and will offer user to enter his address and his country for shipment information.

2.1. Design Patterns Used and UML Diagrams

2.1.1. Overall Package System

Before moving to design patterns, below there is the UML diagram for overall packages of the system.

The system has four different packages, which are ***viewer***, ***item***, ***store*** and ***user,*** in its implementation. The ***viewer*** package holds the classes for viewing operation for each type of document that is selected from the list; ***order*** package holds the classes about the ordering and shipment. The user package has user related classes, and ***store*** is a helper for the

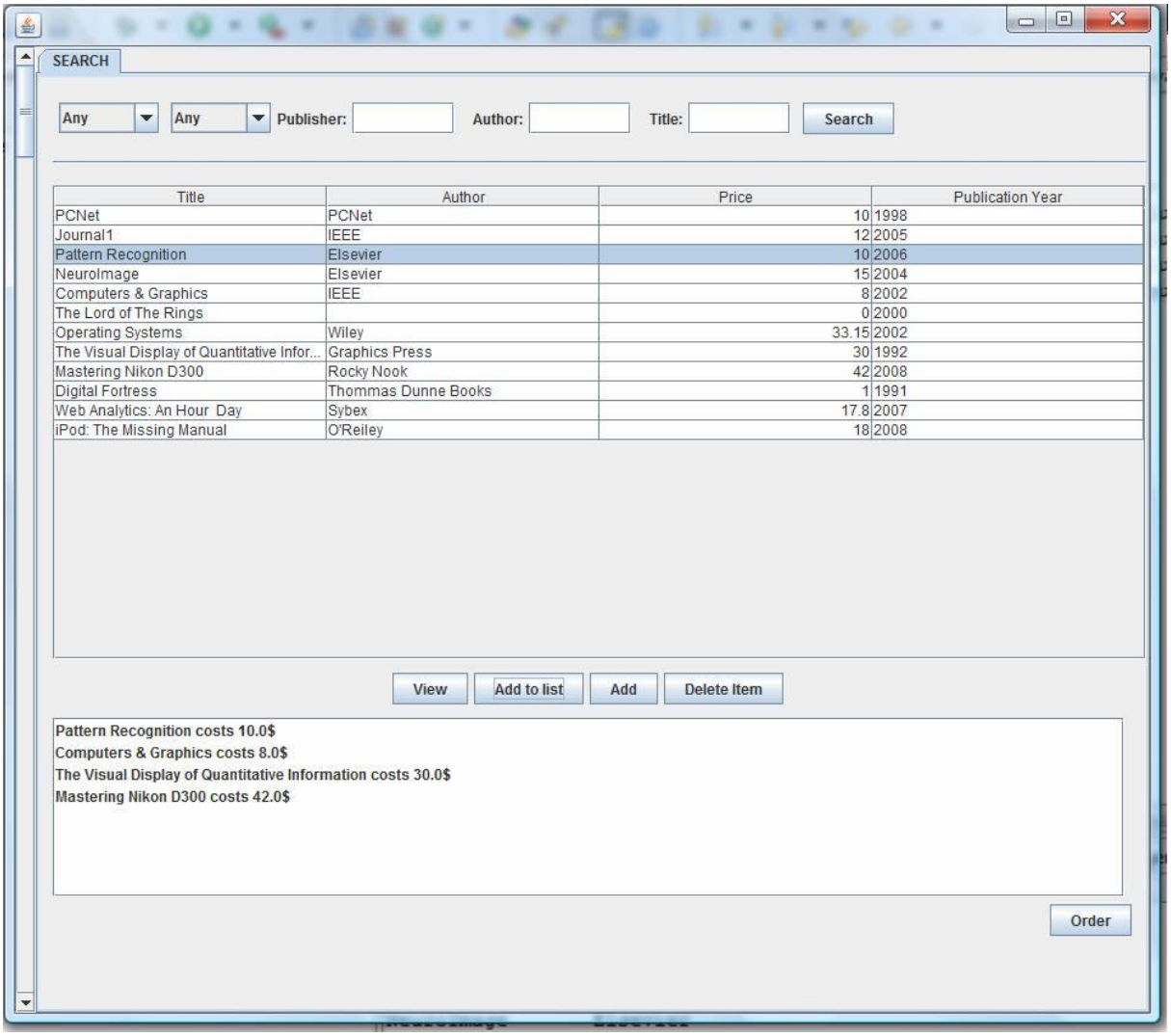
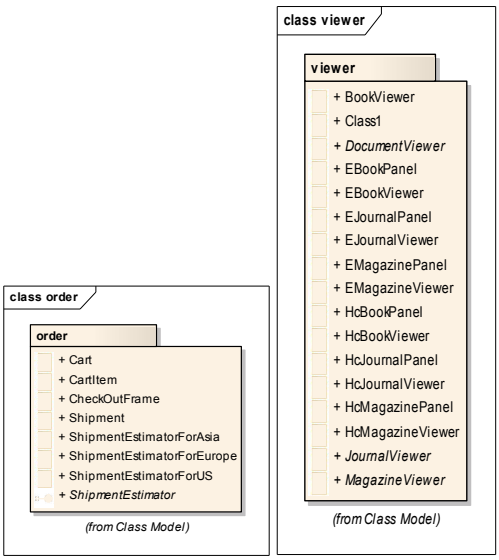


Figure 1 : The main user interface of the application after user logged in

viewing operation, finally, the package ***item*** has the book store item types.

In Figure 2 the user package has different types of users which are implemented after aspect development for improvement.



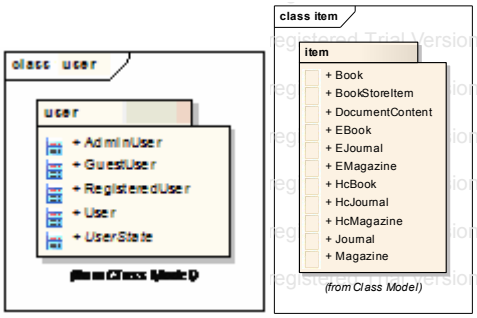


Figure 2 : Order Package and Viewer Package, User Package, Item Package

2.1.2. Design Patterns

Three different design patterns are implemented for the BookStore Application; Abstract Factory Method, Strategy Method and State Method. The first two design patterns are developed in the original BookStore Application, whereas the last one, State Method, is added after the production aspects are added to improve the system.

2.1.2.1. Abstract Factory Design Pattern

The Abstract Factory is a creational design pattern that is used for grouping the object factories of a common theme. The factory will decide the concrete object type of its abstract object to be created. In this application, this design pattern is implemented for the view operation of documents. In other words, we will have a DocumentViewerFactory which will select among the soft and hardcopy versions of document types, Book, Magazine and Journal. The abstract class DocumentViewerFactory, will decide on which of the six different types of documents will be displayed on the common panel of the dialog box that comes up when the "View" button is pressed for the corresponding selected document item, from the table. The UML diagrams for the viewer and store packages are given in Figure 3

2.1.2.2. Strategy Method Design Pattern

The Strategy Design Pattern is a behavioural design pattern that is used for the differentiation between the different implementation of same operations. In other words, same operations differ for different types of objects in implementation, and strategy pattern handles this by implementing the different implementations of same operations in subclasses. Here, we had our *shipment* procedures using strategy design pattern. For shipment we have a Shipment class which uses a ShipmentEstimator as an abstract class which has subclasses of different shipment estimator classes for different countries. The UML diagram for the corresponding system is given in Figure 4.

2.1.2.3. State Method Design Pattern

The State Design Pattern is a behavioural design pattern which is used for changing the state of an object at run time. Here, the state design pattern is used for changing the state of the user one of the three states; Guest, Registered and Admin User. To give an example of a state change; we can give the scenario of a guest user who tries to give an order. When a guest user presses the *Order*

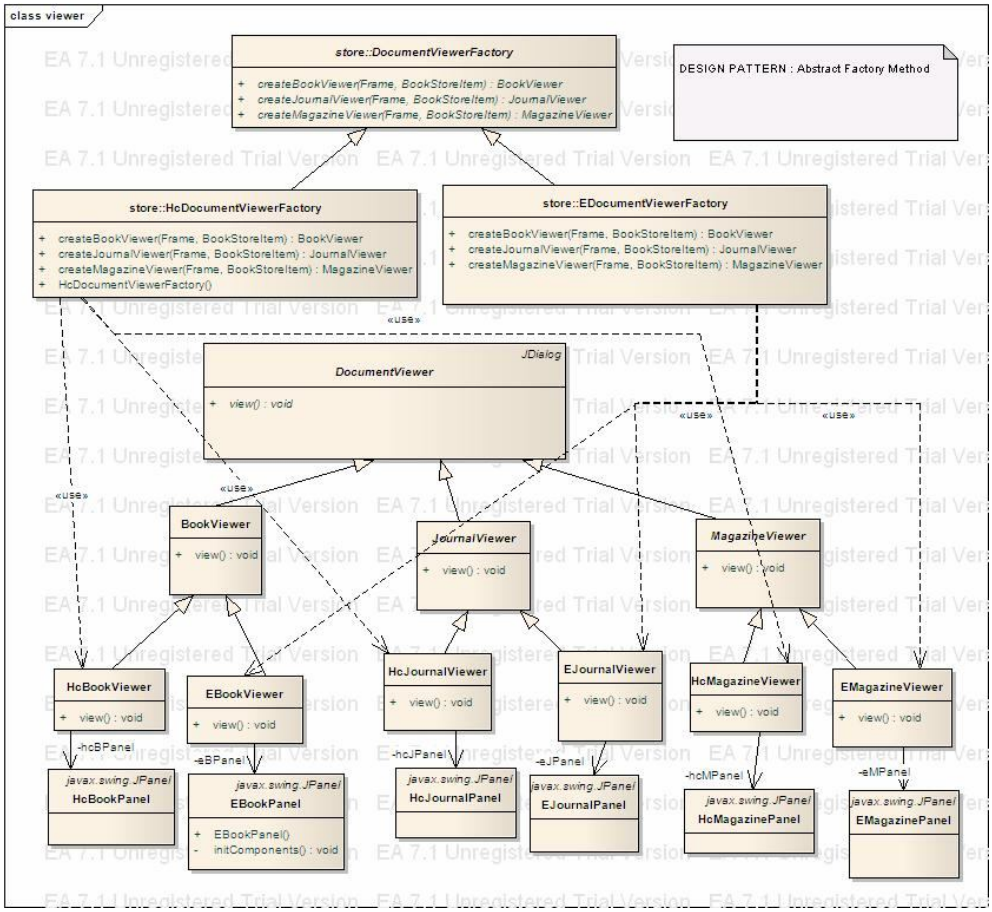


Figure 3 - Abstract Factory Design Pattern

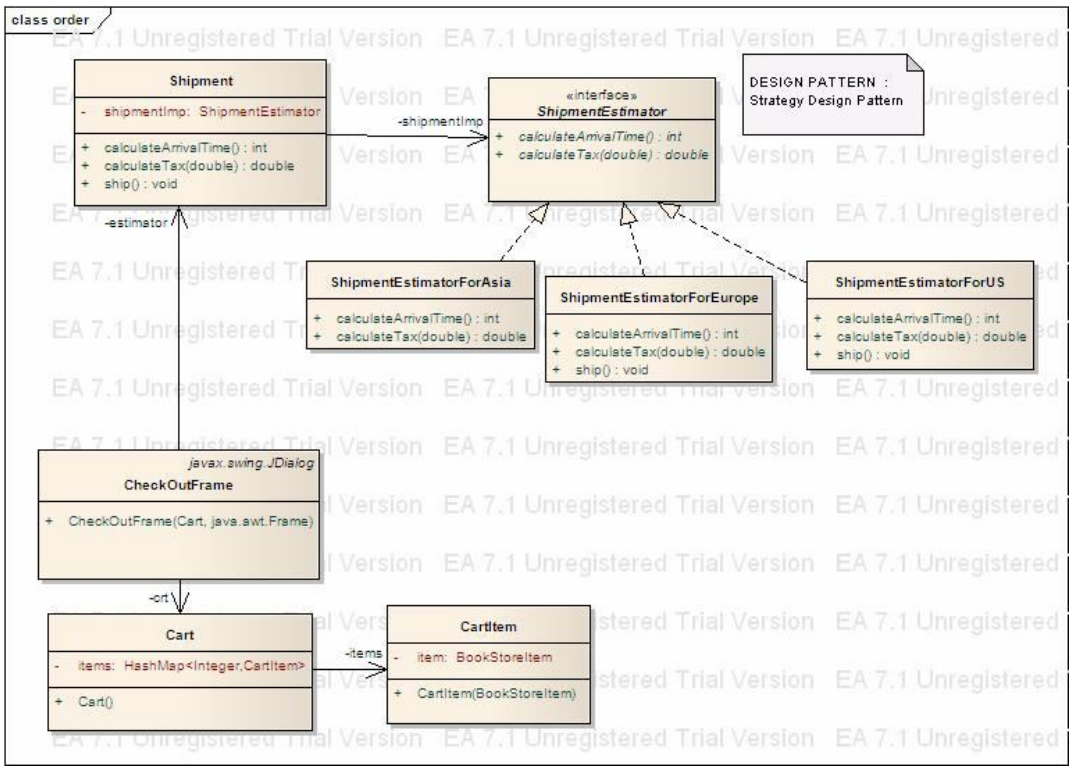


Figure 4 : Strategy Design Pattern for Shipment

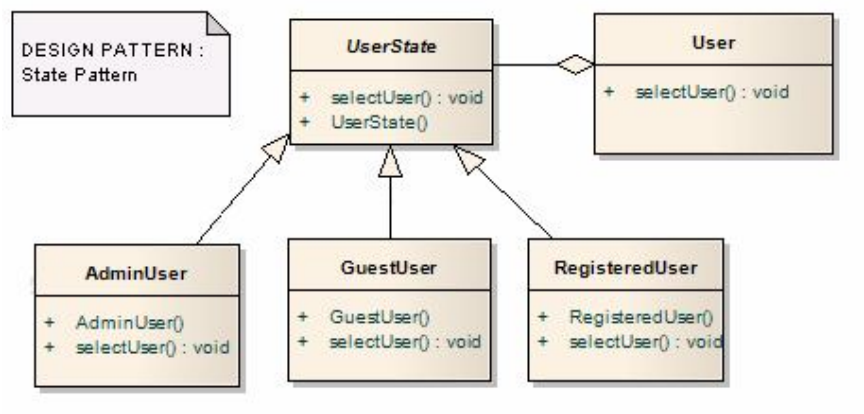


Figure 5 – State Design Pattern for User State

Button a registration user interface will come up and the filling the form he can change his state from *guest* to *registered*, and become a registered user. After changing his state he may give order as he wants from the main user interface. The corresponding UML diagram is given in Figure 5.

3. CROSSCUTTING CONCERNS

E-Commerce Systems are electronic applications that provide users to sell and buy products throughout the world especially via internet [10]. In an e-commerce application the shipment, financial persistency and security of the personal information of the users are some of the vital concerns.

In our e-commerce bookstore application, we noticed that improving the system will result in some of the concerns to be scattered over multiple classes and crosscutting of the basic functionality of the code in these classes. Object oriented programming cannot handle these crosscutting concerns to such an extent that Aspect-oriented Software development can. Therefore, we have especially focused on the concerns of the e-commerce applications. The concerns of our e-commerce book store application can be listed as follows:

- Authorization and Authentication
- Null Checking
- Adding new functionalities to the system as the applications grows with the need of the customers.

Let us give wider information on these concerns.

3.1. Authentication and Authorization Concern

User authentication and authorization concerns are related to determining who is allowed to perform which operations. We need to ensure that users are authenticated to do certain operations. Therefore, authentication is the process of determining whether someone is who he declares to be.

Unauthenticated users have accesses to web pages which do not have any authentication policies. For the authors of [2] authentication is vitally important, since a web application can be a piece of a cake for an intruder, and needs to be protected.

In our e-commerce bookstore system, we used development aspects which will be explained in section 5 to handle authentication concerns. Authorization, on the other hand, is the process of deciding on the permissions of a user over the system. For example in our system there are three types of users, Admin User, Guest

User and Registered User, each of which have different authorizations. A Guest User cannot do anything but the search operation. A Registered User can search, order and view the documents in the database. Admin User on the other hand has all three accesses on the system, plus he can add/delete items from/to system. Our aspects for authorization handle deciding on the accesses to be provided to a user.

3.2. Null Check

In our e-commerce bookstore application system, we use a database to hold the book items (such as magazines, journals, books and their online versions). Null checking will control the problems that might occur about the database.

3.3. Adding New Functionalities

Since e-commerce applications give services to people from all over the world by being an online application, user needs and new demands will come up with the increasing amount in the usage of the e-commerce applications. Therefore, new functionalities and new objects may be needed to be added to the system. In our e-commerce application, at first, every user has the same authentication regardless of whether they should be a guest user, an admin user or a registered user and any user could reach the same user interface. However, with aspect-oriented development, in order to provide different authentication rights to the users, new types of users; such as Admin Users, Registered Users and Guest Users, are introduced to the system.

4. ASPECT-ORIENTED PROGRAMMING

In the previous section we defined our concerns that are scattered over multiple classes. Scattering results in low cohesion and high coupling which is an important weakness for software development. Aspect-oriented programming is a useful method for solving these problems.

With the help of AOP, developers can produce far less complex software systems. AOP not only provides effective mechanisms at the development stage, but also provides easiness in maintaining and upgrading the system. Below we discuss our solution to the crosscutting concerns described in section 2.

4.1. Handling Aspects

In our work, we take the advantage of JBoss AOP for aspect-oriented programming. JBoss AOP is an AOP framework which provides pure java syntax to programmers. In JBoss AOP, aspect weaving is performed via reflection with the help of xml files that define the bindings between join points and advices.

As defined in the previous sections, our case study involves a number of crosscutting concerns. The following sub sections describe our aspect-oriented solutions which address these problems.

```
1 private void viewDoc(BookStoreItem item){
2       if(!this.user.isAuthenticated())
3       throw new

4       UnsupportedOperationException();
5
6       DocumentViewer viewer = null;
7       .
8       .
9       .
10      viewer=
11  docFactory.createMagazineViewer(this,item);
11
12      viewer.view();
13  }
```

Figure 6 – Example insertion of authentication related code to legacy code

4.1.1. Authentication and Authorization

Authentication and authorization of users have vital importance for e-commerce applications. It is necessary to be able identify users and control their actions while interacting with the system.

Authentication and authorization of users have vital importance for e-commerce applications. It is necessary to be able identify users and control their actions while interacting with the system.

```
1 public class AuthenticationAspect{
2       public Object
3       login(MethodInvocation method)
4       throws Throwable{
5       //before login
6       UI3View frame = (UI3View)
7   method.getTargetObject();
8       new
9 LoginDialog(frame,true).setVisible(tr
10  ue);
11 Object result = method.invokeNext();
12              //after login
13              return result;
14      }
15      public Object
16     checkPermission
17  (MethodInvocation method) throws
18 Throwable{
19      UI3View frame = (UI3View)
20        method.getTargetObject();
21      User usr =
22 ((IUserMixin)frame).getUser();
23      if(usr.hasPermission(
24     method.getMethod().getName()))
25      return method.invokeNext();
26              else
27  JOptionPane.showMessageDialog(
28                      frame,
29       "You have not permission for "+
30         method.getMethod().getName(),
31  "Error",JOptionPane.ERROR_MESSAGE);
32              return null;
33      }}
```

Using Introduced Members

Figure 7 – Authentication Aspect

For our case study, we would prefer the users to be able to browse the documents in the system without any restrictions. However, if a user wants to view a document's soft copy, order an item, add an item or delete an item the system should confirm the identity of the user.

In order to handle these requirements, we have to make modifications on our legacy code. Thus, we have to add a bunch of code into the action handlers of the "View" button, "Order" button, "Add" button, and "Delete" button that prompts user to login to system (see Figure 6). This kind of approach leads multiple code segments related with one single concern. Therefore dealing with and tracking these concerns becomes inefficient.

However, we can solve the scattering problem using AOP. Then, we define an aspect, called "*AuthenticationAspect*" (see Figure 7) that gathers authentication and authorization related procedures. The AuthenticationAspect introduces two methods; "*login*" for authenticating the user, and "*checkPermission*" for authorizing the user.

To bind these advice methods to action handlers mentioned above, we need to specify them in the jboss-aop.xml file (see Figure 8).

This file defines the aspects, advices, and pointcuts for the use of the framework. Aspect weaving is performed via reflection in accordance with the definitions in this file.

Another issue in this solution is that we have to add a private field of class User to the application in order to store authentication and authorization information. AOP paradigm provides introduction mechanism which allows programmers to extend existing classes and add new members to them.

```
1 <aop>
2   <pointcut expr="call(public void

3 UI3View-

3 &gt;display())" name="show"/>
4  <pointcut expr="execution(private void

5 ui.UI3View&gt;

6 viewDoc(item.BookStoreItem))"

7 name="view"/>
8    <aspect

9 class="aspect.AuthenticationAspect"

10 scope="PER_VM"/>
11    <bind pointcut="show">
12        <advice

13aspect="aspect.AuthenticationAspect"
14 name="log in"/>
15    </bind>
16    <bind pointcut="view">
17        <advice
18 spect="aspect.AuthenticationAspect"
19  name="checkPermission"/>
20   </bind>
21   <bind pointcut="execution(public void

22 ui.UI3View-&gt;display())">
23        <advice

24 aspect="aspect.AuthenticationAspect"

25 name="login"/>
26   </bind>
27    <introduction class="ui.UI3View">
28       <mixin>
29   <interfaces>
30 user.IUserMixin</interfaces>
31            <class>user.UserMixin</class>
32            <construction>new

33 user.UserMixin()</construction>
34        </mixin>
35    </introduction>
36 </aop>
```

Figure 8 – Sample jboss-aop.xml file.

In JBoss AOP, introductions are done using mixin mechanism. As the name reflects, mixin mechanism uses a mixed data structure for each member introduction to a class. In order to add a field to a class, we first define an interface that includes the methods which accesses that field. Then we place the implementation of the interface which will be mixed with the target class of introduction in runtime. After that, we need to specify how these two classes will be mixed in jboss-aop.xml (see listing 3). Then we can access the field within several places of the code.

4.1.2. Exception Handling:

Exception Handling is used for handling the problems that might occur during the execution of a program. In object oriented programming, exception handlers are implemented for each and every point that an exception might occur using the try and catch code blocks. Implementing exception handlers with OOD not only complicates coding but also makes the code scattered. Moreover, implementing new exception handlers to the system causes a lot of code blocks to be added, which means huge amount of intervention to the code. Aspect-Oriented Development on the other hand, saves programmers from changing and adding code blocks to such an extent. Rather programmers will easily implement advices for the exception handlers in an aspect, and will use them by binding with the specified pointcuts which are the places for the handler advices to be invoked.

Besides, implementing all the exception handlers in a separate aspect code will increase modularity. Rather than having exception handler codes all over the program, it will be coherently collected in a separate part.

4.1.3. Enforcement Aspects:

In our project we needed to use enforcement aspects for restricting the invocation of some of the methods from certain classes. For example, in our program, in order to view the information of the selected document, user will press the “View” button after selecting a

document from the list. However, it should be under control that no programmers should ever invoke the *getContent()* method for any of the six different panels, other than the *EDocumentViewerFactory* class. This is because the content field information of a document should only be available for softcopy versions of the documents, and since the softcopy versions are viewed through *EDocumentViewerFactory* it should not be allowed to be invoked from somewhere else. In case of such a mistake, AOD will handle preventing those methods to be invoked from some place else. However, in JBoss, contrary to aspectj, implementing enforcements and controlling the invocation during compile time is not handled. Therefore, in order to provide this opportunity of control over method calls, we have implemented advices which will prevent the invocation of those methods during run time.

4.1.4. Null Check

As it is explained in section 3, null check will check on the database problems. In Database Systems it is often possible for a primary field to be null, which will arise problems for the insertion to the data. In order not to cause any problems during insertion operations, null check should be done before execution of an insert statement. This is the case where Null Check concern arises in our application. Before invoking the addItem operation of the DBManager class, a null check is done.

5. RELATED WORK

There are some previous works that compares the aspect-oriented software development and object oriented software development in web applications. One of these studies, is Reina, A. et al [2]. To show the advantages of the aspect-oriented development in web applications, they developed an application using AspectJ and Java Server Pages (JSP) and Cocoon, which is a web framework based on XML to publish web pages. The work in this paper will not be discussed in detail, however the two common concerns that they consider will be explained shortly. These common two concerns are similar to the ones that are in our e-commerce web application. First one is security (authentication) and the second one is exception handling. On the other hand, they also address some different concerns from ours which are Pooling, Caching and Logging concerns. They handle authentication concern by using Cocoon to publish web pages and Cocoon itself already handles authentication concern via a module. In Cocoon, the authentication handler is an object that controls the access to the resources. Each resource can be related to one authentication handler and one authentication handler manages the access rights to all the resources that are related to it. In this system, you need a username and a password to access the main page and if the user isn't authenticated, the login page will be shown. Also, they handle another concern which is "Design by Contact" and they state that the implementation of this concern using AspectJ is reduced. The NullContract aspect is pointcut which picks an execution of all the database methods. If any null parameter is found, then an IllegalArgumentException is thrown. Also, Cocoon itself has a form-validator, in which the characteristics of the parameters of the form are described. A property named nullable is used and indicates that the parameter can not be null, so Design by Contact concern is handled.

6. DISCUSSION

Using aspect-oriented software on the projects that are designed carefully and which obeys the code standards is a real advantage on improving the system with less code generation and more reusability. It is a fact that making a small change in the system requires a lot of changes in the whole project.

Programmer should add or remove code from a numerous class. However using aspect-oriented development saves programmer from intervening into code too much, rather, writing aspect codes for the corresponding changes is sufficient. However, since aspect-oriented development is a new technology in programming, programmers do not yet implement in a way to make the usage of aspect easier. Writing aspect codes for previously implemented programs will be harder in the sense of not using standard naming for the classes or methods. On the other hand, knowing a program can be improved later with aspect-oriented programming will enforce programmers to use standardized naming. In our project we have used JBOSS for developing aspect-oriented programming. We would also like to give a brief comparison between the aspectj and JBoss implementation. One of the most common aspect-oriented programming languages is aspectj whose documentation is actually better than the JBoss language. Therefore, not having enough experience on the subject make us use the documentation and web help for the problems encountered; however the documentation for JBoss is not adequate enough. Furthermore, the plug-in IDE for aspectj is quite better than the JBoss. The implementation of aspectJ is easier to understand compared to JBoss, however JBoss has advantages over aspectj on having the ability to provide dynamic AOP. What makes it harder to understand the implementation of JBoss is mostly the concept of "reflections" -the message passing procedure- in JBoss. However, once concepts are strengthened with aspectj, it becomes easier to understand. In aspectj, the codes are written in separate files with extension ".aj"; however JBoss codes are written into .java files, but only the pointcuts are written into a different file (".xml" files) and bind to the corresponding places with the keyword "bind".

7. CONCLUSION

In the implementation period of our e-commerce book store application, we have experienced the usage an advantages of aspect-oriented programming with the need of an improvement to the system. Even though the programmers will pay great attention in designing and implementing the projects in such a way that it will be helpful to make changes in the future, it might be quite difficult to foresee the upcoming technology and its requirements. Therefore, especially making changes to the system with aspect-oriented programming rather than object oriented programming is a high better way and makes the project for later use, more reusable and maintainable.

With AOP, adding authentication and authorization, new user types and even exception handling is much easier than the object oriented version of the solution.

On the other hand, it is worthwhile to mention that, using such new technology in programming language is a little bit difficult in the sense of finding documentation and samples on the web. However, once the basic idea is understood AOP becomes preferable over recently used methods.

Using aspect-oriented software development, helps programmers to write neater code, and enforces them to obey the code standards and design policies. This way, the programmer will not deal with the scattered and tangled code that results from crosscutting and handles the concerns without making changes all over the code.

8. ACKNOWLEDGEMENT

We would like to thank to Ass. Prof. Bedir Tekinerdogan, for teaching us the steps of the Aspect-oriented Software development during the CS 586 course and organizing the Third

Turkish Aspect-Oriented Software Development Workshop.

9. REFERENCES

[1] Lippert, M., Lopes C. V. A Study on Exception Detection and Handling Using Aspect-Oriented Programming

[2] Reina, A. M., Torres, Toro, M. Aspect-Oriented Web Development vs. Non Aspect-Oriented Web Development

[3] Kiryakov, Y., Galletly J., Aspect-Oriented Programming- Case Study Experiences, 2003

[4] Sun Microsystems Java Pet Store specification: http://developer.java.sun.com/developer/releases/petstore/

[5] Dounce, R., Motelet O., Südholt M., Sophisticated Crosscuts for E-Commerce,

[6] Dounce R., Südholt M., A Model and A Tool for Event-Based Aspect-Oriented Programming (EAOP), 2002

[7] JBOSS, http://www.jboss.org/

[8] http://www.ebuyer.com

[9] http://www.ebay.com/

[10] http://en.wikipedia.org/wiki/Electronic_commerce

Analysis and Implementation of Database Concerns Using Aspects

Murat Kurtcephe, Oğuzcan Oğuz and Mehmet Özçelik

Department Of Computer Engineering, Bilkent University
06800 Bilkent, Ankara, Turkey
{kurtcephe,oguzcan,,ozcelik} @ cs.bilkent.edu.tr

Abstract—*Data intensive applications have to deal with different concerns which cannot be easily modularized using conventional abstraction techniques, such as OOSD. In this paper we will discuss the implementation of an example database system on the case study, Car Rental System, using an aspect-oriented approach. We will focus on the concerns persistence, synchronization, failure management and refresh from database. We will report on our lessons learned as performance issues and hardship of finding the pointcuts for the aspects.*

***Keywords*—Aspect-Oriented Software Development, database, persistence**

1. Introduction

Aspect-Oriented Software Development (AOSD) mainly works on the crosscutting concerns to decrease the scattering and tangling codes. Also, it tries to increase the modularity of the system for reuse. Persistence is an already faced issue where some aspect-oriented solutions have been provided before. For an existing system, without any changes in the object-oriented design and without the knowledge of application classes about persistence, we can implement a database system to the application by aspects. Other than persistence, we dealt with synchronization, failure management and refresh from database issues. Synchronization concern focuses on the authorization of a modification operation on an object. Failure management concern deals with the database failures in a simple way. Last concern is refreshing the objects with the new data in database. This concern is a result of multi-user application. As we use the database as a way of communication between different users.

After we determined the concerns to overcome during the implementation of the database system, we decided that aspect-oriented approach would be better choice as we are dealing with a legacy code. For each concerns, we needed to change the existing code for database operations. Also, for synchronization and failure management concerns we needed to access to and change user interface codes, which is not preferable as it permits scattering codes. To maintain molarity and decrease the scattering codes, we preferred aspect-oriented approach.

Our solution includes separation of concerns for the database aspects. We divided persistence aspect in four parts. These parts consist of initialization, addition, update and deletion operations. Other concerns that we face during the implementation are synchronization, failure management and refresh from database. Synchronization aspect focuses on the modification operations. It uses a lock for the modified object and controls lock before entering the modification operation. Failure management aspect checks whether the database connection is valid before every query execution. Lastly, refresh aspect updates the data of the object each time a method of the object is called.

In [1], it is stated that persistence concern can be implemented using AOSD to be reused in other applications which does not have a database system. However, for our case we face with issues where the difference between a temporary object and a stored object is not clear. For example, an update of an object requires both old and new objects. Therefore, a reusable aspect could not differentiate whether an object is database related or not. This brings forward the issue that some projects could require implementing aspects for specific parts of the code. In our case, we faced with this problem and realized our implementation according to that.

We applied a database management system to an existing code by using aspect-oriented programming. This allowed us to increase modularity and decrease the possibility of scattering codes. We also focused on synchronization and failure management to make database management system stable. However, even if aspect-oriented approach helped us implementing some of the concerns, there were some issues which should not be ignored. We faced with performance issues in refresh from database concern and update part of persistence concern. For the first one – refresh

from database – we lacked performance but the implementation was easy. On the other hand, we implemented update part of the persistence concern in a way to increase performance which brought harder implementation. Other problem we faced was hardness to find the necessary pointcuts. Aspect-oriented approach in synchronization, refresh from database and update concerns brought this issue.

In the following, section 2 provides the overview about our case study Car Rental System. Section 3 describes the implementation methodology we use for the aspects. Section 4 focuses on the discussion. Section 5 consists of related work and in the last part there is the conclusion.

2. Case Study: Car Rental System

Our case study is Car Rental System, which is implemented in object-oriented manner. As it can be derived from the name, it is an application allows renting and reserving a car from a branch. In fact design allows many users and many branches but there was no network implementation so it forced us to use it as a single user application. However, with the database integration and refresh aspect we can use the application on different machines in the same time as an online system as the difference can be seen in Fig 1.

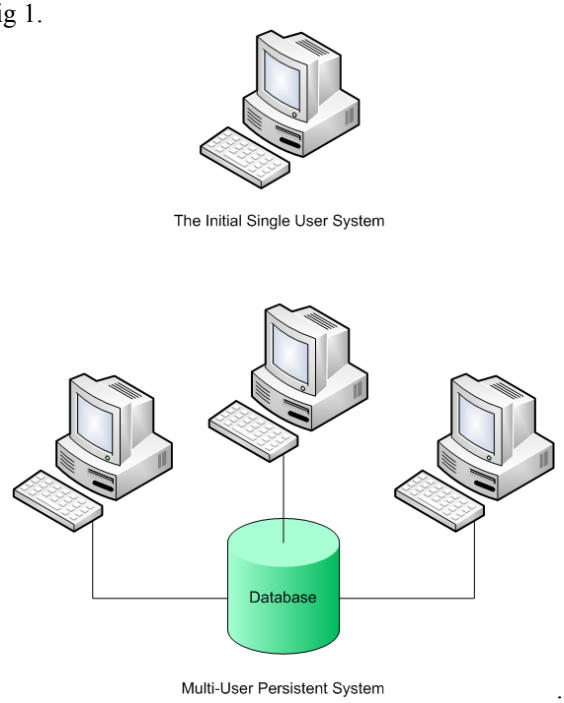


Fig. 1 The View of System After Implementing Aspects

In Car Rental System salesperson, local manager and main manager can login to the application. Before the aspect implementation, we were reading the data from a file which holds the information about the branches, customers, salesperson etc. However, with the aspect implementation, we ignore the file and use initialization aspect to create the objects using the database. After a salesperson logins to system, s/he can add&modify a customer, make rents and reservations for the already added cars and customers. The application enables salesperson to change the time and the car of the existing reservations and rents. As mentioned other than salesperson local manager and main manager can login to system. A local manager, addition to the capabilities as a salesperson, can add&modify a car description -as cars with same model, brand, year and color can be found, the application uses car descriptions where each car has a car description-, can add&modify a car, can add&modify a salesperson and can modify the information about the branch where he manages. All the additions and modifications made by a local manager is limited to the branch s/he manages. A main manager, addition to the capabilities as an local manager, can add new branches to the system and has the authority over all the branches.

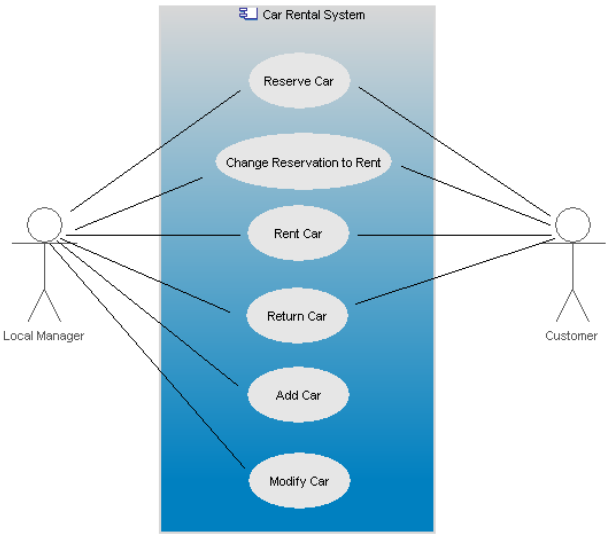


Fig. 2 Use Case Diagram of Local Manager and Customer

To pass a database system we firstly created the tables by using the attributes of the classes to be stored. Fig 3 shows a partial domain model for car rental system which corresponds to the ER diagram in Fig 4.

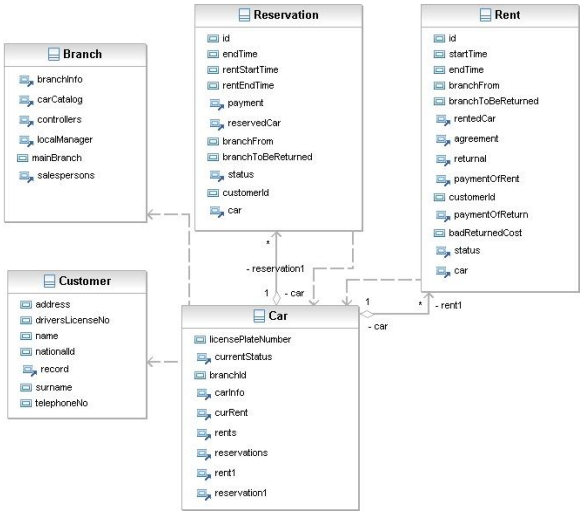


Fig.3 Partial UML Domain Model

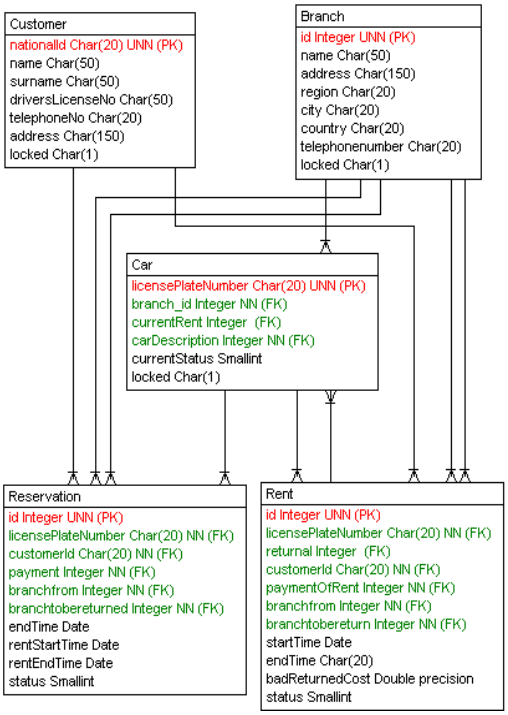


Fig.4 Partial ER diagram of the car rental system

Patterns Used in the Car Rental System

1. Abstraction-Occurrence

For the pattern we have a good example where we hold the descriptions of the cars seperated from the cars themselves. Since the description which holds brand, model, color and year of a car which could represent more than one car. Therefore we don't keep same data for different cars in themselves but in a seperated object called car description. To distinguish cars from each other we use license plate number. Each car has a unique license plate number and a car description which could be shared with other cars.

2. Delegation

As we used classes like Salesperson and LocalManager to reach the functions of Branch and MainBranch class, where the implementation of the functions are not done in both classes but in the class where all the information are hold like CarCatalog. CarCatalog has a function getCars which takes some arguments to specify the cars to return. This method is called by the function of Branch class getCars. It directly calls the function of CarCatalog not to implement the same thing. After that Salesperson reaches the needed implementation by calling the getCars function of Branch.

3. Player-Role

We have reservation, rent and car which have different status depending on the supplied situation. If a car has an accident then car has status "In Maintenance", and if it is rented it has "Rented" as the status and for each status car has it gives different responses to the callee functions. For this purpose we separated the roles,status, from car and used car status as a way to describe the current role of the car.

4. Façade

For the user interface to achieve the needed methods of the each classes we don't create an object for that class and call its functions but have class Salesperson which is some kind of interface between user interface and core classes. User interface has an instance of Salesperson and which has needed functions for GUI and this functions implements or calls the functions of other core classes.

3. Approach For Implementing Database Concerns With AOP

Applications using databases to store persistent data would have database related concerns such as persistency, synchronization, refreshment, and management of database or connection related failures. These concerns need to be addressed by most of the modules of the application and would be scattered through the application code. For this reason, these concerns cannot be perfectly modularized using solely object oriented design; an aspect oriented view is required.

1) Persistence

Persistency concern is composed of four set of operations: initialization operations, store operations, update operations, and deletion operations.

1.1) Initialization

Persistent storage for an application ensures that the application data can be stored and retrieved on demand. The application could be resumed at any time following a crash, a turn off etc. Between the cut point and resume point of an application, the persistent data may be changed by other possible applications that use the same storage. While resuming the application, the stored states and data need to be retrieved from the persistent storage and the application needs to be initialized with the retrieved data appropriately. Since the majority of the database systems are of relational databases with %80 of the market share, a relational database is used in this study. The systems that were not developed with database functionality in mind have their object models designed according to application functionality and concerns. These object models need to be transformed into relational tables to be stored in a relational database system. The resulting entity relationship model needs to be capable of storing all the required states and data to initialize the intended application. (see Fig 3 and Fig 4)

```
1 public aspect Initialization{
2
3   pointcut initializing(): execution(public static void CarRentalSystem.main(..));
4   before(): initializing()
5   {
6     //Initialize: Retrieve persistent data & Instantiate the objects accordingly
7   }
8
9   Branch initBranch(...){...}
10  Car initCar(...){...}
11  Customer initCustomer(...){...}
12
13  ...
14
15 }
```

Fig. 5 Initialization Aspect

To initialize an application with the persistent data, one would retrieve the states and data and then create instances of the objects which the application requires to run, in the memory. We propose an initialization aspect to perform initialization operations upon the start of the application. A refresh aspect that is later to be explained, cannot substitute the initializing aspect since at a point during runtime, there can be some objects that do not need to be refreshed but required to perform operations. For this reason, an initialization aspect is mandatory if we do not want to modify the existing code and modularize the initialization operations. Once the object model residing in the main memory is created, the initialization aspect needs to provide handles to the control of the application in order to ensure that the object instances can be achieved at required points. As we did implement the proposed aspects in Java, automatic garbage collection ensures that the instantiated objects can be reached provided that their handles are valid.

In the initialization aspect, shown in Fig. 5, one would benefit modularizing instantiate object operations. For every object to be initialized, there should be a initObjectName() method (see Fig 5, lines 9-11). These methods are given parameters required to specify the intended object in the database and possibly handles to other previously created objects, and the methods returns handle of the instantiated object. These methods would have high re-usability. The aspect to retrieve data from the database, for instance, would require initialization methods whenever retrieval for instantiation of an object is to be done. In fact, the mentioned methods that are specific to each object in the system can be placed in a separate aspect.

Order of instantiation of the objects is important since some objects would need handles to others and revisiting the objects would make the process more complex. In our example system, a top to bottom order is followed to minimize revisiting. The retrieval of data and instantiation of the objects could be performed in a recursive manner. Recursively instantiating objects would mean that once the initialization is started, an object is retrieved form the database and instantiated whenever it is required. In order not to fetch and instantiate same objects from the database for more than once, a caching mechanism could be employed such that handles to all previously instantiated objects would be stored in data structure in case same objects would be needed again.

During the initialization, the methods that belong to objects, such as set methods, and constructors are highly used to instantiate and set the states of the objects properly. While calling constructors and set methods, special care needs to be taken in order not to trigger other aspects that are waiting these calls to perform store or update operations.

1.2) Add methods

This concern focuses on the addition of the objects. It deals with the insertion queries of the database.

We can not easily modularize this concern by using OOSD because it needs to capture the points where addition operations are done. We have an option to change the legacy code by adding insertion queries where addition operations are done but to modularize the concern using aspect-oriented approach is preferable.

In this application, most of the objects (car, branch etc.) are held in array lists and we use this information

for our advantage. In the first part of the aspect, we get the add operations on the arrays and insert the added object to the database checking the instance of the object. In the second part, other than array list, hash tables are used to hold the data such as customer and salesperson. For these objects, as Fig. 6 shows, we get the put methods and after checking the instance type of the added object, we insert it to the database.

```
1  pointcut putMethods( Branch branch, Object addedObject ) :
2      call( * *.put( Object, Object ) ) && this( branch )
3      && args( Object, addedObject ) && !withincode( * *.get*(..) )
4      && within( core.* );
5
6  after( Branch branch, Object addedObject ) : putMethods ( branch, addedObject ) {
7      if ( addedObject instanceof Salesperson ) {
8          ...
9      }
10     else if ( addedObject instanceof Customer ) {
11         ...
12     }
13     else if ( addedObject instanceof Controller ) {
14         ...
15     }
16 }
```

Fig. 6 Pointcut and advice for addition of the objects held in hash tables

An important point which should not be ignore is that, there are some objects which refers other objects such as a rent holds a return object and two payment objects. Therefore, before inserting the rent object we insert the return and payment objects to the database.

Aspect-oriented approach minimizes the code scattering between the core classes and modularizes the addition concern in an aspect.

1.3) Update methods

Changing and modifying the objects of a program is another important issue about the persistency. Each object can be updated in a place of the execution. Therefore, the update operations in database is a cross cutting concern.

In OOP approach this could be handled by coding the update operations of database inside the code. By doing this, code starts to get tangled and it will be hard to maintain the code.

If we look the problem from AOP perspective then we will see that it is not easy to find the pointcuts where an object is update. Our case study was coded in Java, it is programming rule to declare getters and setters for each variable, and we can assume that each set operation is an update on the database. It could be very easy to implement such an approach. But when we do some search on the code we saw that each operation for example to modify a car description calls five set operations and this means the same object updated four times for nothing. It is a very simple example for our case if we think a larger object which has more that thirty features we can see the performance losing.

```
1  pointcut updateSalesperson( String oldSalespersonId
2  , Salesperson newSalesperson)
3  : execution(* Branch.updateSalesperson( String , Salesperson ))
4  && args(oldSalespersonId, newSalesperson);
5
6  pointcut setStartTimeRentOperations( Date changedTime )
7  :(execution(* Rent.setStartTime(Date)) ||
8  execution(* Reservation.setRentStartTime(Date)))
9  && args(changedTime);
10 pointcut setEndTimeRentOperations( Date changedTime )
11 :(execution(* Rent.setEndTime(Date)) ||
12 execution(* Reservation.setRentEndTime(Date))) && args(changedTime);
13 pointcut setEndTimeReservationOperation( Date changedTime )
14 :execution(* Reservation.setEndTime(Date))
15 && args(changedTime);
16 pointcut setCarStatus(CarStatus newStatus,Car target)
17 :execution(* Car.setStatus(CarStatus)) &&
18 args(newStatus) && this(target);
19 pointcut setCurrentRent(Rent rent,Car target)
20 :execution(* Car.setCurRent(Rent)) && args(rent) && this(target);
21 pointcut setPaymentAmount( double amount, Payment target )
22 :execution(* Payment.setAmount(double))
23 && args(amount) && this(target);
24 pointcut setPayment( Payment payment, Object target )
25 :call(* *.setPayment*(Payment)) && args(payment)
26 && target(target) && within(gui.*);
27 pointcut setRentStatus( RentStatus status, Rent target )
28 :execution(* *.setStatus(RentStatus))
29 && args(status) && this(target);
30 pointcut setCustomerId( String id, Object target )
31 :execution(* *.setCustomerId(String)) && args(id) && this(target);
32 pointcut setBranchToBeReturned( int branchId, Object target )
33 :execution(* *.setBranchToBeReturned(int))
34 && args(branchId) && this(target);
```

Fig 7: Some of the update operations pointcuts

Due to the performance considerations we choose another way for deciding join points. In this approach we try to define the operations which are updating the whole object the line 1 in figure shows a pointcut which is for catching the operation which updates the sales person. If each operation was working like that then we could be able to define a more reusable pointcuts. Rent operations are updating special fields of the rent (as shown in the sixth line of Fig7), such as start time of a rent. Then our first approach will fail because it is a specific field of the object, not the whole object. Therefore, we had to implement both pointcuts which are updating the whole objects, and the specific fields when needed.

After implementing the update concern, we saw that the aspect code we had is not very reusable and it is case specific. Also it is hard maintain such a code when an application has more than hundred different objects. It will be very hard to mine the aspects from the legacy code because of the reasons mentioned before. This approach (taking the operations as pointcuts) is not suitable for managing and re-usability. But on the other hand, taking the set operations as pointcuts decreases the performance of the program. There is a trade-off between performance and code re-usability.

1.4) Delete methods

In our case, the car rental system holds a local copy

of the whole application data in the main memory. The refresh aspect ensures that the local copy of the data is up to date. All the objects that reside in the main memory are unique, meaning that the application data stores exactly one copy of every object at any time. So if a remove operation is to be performed for an object in the main memory, persistent data that belong to that object in the database should also be removed. Deletion operations on the database could, therefore, trap the remove operations to be performed on the object instances and then remove the corresponding persistent data.

However, deletion by aspects does not have to be like this always, because the case specific situations mentioned above would not always hold. In [1], deletion is mentioned to be a concern that the developer should be aware of during the development of the application. This is because, the deletion should be non-fuzzy such that removal of the persistent data should be specifically declared by the application. Otherwise, deletion of an object instance may or may not require or intend deletion of persistent data. Also, automatic garbage collection mechanism of Java Language makes it hard to follow removal of object instances.

In our car rental system, the only objects that can be removed are rent and reservation, and corresponding return and payment objects. So, deletion operations on the database are applied only to the persistent data corresponding to these objects. Removal of the persistent data corresponding to shared objects should not be performed. In our case, the car rental system does not trigger any shared object deletions, however, in any other system this may not be the case. A system that has more than ten thousand lines of code could be too large to delve in to identify shared object removal triggers. This is a case in point for the requirement of non-fuzzy deletions stated in [1].

2) Synchronization

Consistency of the database is critically important for all applications which are working on database. As we know so far, the multi user systems which are working on the same data always brings the synchronization problem.

Synchronization is a cross cutting concern because it is scattered over the whole system. There can be lot of objects which has to be synchronized. For each object the synchronization concerns should implemented.

To implement the synchronization concern we used semaphores. Our semaphore mechanism works by setting a field on the database tables of each object named `locked`. When another operation tries to set the lock of the object, systems doesn't let the user to that because that object is still being modified.

Main problem is our case can be used by different users on the different branches. This means they are going to work on the same data. For example when two users log in to the system to change a car description, they should not be able to modify same car description at the same time.

For implementing the synchronization as a concern, we have to find the pointcuts first. We made the aspect mining operation by our self without using any tools because the synchronization concern is not very easy to localize. We take each modify can be done by GUI is an operation which should be synchronized. The pointcut which is defined in the first line of Fig. 8 is showing the local manager modify operation. While a user is modifying a local manager the advice shown in the fifth line of Fig. 8 is checking if that local manager is being modified by someone else. If so it gives error. Otherwise it is locking this local manager for updates. The after advice declared in the twenty fourth line of the Fig. 8 is the advice which unlocks the local manager after an update.

```
1  pointcut syncLocalManager(BranchManagerWidget caller)
2  : execution( * BranchManagerWidget.localManagerButtonClicked() )
3  && this(caller);
4
5  void around (BranchManagerWidget caller) : syncLocalManager (caller){
6      BranchManagerWidget branchManager=(BranchManagerWidget) caller;
7      int baseIndex = ( branchManager.currentPage - 1 ) * branchManager.MAX_ITEM_COUNT;
8      localManagerIndex=branchManager.managerWidgetUi.tableWidget.currentRow();
9      Branch branch = branchManager.getBranches().get( baseIndex
10     + branchManager.managerWidgetUi.tableWidget.currentRow() );
11     LocalManager localManager=branch.getLocalManager();
12
13     String updateQuery="UPDATE salesPerson SET locked=\"1\" WHERE nationalId=\""
14     +localManager.getNationalId()+"\" AND 0=locked";
15     if ( updateExecute(updateQuery)>0 ) {
16         proceed(caller);
17     }
18     else {
19         QMessageBox.critical( null, "Local manager Modify Error",
20         "Local manager is being modifiying by another user please try again later.");
21     }
22 }
23
24 after (ManagerWidget caller) returning : syncLocalManager(caller){
25     String updateQuery=null;
26
27     BranchManagerWidget branchManager=(BranchManagerWidget) caller;
28     int baseIndex = ( branchManager.currentPage - 1 ) * branchManager.MAX_ITEM_COUNT;
29
30     Branch branch = branchManager.getBranches().get( baseIndex + localManagerIndex );
31     LocalManager localManager=branch.getLocalManager();
32
33     updateQuery="UPDATE salesPerson SET locked=\"0\" WHERE nationalId=\""
34     +localManager.getNationalId()+"\" AND 1=locked";
35
36     updateExecute(updateQuery);
37 }
```

Fig. 8 Synchronization Aspect

As described above it is not an easy task to find the points which have to be synchronized in a legacy code. If it is clear that AOSD is a clear solution for this problem. The aspect code is modular and easy to maintain now but aspect code is not reusable.

3) Refresh

Main purpose of the refresh aspect is to update the information on the object by using database. As Car Rental System worked on just in a single branch, data transfer between the different users can not be established. However, with implementation of a database we can make the application work on a network of different users at the same time. We can use database to realize this network by using a refresher aspect which updates the data on main memory with the new values at the database.

This concern needs access to the all the get operations of the objects. To implement this using OOSD we need to change the existing code by adding refresh queries in get operations. As in simple manner it needs access to get methods, aspect-oriented approach is a better choice.

Current aspect gets each call for get methods of an object and applies an update to that object. If the object holds a reference to another object then we just set the id of the object to the new value from database as it can be seen in line 32 of Fig 9. When get methods for the referred objects are called their attributes will be update which will lead to a stably refreshed application.

```
1  pointcut allMethods( Object obj ) :
2      target( obj ) && ( execution( * core.*.get*(..) )
3      || execution( * core.*.find*(..) ) || execution( * core.*.search*(..) ) )
4      && !cflow( execution( public static void Fill.main(..) ) )
5      && !cflow( execution( * DataUtil.*(..) ) )
6      && !cflow( this( Persistence ) ) && !cflow( this( Refresh ) )
7      && !cflow( execution( * core.*.update*(..) ) )
8      && !cflow( execution( * *.modify*(..) ) );
9
10 before( Object obj ) : allMethods( obj ) {
11     if ( obj instanceof Branch ) {
12             ...
13         )
14         else if ( obj instanceof BranchInformation ) {
15         ...
16     )
17     else if ( obj instanceof Car ) {
18         ...
19     )
20     ...
21     else if ( obj instanceof Salesperson ) {
22             Salesperson salesperson = (Salesperson) obj;
23             selectQuery = "SELECT branch, name, surname, password, type" +
24                           " FROM salesperson" +
25                           " WHERE nationalId = '" +salesperson.getNationalId() + "'";
26
27             try {
28             query.executeQuery ( selectQuery );
29
30             ResultSet rs = query.getResultSet ();
31             if ( rs.next() ) {
32                 salesperson.getBranch().setId( rs.getInt( 1 ) );
33                 salesperson.setName( rs.getString( 2 ) );
34                 salesperson.setSurname( rs.getString( 3 ) );
35                 salesperson.setPassword( rs.getString( 4 ) );
36                 }
37             else {
38                 System.out.println( "Salesperson select query error for " +"salesperson with national
39                 +salesperson.getNationalId() );
40             }
41             }
42             catch( SQLException e ) {
43                 System.out.println( e.toString() );
44             }
45         )
46     ...
47 )
```

Figure 9: Example code from refresh aspect

As in other concerns, aspect-oriented approach helped us to modularize the system and implement the system without any change in the legacy code.

The main problem we face during the implementation of the refresh aspect is the time duration to check the database for updates. If the duration is minimal then there is a performance issue as we need to update the data for each object every time a refresh is done. However, if the time interval for refreshes is large, then we can face with synchronization issues. Since, the user can modify a data which has been modified before but has not been updated. This issue can be implemented as a synchronization problem but we did not have time to overcome that problem.

The second problem is that with the system of refreshing the ids of the referred objects we ignore the part where there are arrays of objects. If there are changes in the arrays we can not refresh those parts because we work on a single object refresh. However, for domain specific application deletion of a object or addition of a new object is not an issue. For example, adding a new car to a branch will add a row to car table of the database. On the other hand, our refresh aspect just gets the cars of the branch which were initialized at the beginning of the application but ignores the added cars after initialization. This problem can be implemented in the same way initialization aspect works, but we should check whether each object in the arrays are present in database, if not remove those objects. After this operation, check whether each row retrieved from database present in array of object, if not exist add it to the array.

4) Failure Management

This aspect mainly focuses on the database failures. It needs to access to points where database queries are called. As we implemented other database aspects this concern focuses on the aspects and tries to control the database validity before any query is executed.

To modularize the failure management concern in a specific module, other than implementing them inside other aspects such as persistence and synchronization, we choose aspect-oriented approach for our implementation.

Implementation gets the specific points where queries are executed. For each button click we check whether we have a valid connection to the database as it can be seen in line 15 and 28 of Fig 10. If there is not a valid connection, we ask for a re-connection as it can be seen in line 20 and 35 of Fig 10.

```
pointcut buttonClicked( QWidget parent ) : target( parent )
    && ( execution( * gui.*.*Clicked(..) )
    || execution( * gui.*.modifyInformation(..) )
    || execution( * gui.*.searchCustomer(..) )
    || execution( * gui.*.modifyCustomer(..) ) );

private void showConnectionFailureDialog( QWidget parent ) {
    QMessageBox.warning( parent , "Connection problem!", "Connection can"
    +" not be established! Re-try later or consult to your " +"database provider!" );
}

void around( QWidget parent ) : buttonClicked( parent ) {
    Connection con = Persistence.con;
    if ( con == null ) {
        if ( QMessageBox.question( parent, "Database Connection error!",
        "Database connection is lost! Do you want to re-connect?",QMessageBox.StandardButton
        , QMessageBox.StandardButton.No )== QMessageBox.StandardButton.Yes.value() ) {
            if ( Persistence.connect() )
                proceed( parent );
            else
                showConnectionFailureDialog( parent );
        }
    }
    else {
        try {
            if ( !con.isValid( 1 ) ) {
                if ( QMessageBox.question( parent,
                    "Database Connection is invalid!",
                    "Database connection is invalid! Do you want" +" to re-connect?"
                    , QMessageBox.StandardButton.Yes,QMessageBox.StandardButton.No )
                    == QMessageBox.StandardButton.Yes.value() ) {
                    if ( Persistence.connect() )
                        proceed( parent );
                    else
                        showConnectionFailureDialog( parent );
                }
            }
            else {
                proceed( parent );
            }
        }
        catch ( SQLException e ) {
            showConnectionFailureDialog( parent );
        }
    }
    return;
}
```

Figure 10: Example code from failure management aspect

The main problem of the current implementation is that we ignore the failure between the button click and query execution. If the system requires for a query to be executed, some time after button click we can not be sure that the connection is still valid. Therefore, this type of solution is not enough for bigger applications working on unstable database systems.

The second problem of the current applied solution occurs if there is more than one query to be executed in the database. After the button is clicked, if there is a database failure after the first query then we are expected to undone the first query. To do this, we should be aware of which type of query is executed (insert, update, delete). If it is an addition, we can delete the added object which is hold in the persistence aspect. For delete operation same mentality goes. However, for update operations we must have knowledge of old object to set back the attributes to previous values. To do that we should get the update method of the object and use a before advice, but before to do the update we get the update method and use an after advice. This means that the query always will be executed after the update method is finished, then we need to take both before and after advices for the method and keep the old object in the aspect and then use it when there is exception caused by database failure. However, this will lead to keep the all updated data in the aspect for any failure, which would cause significant performance issues if the system consist of millions lines of codes.

4. DISCUSSION

We analyzed an existing system for finding the database aspects and the implementation of the system is done by using AspectJ. In the first times, we think that it will be easy to localize the aspect from the code because our legacy code was well written and we were having all of the documentation of it. After some workings on the code, we saw that it is not easy to find the join points of the legacy code for some of the database concerns.

In our case we were using a car rental system. It was not a very complicated system. Also the legacy code was relatively short. Our technique can be considered as aspect mining on legacy code. Other mining techniques [2], [3] would not work because we were not searching for a tangling code or scattered concerns. In our approach database aspects are added as a new concern. These new concerns could be hard coded to the existing code but this would lead to coupling and low cohesion. AOP techniques should be used for having a clean code and maintainable system.

Our approach is working on the legacy code, this makes some of the concerns hard to implement. On of these concerns is the update operation of the persistence concern. Update operations can be catch by getting the set methods calls as a pointcut. When an object updated, say object has twenty features, there will be twenty different update calls to the database instead of updating the whole object. This will lower the overall performance of the system. Another issue which is similar to this exists in refresh aspect. In this aspect we are getting the get methods as a pointcut. Therefore, for each get method call of the object a select query is executed on the database. For example when user tries to search for a car refresh aspect will execute six select statements for each car. If we had more than fifty cars that will cause 300 select queries just for a simple listing operation. For having an easy coded and reusable aspect we took only the get operations for each class as a point cut for refresh aspect. But we lost the performance.

In our case, deletion part of the persistency concern was easy to adapt since the existing application does not allow any fuzzy deletions. But this is not the general case; removal of object instances would not always intend deletion of corresponding persistent data. So, deletion of persistent data needs to be explicitly declared in the application code. This requirement prevents the deletion form being fully modularized by aspects.

5. Related Work

Rashid et al. defines persistency as an aspect [1]. Their work is intended to provide an insight into the issue of developing persistent systems without taking

persistence requirements into consideration during development and adding the persistency functionality at a later time. They try to answer the question whether persistency functionality can be added to a system at later time. The conclusion they have come to, is that developers cannot be oblivious to all of the operations that are required for persistency. Still, some of the persistency operations could be left out of the development process and can be added later. Retrieval and deletion operations need to be exposed whereas store and update operations do not need to be accounted. They have also explored re-usability of persistency aspects. The suggested persistency aspect is found to be very simple to adapt and reuse; however, for effective reuse a specification that clearly defines the behavior of the aspects would be required.

Soares et al. described adding persistency and distribution aspects to an existing web based system[4]. Functionality of the existent persistency code that is embedded into the system code is replaced by the persistency aspect. Persistency aspect address following major concerns: connection and transaction control, partial object loading and object caching for improving performance, and synchronization of object states with the corresponding database entities for ensuring consistency. They identified some disadvantages of AspectJ such as lack of re-usability and proposed some minor modifications for improvement. They stated that AspectJ's powerful constructs must be used with caution to avoid undesirable side effects. Pointcuts are stated to be highly specific to a system and lacking re-usability. For these reasons, they suggest support for aspect parameterization. Persistency and distribution aspects were found to be not completely independent due to shared exception handling and synchronization mechanisms. Some of their aspects are defined as abstract and reusable, whereas others are application specific.

6. Conclusion

In this paper, aspectising database aspects such as persistence, synchronization, failure management and refreshment of the objects from database has presented. The AOP approach is usually used for eliminating tangling code but in this paper only adding the new concerns to an existing application is being considered. Modularity and the maintainability of the system considered as an important issue and not a single line of the legacy code is changed.

We had a car rental system as a case study. In this system a user was able to rent and reservation cars. The weak points of the system were: working on files and only a single user can work on the same program. The existing system should not be changed but new concerns thought to be added.

These new concerns were only about the database issues. For implementing this AOP approach is used.

We define the database aspects which should every application support such as: persistency, synchronization, and failure management and refresh operations. Each concern declared as an aspect in the system. We first find the join points for each aspect. After defining the join points the business logic of each concern coded in the aspect. AspectJ is used for implementing AOP techniques.

It this paper we tried analyze and implement database concerns for an existing code. For a car rental system proves that database aspects can be applied to a legacy system without changing its code. Also it is now easy to maintain the new added concerns.

As a future work we are planning to add a different database aspect called transactions. In this paper failure management section acts like supporting transactional behavior but real transaction mechanisms are more complicated. Also it is really hard to say that the aspects which are presented on this paper are reusable. We are planning to declare more reusable [1] aspects especially for the persistency concern. We encountered a tangling in our code. Each SQL statement could be produced according to the type of the class and the type of the operation. An aspect such, `SqlStatementProducer` can handle this tangled code in our system.

7. Acknowledgment

Thanks to Asst Prof Dr. Bedir Tekinerdogan who is the main organizer of the –Third Turkish Aspect-Oriented Software Development Workshop", for helping us during the development of this paper and sharing generously his comments and feedbacks with us.

Thanks to all participants who attend this workshop and share their ideas.

References

[1] Rashid, A. and R. Chitchyan (2003) *Persistence as an Aspect*. 2nd International Conference on Aspect-Oriented Software Development. ACM. Pages 120-129.W.-K. Chen, *Linear Networks and Systems* (Book style). Belmont, CA: Wadsworth, 1993, pp. 123–135.

[2] A. van Deursen, M. Marin, and L. Moonen. Aspect Mining and Refactoring. In *First Intl. Workshop on REFactoring:*

Achievements, Challenges, Effects (REFACE), 2003.
[3] S. Breu. Aspect Mining Using Event Traces. Master's thesis, U Passau, March 2004.
[4] S. Soares, E. Laureano, and P. Borba, "Implementing distribution and persistence aspects with AspectJ", OOPSLA, 2002, ACM Press, pp. 174-190.

Aspect-Oriented Refactoring of Visual MIPS Datapath Simulator

Hidayet Aksu, Özgür Bağlıoğlu, and Mümin Cebe
Department of Computer Engineering, Bilkent University
Bilkent 06800, Ankara, Turkey
{haksu,ozgurb,mumin}@cs.bilkent.edu.tr

Abstract

EZ_MIPS is a visual MIPS datapath simulator, produced for educational purposes, which aims to make understanding of MIPS working principle more easily. In this paper, we analyze and refactor the EZ_MPIS to enhance its quality and add some new features. The restructuring process consists of two steps. For this, we first introduce a set of object-oriented patterns. Then the tool is analyzed from the aspect-oriented perspective. We have identified the crosscutting concerns and we have refactored these concerns to aspects. In addition, we have applied enforcement aspects to ensure that the adopted polices would enhance the development process both in time and quality. We also refactor some other scattered code segments using aspects. We also add a new concern profiling to our program in order to get the execution information about the simulation. Based on our experiences we describe our lessons learned. The OOP and AOP approaches will absolutely enhance the process of design and implementation.

1. Introduction

MIPS (Microprocessor without Interlocked Pipeline Stages) which is developed a team led by John L. Hennessy at Stanford University in 1981, based on RISC architecture. The basic attempt was able to enhance performance through the implementation of instruction pipelines, which is well known but difficult to actualize [3]. MIPS focused on almost entirely on the pipeline making sure it could be run fully integrated. The most important reason for this is all the instructions need of one cycle to complete its cycle in CPU. This problem is solved by pipelined architecture that increases speed with reduced delays and CPU is used as much as possible without any gaps between instructions. MIPS has a wide variety of usage area in today's computer technology including DVD players, Networking devices such as WLAN Access points, televisions, portable devices such as digital cameras, video game consolers such as play station etc[4]. As seen from these example usage areas, we can see the impress of MIPS in every part of electronic devices. For this reason it is important to visualize the MIPS architecture in order to fully understand and become professional in MIPS architecture which is widely used.

The domain of simulations takes much attention in order to make the problem more understandable and to monitor the feasibility of the problem. With all these contributions of MIPS in mind, it is important to develop a visual MIPS for students to grasp the concepts of MIPS architecture fully. To support education activities o simulators we have developed a MIPS simulator by using object-oriented techniques which is commonly used in software development process. EZ_MIPS aims to make his task easier with a visual datapath simulator. EZ_MIPS will be helpful in understanding the MIPS architecture which is widely discussed in [5] has been taught in most of Computer Architecture courses.

In general, object-oriented approaches providing many advantages by breaking systems into unit parts to manage the whole system easily [1], but it is inadequate for encapsulating the concerns that are scattered over whole components such as concurrency, failure handling and logging [2]. These kind of scattered concerns break down the integrity and cause the coupling of different components. This causes in code scattering, and code tangling which in turn software becomes hard to maintain, because of low-cohesion and high coupling. Aspect-Oriented programming (AOP) proposes powerful solutions to this difficulty by identifying these concerns and encapsulating them efficiently. AOP efficiently separates the crosscutting concerns, by providing explicit concepts to modularize them and integrates them with the system components.

EZ_MIPS is first implemented in 2003 with agile programming. In our experiences maintaining the EZ_MIPS was not trivial, because it is implemented with agile development. In this implementation, there are some design defects and bugs. As such, we decided to analyze the EZ_MIPS and refactor it to reduce complexity and enhance maintainability. Then, the design of modules is improved by using OOP and AOP principles.

The process applied in this paper with OOP and AOP approaches, software restructuring, is an essential activity in software engineering, according to Lehman's second law of software evolution, which address increasing complexity: As a

program evolves, its becomes more complex unless work is done to maintain or reduce it[11].

Shortly, in this paper, the EZ_MIPS will be refactored from the perspective of object-oriented approaches, and whenever it is not sufficient aspect-oriented approaches. The purpose is to make the program more stable by using de-facto patterns and cleaning code from crosscutting concerns, scattering code segments finally purifying design by means of aspect-oriented approaches.

The rest of this paper is organized as follows. In the next section, we will give a short introduction of selected case, EZ_MIPS, for reengineering with OOP and AOP approaches. The object-oriented reengineering process using well known design patterns applied to EZ_MIPS are given in section 3. Then we continue with detailed identification of crosscutting concerns and scattered code segments in the simulated model and we discuss the solution architecture of these concerns using aspect-oriented approaches. In section 5, we mention related work, and finally in section 6, we provide our concluding remarks.

2. Problem Case: EZ_MIPS

As explained before the simulation model discussed in this paper is based on the MIPS instruction set. Having an idea about how a computer datapath works and in what manner a datapath could be improved, are of fundamentals of computer science. However, sometimes, it becomes difficult to understand the manner in which a given datapath works through the assistance of books. At that times usage of some tools, with which the user enable to interact, become valuable to overcome these difficulties. EZ_MIPS was designed to be one of these tools. EZ_MIPS is, simply, a simulation of the datapaths that are discussed in computer architecture courses. EZ_MIPS makes it exciting to study on and easy to understand MIPS datapaths. For each datapath (Single Cycle Datapath, MultiCycle Datapath, and Pipelined Datapath [5]) there is a dedicated edition of EZ_MIPS.

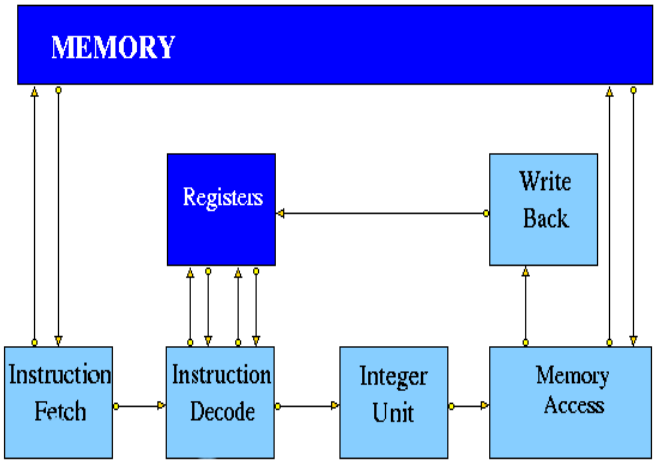


Figure 1. Typical MIPS Pipeline

As visualized in

Figure 2 a MIPS processor can overlap the execution of several instructions, potentially leading to big performance gains by using pipelined approach. MIPS composed of several parts that process the instructions in a pipelined manner. The fundamental parts are instruction memory, registers, arithmetic logic unit, forwarding unit, data memory, and multiplexers [6]. The architecture composed of 32 general purpose registers. MIPS registers take both less time to access and have a higher throughput than memory The data types used in MIPS are byte, halfword (2 bytes), and word(4 bytes). As seen from

Figure 1 the instruction is fetched by MIPS, then it is decoded according to instruction type and the necessary values are read from registers. Then in integer unit (ALU) the execution of instruction is done. ALU is used for data and address arithmetic, so load and store instructions are processed by ALU before sent to memory access unit and memory. Also, ALU is responsible from the additions required for integer test and relative branch instructions so that the Memory Access Unit executes branches. Lastly, if required the memory access is called and the results from both arithmetic/logic and load instructions are written to registers by write back unit [7].

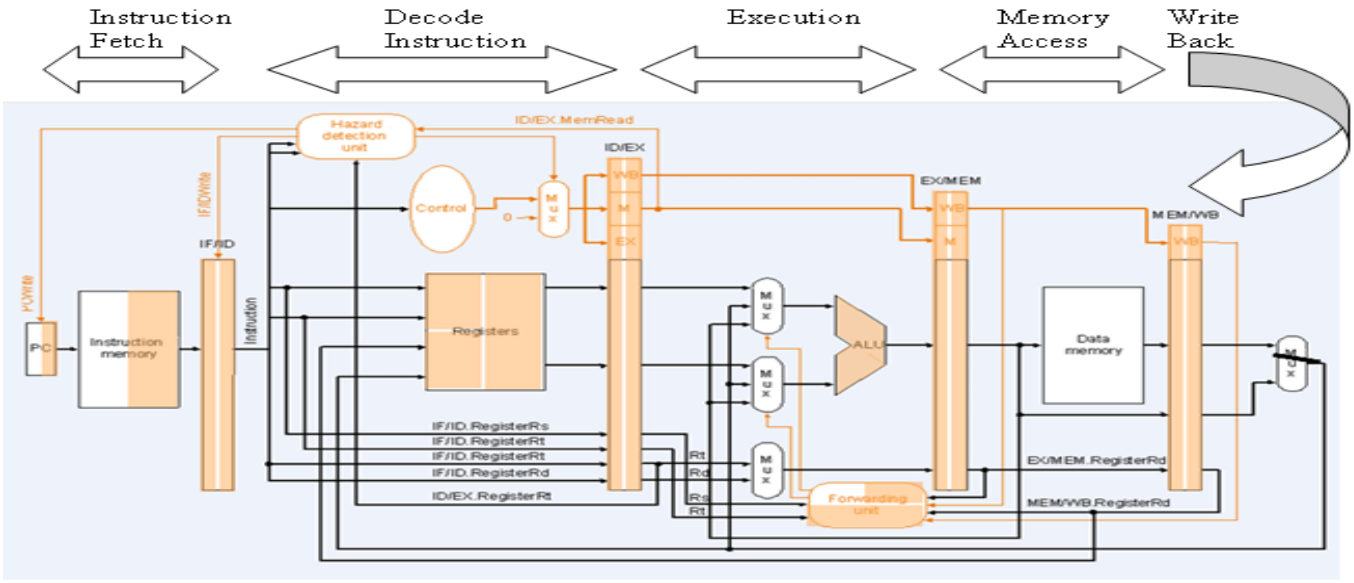


Figure 2. MIPS Visual Datapath [Taken from EZ_MIPS]

The projection of

Figure 1 is implemented in

Figure 2 which is implemented by EZ_MIPS. During the execution of instructions, all these units are mostly in busy state because of pipelined structure. The instructions are executed as shown in

Figure 3.

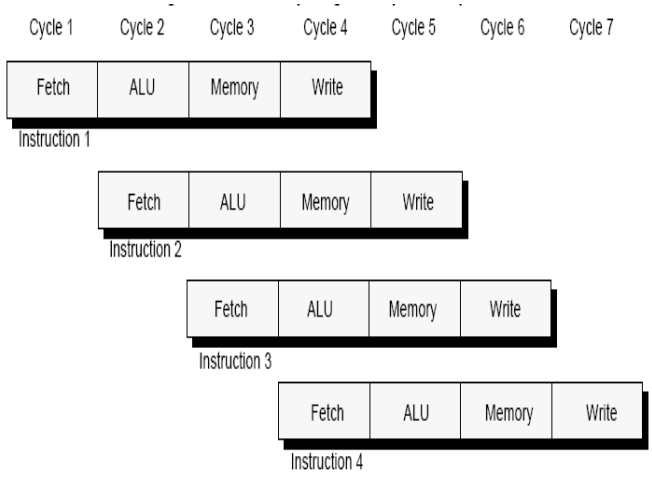


Figure 3. Pipelining of instructions via MIPS[8]

Lastly, EZ_MIPS gives the full details of all cycles going in all units of MIPS by datapath visualization tool, data memory and register values. Also the ALU execution can be monitored within any time in execution of programs via tooltips and external windows. Also the simulator has a variety of look and feel options that provides more appealing interfaces.

The main features of EZ_MIPS are as follows:

- Control of execution speed, including single step to variable speed
- Thirty-two registers and 1KB memory unit visible at the same time, selectable via tables,
- "spreadsheet" (WYSIWYG) modification of values in registers and memory and instruction sets,
- Selection of data value display in binary, decimal or hexadecimal formats
- "surfing" through memory using memory window
- Toolbar icons for every menu item
- Visual Datapath simulator with various tooltip options to simulate instuctions at different speeds.
- Different look-and-feel options.
- Have opportunity to start execution of instructions from selected instruction step of instruction set.
- Modification of instructions read from file in anytime.
- Showing the status of program in information bar.
- Profiling of environment variables in simulation (new feature comes from AOP approach)

After defining the properties and features of simulator, the next sections mention about analysis of its design from various perspectives.

3. Restructuring with Design Patterns

When we look at the current implementation of EZ_MIPS there are some cases where best practices of engineering have not been adopted. By using the reverse engineering approaches, some of the patterns will enhance the usability of codes and makes system more stable, less coupled and cohesive. In these kinds of situations improving the quality of design is done by adopting

existing design patterns to these problematic parts. In this section, the problems with current design or the demands needed about current design are analyzed and adopted patterns are explained in detail.

For reengineering of patterns, we followed a migration strategy which is an incremental approach widely used in reengineering of current systems. The strategy starts with analyzing the system from the object-oriented perspective, finding the demands and problems in current design and prototyping the target solution for the defective parts and incrementally migrating from current to target solution. In this process, we always have a running version in all steps of migration. And with each unit of change in design we tested the system's stability. After attaching new patterns we have more cohesive and less coupled code. In the following lines, we present our changes by using existing patterns.

3.1. Abstract Factory Pattern

In EZ-MIPS simulation, we provide to users to change GUI themes easily. As the simulator aims to teach MIPS structure easily, we want to user can change the theme of GUI that he/she feel more appealing and comfortable with it. Considering the desired functionality, each time we want to create different types of themes it would require a considerable amount of afford to add new theme to simulator. By using the Creational Design Patterns, the amount of afford is going to decrease seriously. Also, our main concern is creating different themes with separating the details of implementation of a set of themes from the execution of system.

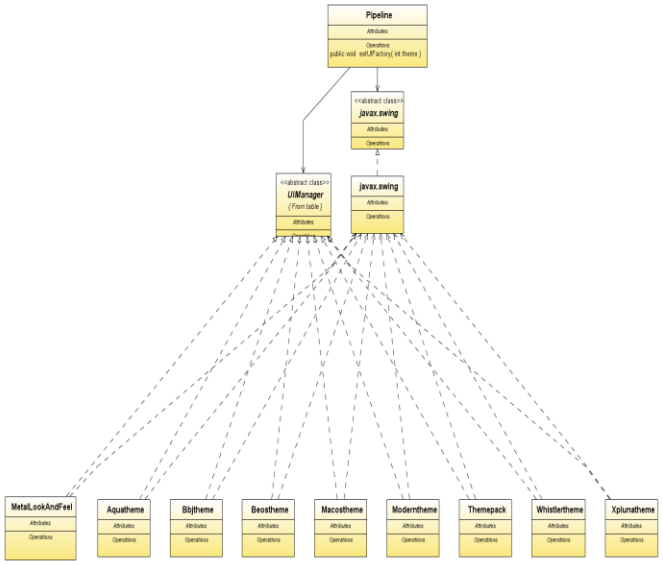


Figure 4. EZ_MIPS Abstract Factory Implementation

Abstract Factory promises us the encapsulation of a group of individual factories that have a common theme and the usage of generic interface to create concrete objects that are part of the theme [9]. By using Abstract Factory pattern, the client does not know the theme of concrete objects since it uses only the generic interfaces of their products. We provide to use a set of theme factories by defining UIManager class that declares an interface for creating different types of themes. Thus, by only selecting the specific theme option user will stay independent of the implementation details how it is performed. Figure 4 shows the class diagram of the abstract factory design of the classes.

3.2. Observer Pattern

The observer pattern (sometimes known as publish/subscribe) is a software design pattern in which an object maintains a list of its dependents and notifies them automatically of any state changes, usually by calling one of their methods [9].

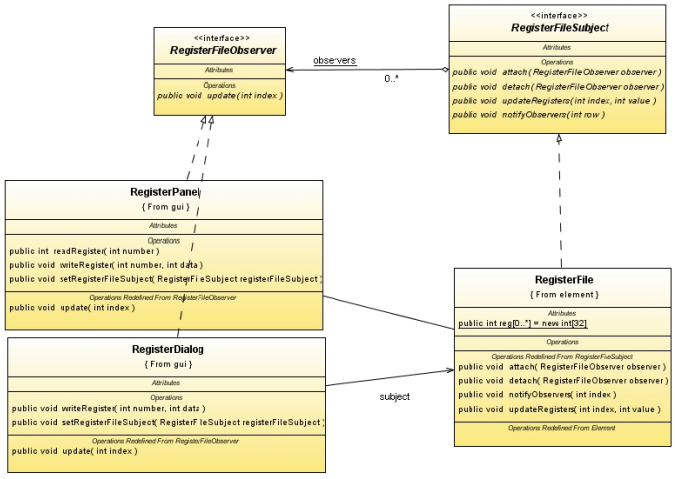


Figure 5. EZ_MIPS Observer Pattern

In EZ-MIPS simulation, we have two windows (Register Panel and Register Dialog) to show the value of registers at the time of the current instruction set. Throughout the step by step execution of instruction set, these two windows should be notified about the new values on the registers. To provide this functionally, we use Observer Pattern. Before applying Observer Pattern, we face many difficulties. For each element in MIPS, we should keep tracks and updates the associated windows values. This cause to hard to manage to code, when there is a need for modification. However, in this pattern there are two components subjects and observers [9]. Subjects

represent the core data and when a modification is occurred on the data, subject notifies all the attached observers to update themselves.

Figure 5 shows the diagram of pattern in our design.

3.3. State Pattern

In EZ_MIPS simulation environment, users have the chance of tracing the given instruction set by using three kinds of number states. These states are: binary state, decimal state, and hexadecimal state. The number states are implemented in all elements of EZ_MIPS Simulator. Without this pattern we have to add all the elements switch statements, which makes code less readable. To solve this problem we proposed a state pattern which is a family of behavioral pattern. As Gamma mentioned, this is a clean way for an object to partially change its type at runtime which is exactly our case [9]. The modified UML diagram is shown in Figure 6.

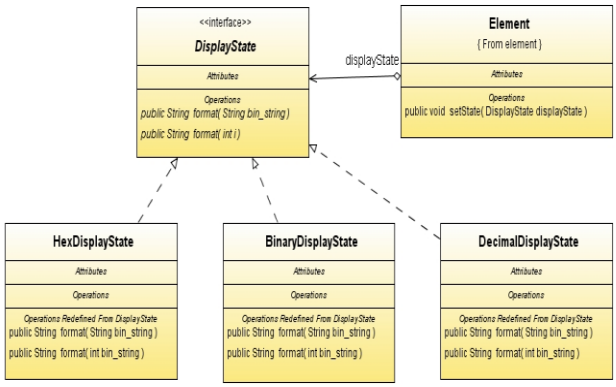


Figure 6. EZ_MIPS Sample State Pattern

4. Introducing Aspects

Aspect-oriented software development is a relatively new approach to software development and design, which points out limitations of object-oriented software development. Aspect-oriented software development adds crosscutting concerns to address those features that cut across modules [10].

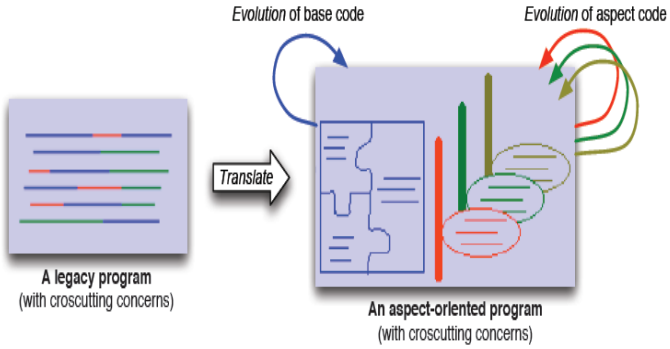


Figure 7. Cross fertilization of software code with AOP[11]

As visualized in Figure 7, in order to perform aspect-oriented development, we have to translate existing code into their aspect-oriented equivalents and re-factor them continuously [11]. This refactoring process starts with identification of concerns scattered in implementation, redesigning of them using AOP principles and implementing these concerns.

The reverse engineering of EZ_MIPS implied that there are some crosscutting concerns and scattering code in the simulator. Although we apply OOP paradigms to EZ_MIPS simulation, we can not exactly provide encapsulation and separation of concerns. These concerns should be removed to provide modularity and clarity of the implementation.

In this section, we will mention two kinds of aspects; development and production aspects. Development aspects which can be easily removed from production builds aims to the development process easier and faster. Production aspects are intended to be included in the production builds of an application. Production aspects insert functionalities to an application rather than only adding more visibility of the internals of a program [12].

4.1. Policy Enforcement

The policies and contracts are the core components in the project's life cycle. These rules make all the development more concrete, organized, and systematic. So the enforcement policies are necessary for robust development and coordination. In our reengineering process, we have defined some rules to improve code in a team. By these rules, the development process becomes more comfortable. Some of these implementation policies are as follows:

1. There should not be no System.out.{print*(),error*()} in the project. Because this project is GUI based, there is no

meaning of these calls. We give warning when there is a call.

2. The Class names, method names and variable names should be compliant with Java's coding conventions. This is for making code more readable.

Another problem is some of the object creations should not be called from some modules. This may include the creation and execution of MIPS elements being in one object (e.g. DataPath class). This is a powerful mechanism for development of project. Enforcing this constraint makes the program more stable and unnecessary or wrong creation of MIPS Datapath elements will not be allowed. Note that these policies and enforcements are regarded as development aspects which are so important in the process of project implementation lifecycle. Here is the following sample showing policies that are discussed:

```
pointcut consoleout():(get(* System.out) || get(* System.err)) &&
!within(aspects..*);
public pointcut classWithLowerCase() :
        initialization( pipeline..a*.new(..))  ||
        initialization( pipeline..z*.new(..));
public pointcut methodWithUpperCase() :
        execution(* pipeline..*+.A*(..))  ||
        execution(* pipeline..*+.Z*(..));
declare warning :
    consoleout() :
    "output to console not allowed. Consider using logger.";
declare warning :
        classWithLowerCase() :
        "class names should not begin with lower case letter.";
declare warning :
        methodWithUpperCase() :
        "method names should not begin with upper case letter.";
```

4.2. Consistency Verification

Another crosscutting concern is for providing the correctness of execution of each element. Each time we need to control whether the code of element fine or not, we must put some checker into pre and post conditions of each operations. So this means each time we want to check if the retrieved input is correct, and the result of the operation is done correctly, we must put the validation control into these methods or operations. This is why we implement an aspect called Consistency Verification Aspect to encapsulate common issues associated with this concern.

In the system all the wires are implemented as an object. For example ALU unit must get 2 objects created by 2 MUX elements shown in Figure 2. Also the inputs must have some characteristics (such as the format of incoming elements). As this example other elements have some pre and post conditions (object creation, object format of input and output elements). These concerns are done by consistency verification. And this aspect eases the debugging and verification of the flow of simulation. Note that the example that is discussed is implemented in the following aspect:

```
pointcut aluWork(ALU alu ):this(alu) &&
(execution(void ALU.work(boolean)));

Wire aluOut = WireSet.wires[35];
Wire aluZeroOut  = WireSet.wires[36];
Wire aluInA = WireSet.wires[32];;
Wire aluInB = WireSet.wires[34];;

before  (ALU alu ): aluWork(alu) {
    // null check
    if ( aluInA == null || aluInB == null )
        throw new IllegalArgumentException("ALU inputs are not valid.");
    // bounds check
    if ( aluInA.getValue().length() > 32 || aluInB.getValue().length() > 32 )
        throw new IllegalArgumentException("ALU inputs are is out of bounds.");
}

after  (ALU alu ): aluWork(alu) {
    // null check
    if ( aluOut == null || aluZeroOut == null  )
        throw new IllegalArgumentException("ALU output is not valid.");
    // bounds check
    if ( aluOut.getValue().length() > 32 )
        throw new IllegalArgumentException("ALU output is out of bounds.");
    // bounds check
    if ( aluZeroOut.getValue().length() > 1 )
        throw new IllegalArgumentException("ALU Zero output is out of bounds.");
}
```

Note that there are two verification points, before ALU execution process and after ALU execution process. This aspect makes the input and output bounds checking and validations.

4.3. Tooltip Concern

Another crosscutting concern that we have faced during reengineering is originated from the need to inform user the current state of each element in EZ_MIPS simulation. To do so, we show a tool tip panel that represents previous and current status of element at any clock cycle. It interests all or the elements current state. Some of these elements are instruction memory, ALU, registers, hazard detection unit, data memory, forwarding unit. The tooltip includes information about the inputs and

outputs of each unit, the inner status of each element (such as registers and memory segments), and execution status of elements. This makes the maintenance of tooltip harder, if performed with OOP principles. Also when some updates are necessary in tooltip, these modifications affect all parts of various elements in design and implementation which makes maintenance and development harder. Figure 8 shows the effects of this crosscutting concern over the implementation. Here, because the tool tip concern and execution of MIPS concern are indeed separate concerns, these two concerns should be separated to carry out the modularity.

```
if (control == "010") {       // addition
     result = a + b;
     tooltip = "<html>" + colorA + a + End + " + "
     + colorB + b + End + " = " + colorR + result
     + End + ' ';
 } else if (control == "110") {     // substruction
     result = a - b;     //
     tooltip = "<html>" + colorA + a + End + " - "
     + colorB + b + End + " = " + colorR + result
     + End + ' ';
 }
  else if (control == "000") {      // and
     result = a & b;
     tooltip = "<html>" + colorA + a + End + " end
     " + colorB + b + End + " = " + colorR +
     result + End + ' ';
 } else if (control == "001") {     // or
     result = a | b;
     tooltip = "<html>" + colorA + a + End + " or
     " + colorB + b + End + " = " + colorR +
     result + End + ' ';
 }
```

Figure 8. Crosscutting Code Segment for Tooltip

To separate ToolTip concern from business logic, we propose to use ToolTip Aspect shown in Figure 9. In this aspect, we define a pointcut called to captures calls to work method of ALU element. After that, we define an advice to simply set the tooltip of ALU element. By the help of AOP technology, we have differentiated the crosscutting tooltip concern from execution logic. To do so, we increase the modularity of the system by encapsulating the aspect code.

```
public privileged aspect TooltipAspect {
        pointcut adderwork(ALU alu ):this(alu) &&
          (execution(void ALU.work(boolean))|| execution(* Element.setState(..)));

        after(ALU alu ): adderwork (alu)
  {
                if(alu.output == null ){
                        alu.setToolTipText("ALU....");
                        return;
                }
                tooltip = "<html>" + colorA + formatted_a + End + op + colorB
                                + formatted_b + End + " ="
                                + colorR + formatted_out + End
                                + ' ' + "<br>aspect ZeroOut: " + colorR
                                + (wireSet.wireSet[36].getValue())  + "</font>"
                                + "</html>";

                alu.setToolTipText(tooltip);

        };
}
```

Figure 9. Example Tooltip Aspect

4.4. Profiling of Simulation Variants

The next crosscutting concern is for profiling the execution of the each elements of the simulation. To measure the totality of element's behavior from invocation to termination, to make statistical summary of the events observed and to see ongoing interaction, we need a profiler that crosscuts all elements in the EZ_MIPS. Thus the same separation of concerns also is need for the profile concern. Considering this, we decided to develop an aspect called Profiling Aspect. With the help this aspect, there is no need to make any modification of implementation of elements in the simulation. We just implement the desired profiling requests for each element in Profiling Aspect. Figure 10 represents a sample profile window at the end of the execution of EZ_MIPS.

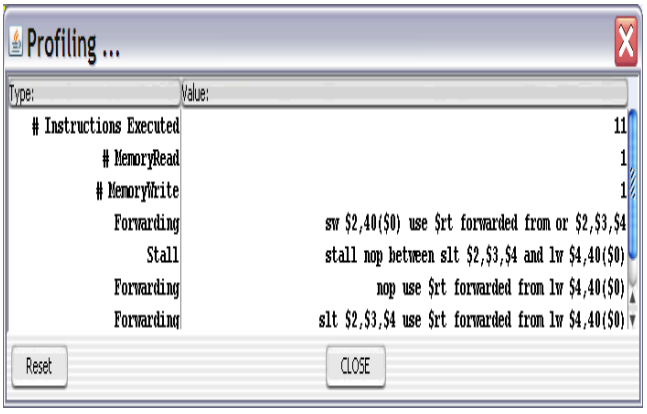


Figure 10. Example Profiling Aspect

The implementation details of profiling is done as follows:

```
pointcut endOfMain():
  (execution(* Main.main(..)));
pointcut intructionMemoryWorks(InstructionMemory instr, boolean clockEdge):
    this(instr) && args(clockEdge) &&
execution(void InstructionMemory.work(..));
pointcut dataMemoryRead(DataMemory dataM):
    this(dataM) && execution(* DataMemory.readMemory(..));
pointcut dataMemoryWrite(DataMemory dataM):
    this(dataM) && execution(* DataMemory.updateMemory(..));
pointcut forwardingUnitWorks(ForwardingUnit forwardingUnit):
    this(forwardingUnit) && execution(void ForwardingUnit.work(..));
pointcut hazardDedectionUnitWorks(HazardDedectionUnit hazardDedectionUnit):
    this(hazardDedectionUnit) && execution(void HazardDedectionUnit.work(..));
```

```
void updateInstructionsExecutedCount() {
    profilingMemory.numOfInstructionsExecuted++;
    profilingMemory.model.fireTableDataChanged();
}
void updateMemoryReadCount() {
    profilingMemory.numOfMemoryRead++;
    profilingMemory.model.fireTableDataChanged();
}
void updateMemoryWriteCount() {
    profilingMemory.numOfMemoryWrite++;
    profilingMemory.model.fireTableDataChanged();
}
void addMessage(String type, String msg) {
    profilingMemory.addMessages(new ProfileMessage(type, msg));
}
void processForwardingMessage(ForwardingUnit forwardingUnit) {
    String instr00 = WireSet.labelSet[3].getText();
    String instr01 = WireSet.labelSet[5].getText();
    String instr10 = WireSet.labelSet[4].getText();
    if (forwardingUnit.mux3a.getValue().equalsIgnoreCase("01")) {
        addMessage("Forwarding", "\"" + instr00+ "\" use $rs forwarded from \"" + instr01 + "\"");
    } else if (forwardingUnit.mux3a.getValue().equalsIgnoreCase("10")) {
        addMessage("Forwarding", "\"" + instr00+ "\" use $rs forwarded from \"" + instr10 + "\"");
    }
    if (forwardingUnit.mux3b.getValue().equalsIgnoreCase("01")) {
        addMessage("Forwarding", "\"" + instr00+ "\" use $rt forwarded from \"" + instr01 + "\"");
    } else if (forwardingUnit.mux3b.getValue().equalsIgnoreCase("10")) {
        addMessage("Forwarding", "\"" + instr00+ "\" use $rt forwarded from \"" + instr10 + "\"");
    }
}
void processHazardDedectionMessage(HazardDedectionUnit hazardDedectionUnit) {
    String instr00 = WireSet.labelSet[2].getText();
    String instr01 = WireSet.labelSet[3].getText();
    if (hazardDedectionUnit.mux.getValue().equals("1")) {
        addMessage("Stall", "stall nop between " + instr00 + " and "
                + instr01);
    }
}
```

```
after(InstructionMemory instr, boolean clockEdge):
    intructionMemoryWorks ( instr, clockEdge){
    updateInstructionsExecutedCount();
}
after(DataMemory dataM):
    dataMemoryRead ( dataM ){
    updateMemoryReadCount();
}
after(DataMemory dataM):
    dataMemoryWrite ( dataM ){
    updateMemoryWriteCount();
}
after(ForwardingUnit forwardingUnit):
    forwardingUnitWorks ( forwardingUnit ){
    processForwardingMessage(forwardingUnit);
}
after(HazardDedectionUnit hazardDedectionUnit):
    hazardDedectionUnitWorks ( hazardDedectionUnit ){
    processHazardDedectionMessage(hazardDedectionUnit);
}
```

In these code segments first pointcuts are introduced which will crosscut all execution parts of EZ_MIPS: data memory, instruction memory, forwarding unit, and hazard detection unit. Then the necessary aspect codes are added for profiling of simulation elements.

6. Discussion & Conclusion

As far as we explained in this paper, we have improved existing code quality with object-oriented and aspect-oriented thinking. In the process of restructuring the existing design and implementation we have performed the following two operations:

1. Improved the code quality in order to make code more cohesive, modularized and less coupled. This process is regarded as software refactoring, which aims developing the internal structure of a program without changing its external behavior. This process includes adding new design patterns to our existing design and finding existing concerns or tangling code segments scattered over objects such as tooltip concern.
2. Add new features using aspect-oriented techniques. In our project, we insert profiling feature for the distinct points of simulation in order user to monitor the MIPS behavior better.

Within this process we have visualized that aspect-oriented development eases the implementation of crosscutting concerns and improves reuse of code which is a fundamental principle of software engineering. As explained in [13] and [14] AOP provides the following advantages:

- The components of software are better modularized.
- The modularization provides better separation of concerns and a clearer and cohesive responsibility of the system components, and therefore the end products are better maintainable and reusable.
- These advantages lead to reduction of time-to-market by a better modularized and simpler design, resulting in a reduction of costs.

To conclude, we had a chance to make our hands dirty with experience on AOSD. Also by practical usage of aspects we give a proof of concept that aspect-oriented development will ease the implementation of crosscutting concerns, which are regarded as secondary requirements for modules. We also improved the quality of code by eliminating tangling code segments. In the development

process with AOP techniques, we experienced that the deployment time of the program takes time as addition of new aspects. The only problem may be this concern which makes the development a little bit slower. However in reengineering process AOP development is faster than OOP development, which needs special analysis and management of scattered codes and crosscutting concerns.

Acknowledgement

Thanks to Asst. Prof. Dr. Bedir Tekinerdogan, who organized the "Third Turkish Aspect-Oriented Software Development Workshop" (TAOSD 2008), for generously sharing his comments and feedback with us.

Also thanks to our class mates and all the participants of the TAOSD 2008, with whom we had a most beneficial workshop, sharing experiences related to AOP.

References

[1] C. Timothy, "*Object-Oriented Software Engineering"*, Lethbridge and Robert Laganiere, McGraw-Hill.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C.Maeda,C. Opes, J. M. Loingter, J.Irwin, *"Aspect-Oriented Programming"*, In Proceeding of ECOOP'97, Springer-Verlag LNCS, 1241

[3] MIPS Architecture. http://en.wikipedia.org/wiki/MIPS_architecture. Last accessed 15 December 2008.

[4] T. Langens *et al*, "MIPS Architecture",2000.

[5] A. D. Patterson, J. L. Hennessy, "Computer Organization and Design: The Hardware/Software Interface". Third Edition. Morgan Kaufmann. 2004.

[6] S. Jonathan, "http://gicl.cs.drexel.edu/people/sevy/architecture/MIPSRef(SPIM).html." last accessed 15 December 2008. The hardware / software interface

[7] HASE Project, "Simple MIPS Pipeline", Institute for Computing Systems Architecture, School of Informatics, University of Edinburgh

[8] MIPS Technologies Inc**.** MIPS32® Architecture For Programmers, "Introduction to the MIPS32® Architecture"

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison Wesley Longman Inc., 1995.

[10] B. Sue, H. Mark, *"Aspect-oriented Software Development: Towards a Philosophical Basis"*, Technical Report. Department of Computer Science, King's College London. 2006.

[11] T. Mens, K. Mens, T. Tourw´e. *"Software Evolution and Aspect-Oriented Programming"*. ERCIM. 2004.

[12] Introduction to AspectJ http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html. Last accessed 18 December 2008.

[13] R. Laddad, AspectJ in Action, "Practical Aspect-Oriented Programming",Manning Publications Company, 2003.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and G. Griswold, William, *"Getting Started with AspectJ,"* ACM Communications, vol. 44, no. 10, pp. 59–65, 2001.

Developing Aspects for a Discrete Event Simulation System

M. Uğur Aksu, Faruk Belet, Bahadır Özdemir
Department Of Computer Engineering, Bilkent University
{aksu, fbelet,bozdemir} @ cs.bilkent.edu.tr

Abstract

Simkit is a simulation modeling tool. Like many simulation systems, Simkit implements a number of concerns, such as event scheduling, event handling, keeping track of a simulation's state and creating random number generators. It appears that these concerns crosscut over multiple modules in the system. This increases the complexity and reduces the maintainability and ease of use of this tool. In this paper, we identify the crosscutting concerns in Simkit and try to refactor it with Aspect Oriented Software Design.

1. Introduction

Simulation is the practice of using abstraction and idealization to produce less complex models of real life systems, whose present or proposed alternative behavior can be exercised in a feasible and less costly way. [1, 2]

Event Graphs are a way of graphically representing DES models, like Petri Nets, System Dynamics and UML. In a high level view, Event Graphs employ nodes and edges; to denote events and to scheme interactions among events, respectively. Very often, Event Graphs are preferred due to their expressive power.

Simkit is an open source Discrete Event Simulation (DES) tool that has been developed for educational purposes. It has been designed with object oriented approach and the Java language has been used for the implementation of it. Simkit aims to make implementation of Event Graph models as easy as possible. In this respect, Simkit simply extends the basic Event Graph paradigm by a component architecture that is based on loose coupling of simulation components. [3]

In this paper, we argue that some general and domain specific concerns of Simkit are crosscutting over multiple modules. This increases the complexity and reduces the maintainability and ease of use of the tool. This argument can be justified by the fact that the domain requirements for simulation systems are fairly complex and object oriented paradigm is not capable enough to deal with such complexity.

Concerns we face in Simkit can be architecturally divided into two groups: (1) Concerns that apply directly to the Simkit packages itself that implement library functions. (2) Concerns that appear in the components/classes defined by simulation modelers.

We propose to refactor Simkit using Aspect Oriented Software Development (AOSD) paradigm. AOSD is an emerging software development technology that seeks to modularize software systems by expressing scattered and tangling concerns separately from the core logic of the systems. It is usually practiced to enhance readability, modularity, maintainability and extensibility of software systems.

The remainder of this paper is organized as follows. In section 2, we provide a short background on simulations and DES modeling with Event Graphs. In section 3, we discuss the object oriented design of the Simkit and explain the Event Graphs in more detail with its corresponding implementation in Simkit. In section 4, we try to identify crosscutting concerns of Simkit and propose solutions i.e. aspects in the parlance of AOSD paradigm, to the crosscutting/tangling concerns observed. In section 4, we discuss our findings from this study. Finally, in section 5, we provide our conclusion and describe related future work.

2. Simulations & Discrete Event Simulation (DES) Modeling with Event Graphs

Simulation is "the process of describing a real system and using this model for experimentation, with the goal of understanding the system"s behavior or to explore alternative strategies for its operation" [1] Using simulations we can improve our understanding of systems" behavior and the effects of interactions among components.

Simulation systems can be categorized in different ways and one of them is based on the properties of a simulation model"s state variables. [2] Using this classification approach, models can be described either as static or dynamic. Dynamic models can be further categorized into discrete event or continuous models. In Discrete Event Simulations (DES), all system state changes are mapped to discrete events, assuming nothing relevant happens in between, while in continuous simulations states change steadily over time. [2]

There are three fundamental components for DES models. These are a set of events, a set of state variables and the simulation time.

In DES, events occur at the points in time at which state variables change values and simulation time is updated to the current event's time. Events are kept in an Event List which is simply a to-do list. It holds information regarding what event is being scheduled and the simulation time at which the event is to occur. [1]

Event Graphs, a way of graphically representing DES models, uses nodes and edges to depict DES models. Nodes denote events and edges are used to schedule other events with two optional parameters: a Boolean condition and/or a time delay. [1] Figure 1 shows the basic construct for event graphs.

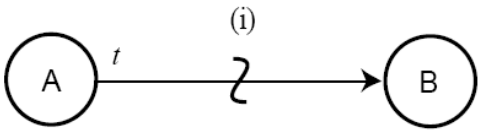


Figure 1. Fundamental Event Graph Construct [1]

Figure 1 is interpreted as follows: When event A occurs, event B is scheduled to occur at time *t* if the condition *i* is true.

More detail on DES and its corresponding implementation in Simkit will be elucidated in the next section.

3. Object Oriented Design of Simkit and its Correspondence to Event Graphs

Simkit is an Event Graph based DES modeling tool with an implementation of general purpose high level programming language, augmented by a package of suitable simulation related library procedures. Every element in an Event Graph model has a corresponding element in Simkit.

With regard to the object oriented design, three fundamental packages of Simkit are: (1) simkit: where the core simulation related logic is implemented; (2) simkit.random: a random number generation factory and (3) simkit.stats: a utility package to collect simulation related statistics.

A simplified object oriented design of Simkit that depicts some core components is shown in the Figure 2.

DES related core components of Simkit that implement event handling, state changes and simulating time are explained at the next subsections. More detail on DES with Simkit can be found at [4].

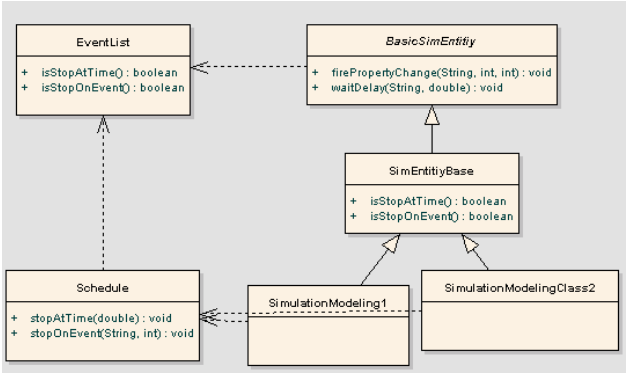


Figure 2. A Simplified Object Oriented Design of Simkit

3.1. Future Event List

As all DES frameworks require an implementation of a Future Event List (FEL), Simkit implements a FEL in a class called simkit.Schedule. FEL is represented by a variable java.util.SortedSet in the package simkit. Schedule class and it contains objects of type SimEvent that contains data on which event it represents and a scheduled time to occur information.

Instead of directly placing events on FEL, the programmer invokes the waitDelay() method on an instance of simkit.SimEntityBase so that the details of the FEL could be hidden from the modelers.

3.2. SimEntity and SimEntityBase

An abstract class and an interface are employed in Simkit to help encapsulate the activities with regard to scheduling and processing events.

Any class designed to interact with the FEL must implement the SimEntity interface that specifies a set of methods for interaction.

User defined "do" methods in a subclass of SimEntityBase are used to implement the behavior of events. The naming convention for "do" methods is as the event name followed by a "do" prefix. Behavior of scheduling edges are implemented by "waitDelay()" methods for which the simplest signature is as such: (string, double), where the first argument is the name of the event with out the "do" and the second argument is the delay associated with the scheduled event.

The Simkit code corresponding to Figure 1 is implemented in the Figure 3, to elaborate more on user defined "do" methods and calling of waitDelay() constructs.

```
public void doA() {
    <code to perform state transition for event A>
    if (i) {
        waitDelay("B", t);
    }
}
```

Figure 3. Simkit Code for Basic Event Graph Construct [4]

As stated before, the first argument in the waitDelay() methods is a string that stands for an event name. Using Java"s reflection feature, Simkit determines the corresponding method by creating a SimEvent in SimEntityBase and adding it to the FEL. Then, when an event occurs, the Event List invokes a callback method on the object that scheduled it.

3.3. Event Handling Mechanism of Simkit

Event handling convention of Simkit is as follows: When an event occurs, first the simulation time is updated to the time of event that has occurred and all state changes associated with that event are performed. Next, all further events are scheduled and finally the event notice is removed from the Event List.

3.4. Sample Simkit Models

In this subsection, adopting the domain specific graphical notation of Event Graphs, we will explain two sample DES models that will be used in this paper. Figure 4 models a very simple simulation case: Arrival Process.

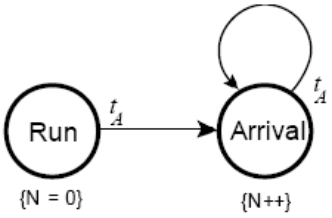


Figure 4. Event Graph for Arrival Process [1] [4]

The initialization of a Simkit simulation run is achieved by a dummy event called Run. At the start of the simulation, a Run event both initializes the state variables and schedules the initial events on the Event List. The initialization of state values are achieved by a method called reset(), while the scheduling of initial events is done by the doRun() method. In the model, there exists a single state variable, numberArrivals (N), which counts the cumulative number of arrivals since time 0.0 and it is incremented by one upon the occurrence of each Arrival event.

After explaining the basic constructs of the model, we can interpret Figure 4 as follows: A Run event is scheduled to occur at simulation time 0.0. With the occurrence of the Run event, first simulation time is set to 0.0 and the state variable N is set to 0. Then an Arrival event is scheduled to occur at time t_A. In the same way, for each Arrival event that is run, first the simulation time is updated to the time of the Arrival event, then state variable N is incremented by one and another Arrival event is scheduled to occur at time t_A.

The Simkit implementation of the Arrival Process is given in the Figure 5.

```
private RandomVariate interarrival;
protected int numberArrivals;
public void reset() {
    numberArrivals = 0;
}
public void doRun() {
    waitDelay("Arrival", interarrival.generate());
}
public void doArrival() {
    firePropertyChange("numberArrivals
        ",numberArrivals, ++numberArrivals);
    waitDelay("Arrival", interarrival.generate());
}
```

Figure 5. Simkit Code for Arrival Process [4]

Another example is the Multiple Server Queue case. In this model, customers arrive to a service facility according to an arrival process and are served by one of the k servers. Customers arriving to find all servers busy wait in a single queue and are served in order of their arrival time. [1]

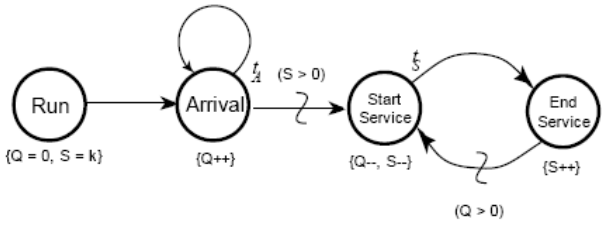


Figure 6. Event Graph for Multiple Server Queue [1]

State variables and parameters for the model in Figure 6 are: Q = # of customers in queue, S = # of available servers, t_A = interarrival times, t_S = service times, k = total number of servers.

3.5. Design Patterns Used in Simkit

In the Simkit library, several design patterns such as observer, factory, builder, and adapter are used. Observer and factory design patterns are the most important ones in the library. In general, we can not state any significant shortcoming in the design of Simkit with regard to design patterns. Yet, we have added another design pattern i.e. façade that facilitates the usage of Simkit for modelers.

3.5.1. Observer Pattern

Simkit uses two observer patterns to implement its component interoperability. [4]

First, the SimEventListener pattern is used to connect simulation components in a loosely coupled manner. SimEvents are always invoked by a callback to the scheduling object that invokes the corresponding "do" method. The SimEvent is then dispatched to every SimEventListener that has been explicitly registered in that object"s SimEvents. [4]

Secondly, the PropertyChangeListener pattern comes into play whenever a state variable changes value. In that case, a PropertyChangeEvent is dispatched to registered PropertyChangeListener objects. The purpose of PropertyChangeEvents is to support generic observation of the simulation state trajectories, as well as any function thereof. [4]

3.5.2. Factory Method Pattern

Simkit uses a combination of RandomVariate interfaces and a factory that is called to produce instances of the desired implementation using only "generic" data which is strings, objects, and numbers. [4]

3.5.3. Façade Pattern

Running a Simkit model requires the implementation of the following tasks:

1. Instantiate desired objects.
2. Register SimEventListener objects.
3. Register PropertyChangeListener objects.
4. Set stopping time or stopping criteria.
5. Set the mode of the run.
   (verbose/quiet, single-step/continuous running)
6. Reset all SimEntityBase instances.
7. Start the simulation.

First of all, if the programmer wants to collect statistics for a property, a new statistics object should be created for each property. In addition, if the programmer desires to keep a log of a process, a new property dumper object needs to be created for each process. Moreover, SimEventListener and PropertyChangeListener objects need to be created. And finally, all these objects should be registered to the corresponding process objects. To sum up, all these operations creates confusion in the main method. Even though there is an interface in the Simkit library, i.e. BasicSimEntitiy that combines elements of a simulation, programmers are still to write simulation classes that implement this interface. [4]

In order to lessen the burden on modelers we have designed a Façade class, "Simulation", which offers a generic interface. For this aim, we have also designed an auxiliary class, "Process", which stores elements of processes. Simulation objects are constructed with an array of Process objects as an argument. The statistics and listener objects are created and registered by the Simulation class. After this point, the programmer should set the parameters for simulations if needed and just run.

The simplified class diagram of the façade pattern is shown in figure 7.

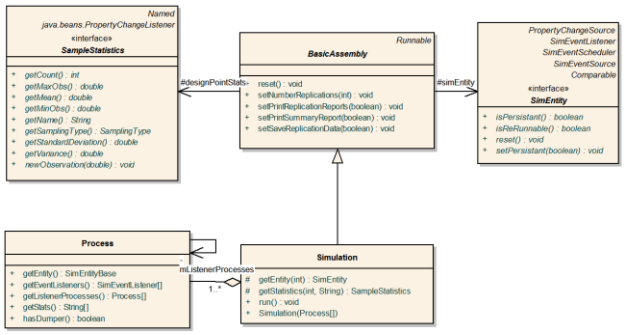


Figure 7. The Simplified Class Diagram of the Façade Pattern

4. Identification of Crosscutting Concerns & Proposed AOSD Solutions

In AOSD approach aspects are generally categorized into three categories: (3) Core Aspects: Form the core part of the system; (2) Development Aspects: Helps the development of the system; (1) Enforcement Aspects: Policies that are desired to hold in the design of the system. We will describe the crosscutting and/or tangling aspects detected in the Simkit by adopting this classification.

4.1. Production Concerns

4.1.1. Simulation Termination Rules

There are a number of ways for terminating a DES. Most commonly used four methods are as follows: (1) Specifying a fixed length model time for the simulation. (2) Defining a simulation termination event. (3) Defining a fixed length sample size. (4) Terminating the simulation in terms of the model"s state.

Taking the best software development advices into use, this concern of terminating simulations can be implemented cohesively and separately from the other simulation related concerns.

For the AOSD solution, we have defined an aspect that combines the first three termination rules that has been implemented in the Schedule and EventList Classes in the Simkit. We used static introduction to move the related code to the target aspect. Moreover, in this aspect, we

have implemented the last termination rule as a new feature, which enables terminating the simulation in terms of the model"s state.

An AspectJ code example for this solution is demonstrated in Figure 8.

```
privileged public aspect SimStopRulesAspect {

    public pointcut somePointInSimRunLoop(simkit.EventList eventList) :
          execution(public boolean simkit.EventList.isSingleStep()) && this(eventList);

    before(simkit.EventList eventList) : somePointInSimRunLoop(eventList) {
        eventList.checkStopState();
    }

    private int simkit.EventList.stopValue = 0;
    private simkit.SimEntityBase simkit.EventList.simObject = null;
    private String simkit.EventList.propertyName = "";
    private boolean simkit.EventList.isStopAtStateIfGreater = false;
    private boolean simkit.EventList.isStopAtState = false;

    public static void simkit.Schedule.stopAtTime(double atTime) {
        defaultEventList.stopAtTime(atTime);
    }

    public static void simkit.Schedule.stopOnEvent(int numberEvents,
            String eventName, Class... eventSignature) {
        defaultEventList.stopOnEvent(numberEvents, eventName, eventSignature);
    }

    public void simkit.EventList.stopAtTime(double time) {
        // Related code..
    }

    public void simkit.EventList.stopOnEvent(int numberEvents,
            String eventName, Class... signature) {
        // Related Code..
    }
    public static void simkit.Schedule.stopOnEvent(int numberEvents,
            String eventName, Class... eventSignature) {
        defaultEventList.stopOnEvent(numberEvents, eventName, eventSignature);
    }

    public void simkit.EventList.stopAtTime(double time) {
        // Related code..
    }

    public void simkit.EventList.stopOnEvent(int numberEvents,
            String eventName, Class... signature) {
        // Related Code..
    }

    public static void simkit.Schedule.stopAtState(int stopValue,
            boolean isStopAtStateIfGreater, simkit.SimEntityBase simObject,
            String propertyName) {
        defaultEventList.stopAtState(stopValue, isStopAtStateIfGreater,
                simObject, propertyName);
    }

    public void simkit.EventList.stopAtState(int stopValue,
            boolean isStopAtStateIfGreater, simkit.SimEntityBase simObject,
            String propertyName) {
        this.stopValue = stopValue;
        this.simObject = simObject;
        this.propertyName = propertyName;
        this.isStopAtStateIfGreater = isStopAtStateIfGreater;
        this.isStopAtState = true;
    }

    public void simkit.EventList.checkStopState() {
        // Related code..
    }
}
```

Figure 8. AspectJ Code for Simulation Termination Rules Aspect

The pointcut in the code snippet above locates a point in the simulation running loop and the before advice is used to inject the "stop the simulation at state" related code at that point.

4.1.2. Persistence: Restoring a Simulation Run

Persistence turns out to be a very generic production concern that has a fairly significant crosscutting and tangling nature in almost any software system that is considerably large.

Simulation runs are almost always costly in terms of time. Yet, simulations might terminate unexpectedly due to a number of reasons such as interruptions by misbehaving operating systems or misbehaving other applications. It is not only frustrating but also time consuming to rerun a simulation. Moreover, sometimes simulations can be terminated properly by users but later on, it might be desired to run the same simulation for larger sample sizes. In such cases, it comes in very handy if a simulation can resume from its last stable state.

To achieve such a persistence feature, a simulation program/package must remember some fundamental properties of the previous simulation run such as: (1) Scheduled but not handled events, in other terms, last stable state of the Event List. (2) Values of the simulation state variables. (3) Random number generation objects and the number of times they have been pooled before the simulation termination.

We have provided a persistence solution using AOSD only for the first two properties described above. Even this partial implementation proved to exhibit significant crosscutting view for the concern.

We have employed 3 aspects in the aspect oriented design. StoreEventListStateAspect stores the last stable Event List; StoreSimVariablesStatesAspect stores the last stable values of the state variables. These values are stored in files each time an event is handled. And the last Aspect PersistenceAspect reads the values stored in files for both state variables and scheduled events, then resumes the simulation run if a method call such as resumeSimulation(true) is provided by the user.

AspectJ code examples for the persistence aspect are demonstrated in Figure 9.

```
public privileged aspect StoreEventListStateAspect {

    private static Formatter fileOutput;

    public void simkit.EventList.storeEventListState() {
        StoreEventListStateAspect.storeEventListState(this);
    }

    public static void storeEventListState(simkit.EventList eventList) {
        // Related code: stores event list state
    }

    public pointcut storePoint(simkit.EventList eventList) :
          call(boolean simkit.EventList.isStopOnEvent()) && this(eventList)
          && (withincode(public void simkit.EventList.startSimulation()));

    after(simkit.EventList eventList) : storePoint(eventList) {
        eventList.storeEventListState();
    }
}
```

```
public aspect StoreSimVariablesStatesAspect {

    private static Map stateVariableTable = new TreeMap();
    private static Formatter fileOutput;

    pointcut propertyChangeMethods(String propertyName, int newPropertyValue):
        call(* simkit.BasicSimEntity.firePropertyChange(String, int))
        && ! within(simkit..*) && args(propertyName, newPropertyValue);

    before(String propertyName, int newPropertyValue) :
        propertyChangeMethods(propertyName, newPropertyValue) {
        stateVariableTable.put(propertyName, newPropertyValue);
    }

    public void simkit.EventList.storeStateVariablesStates() {
        StoreSimVariablesStatesAspect.storeStateVariablesStates();
    }

    public static void storeStateVariablesStates() {
        // Related code: stores state variables' states
    }

    public pointcut storePoint(simkit.EventList eventList) :
          call(boolean simkit.EventList.isStopOnEvent()) && this(eventList)
          && (withincode(public void simkit.EventList.startSimulation()));

    after(simkit.EventList eventList) : storePoint(eventList) {
        eventList.storeStateVariablesStates();
    }
}



privileged public aspect PersistenceAspect {

    public static boolean isResumeSimulation = false;

    public static void simkit.Schedule.resumeSimulation(boolean isResume) {
        PersistenceAspect.isResumeSimulation = isResume;
    }

    public pointcut resetProperty() :
        ! within(simkit..*) && set(protected int simkit.SimEntityBase+.*)
        && withincode(public void reset());

    after() : resetProperty() {
        // Related code: initiliaze using the last state variables' states
    }


    pointcut locateDoRun(simkit.SimEntityBase object):
        execution(* simkit.SimEntityBase+.doRun()) && ! within(simkit..*)
        && target(object);

    void around(simkit.SimEntityBase object): locateDoRun(object) {
        // Related code: set the event list to the last stable state recorded
    }

    public static String readEventListState() {
        // Related code: read event list from a file
    }

    public static String readSimVariablesStates() {
        // Related code: read state variables from a file
    }
}
```

Figure 9. AspectJ Code for Persistence Aspect

The pointcuts in the first two aspects locate points in the simulation running loop. At these points, codes related to writing to files are injected by after advices. The after advice in the third aspect, resets the state variables" values to the values read from the stored file and the around advice puts the events read from the file to the Event List instead of letting the doRun method start the simulation from the very beginning.

The crosscutting nature of this concern is demonstrated in Figure 10.

Figure 10. Crosscutting Nature of the Persistence Concern

4.1.3. Resource Pooling Concern

Resource pooling is a system-wide and generally a scattered concern, that is used to keep the resources created earlier around instead of disposing them, so that they can be reused later on instead of recreating new ones. A common example can be given for database applications where instead of creating and destroying database connection objects each time we need, previously created objects that are returned from the resource pools can be used. This feature is especially important in terms of time-efficiency. While resource pooling usually decrements the time needed of obtaining a resource, this benefit faces us with the challenge of extra memory and other resources consumed by resource pool to store objects created earlier. This is called the space/time tradeoff of resource pooling. Taking this tradeoff into account, a system designer must enable resource pooling only if the desire for improved speed outweighs the cost of extra memory.

In Simkit, we believe that keeping a resource pool for random number generation objects might come very handy to increase simulation compilation time.

In Simkit, random number generation is achieved by a group of factory classes in the simkit.random package. Each of these factory classes implement their own resource pools in a scattered way, to address the concern (faster simulation compilations) explained above. Factory classes initialize a random number generator object whose class name is taken via getInstance(string) method of related factory. By this approach, factory classes implement lazy initializations by searching packages or adding some suffix such as "variate" to the className. Behind getInstance(string) and its variations, each factory keeps a cache of previously created objects. At this point, crosscutting concern appears due to because a caching mechanism that is spread over different factories. To modularize and generalize this caching approach in system wide manner, we convert the scattered caches into a single caching aspect. A simplified class diagram of resource pooling concern is shown in Figure 11.

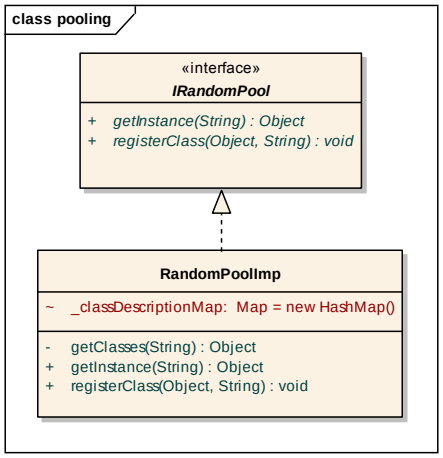


Figure 11. Resource Pooling Class Diagram

In this way, when a new random number generation factory is desired to be added to the simkit.random package, resource pooling mechanism for this newly added factory does not have to be implemented separately, which would be the case for a pure object oriented design that does not take advantage of AOSD paradigm. An aspect code example for this solution is given in Figure 12.

```
public aspect RandomPoolingAspect {

    /**
     * Holds the registered resources in the resource pooling
     */
    IRandomPool _randomPool = new RandomPoolImp();

    pointcut instanceCreation(String className):
        (call(* simkit.random..*Factory+.findFullyQualifiedNameFor(String))
        || call(* simkit.random..*Factory+.getClassFor(String)))
        && args(className);

    /**
     * around advise that returns objects from pool if it exist in the resource pool
     * using RandomPoolImp class.
     *
     */
    Object around(String className) : instanceCreation(className) {
        Object myClass = _randomPool.getInstance(className);
        //Related Code..
        //Return or register object in resource pool
    }
}
```

Figure 12. AspectJ code for Resource Pooling

4.1.4. Observer Pattern Concern

Observer/Listener pattern is widely used especially in multi-threaded programs. When a property of a class observed changes, then related functions of all listener classes are called. However, this cause tangling of fire property change codes into the observed class and scattering of observer pattern. In addition, this requirement may cause defects in the code. For example, the programmer may forget to add the code for calling fire property change functions affects the correctness of a program. To overcome this problem, a core aspect which automatically detects changes of state variables and handles calling corresponding fire property functions can be added.

In Simkit, when a change in state variables occurs, a corresponding fire property change function should be called. Providing an aspect that handles the call of fire property change functions will crosscut the process classes.

The programmer can change the values of state variables inside “do” methods, in Simkit. These changes should be detected by an aspect and corresponding fire property change functions should be called. Changing of state variables outside the “reset” and “do” methods is intended to be restricted by another aspect. Thus, finding property changes in “do” methods is sufficient because fire property change function is not invoked for “reset” functions. Corresponding fire property change functions are different in doRun and other “do” methods. As a result, we divided detecting property change into two parts. In the first part, we only detect property changes in “do” methods other than doRun.

Around advice is used for the pointcut which identifies property changes in “do” methods, because fire property change function should be invoked after setting the value to property; however, we need both old and new values of the property.

In the second part, we just need to identify calls to doRun methods. We use a pointcut which identifies calls to a specific library method in which “do” methods are invoked. Before advice is used to invoke the related fire property change functions at these pointcuts.

An AspectJ code example for this solution is given in Figure 13.

```
public aspect FirePropertyChangeAspect {

    public pointcut propertyChange(Object value) :
        ! within(simkit..*) && set(protected int simkit.SimEntityBase+.*)
        && withincode(public void do*()) && ! withincode(public void doRun())
        && args(value);

    void around(Object value) : propertyChange(value) {
        SimEntityBase target = (SimEntityBase) thisJoinPoint.getTarget();
        String field = thisJoinPoint.getSignature().getName();
        Object old = target.getProperty(field);

        proceed(value);
        // Related call: call corresponding firePropertyChange function

    }

    public pointcut callDoRun(SimEvent event) :
        call(public void simkit.SimEntityBase.processSimEvent(SimEvent))
        && args(event);

    before(SimEvent event) : callDoRun(event) {
        if (event.getFullMethodName().equals("doRun()")
                && event.getSource().equals(thisJoinPoint.getTarget())) {

            SimEntityBase target = (SimEntityBase) thisJoinPoint.getTarget();
            Field[] fields = target.getClass().getDeclaredFields();

            for (Field field : fields) {
                // Related call: call corresponding firePropertyChange function
            }
        }
    }
}
```

Figure 13. AspectJ Code for Observer Pattern Aspect

4.2. Development Concerns

4.2.1. Checking Main Method's Parameters

Checking main method"s parameters is a classic idiom adopted widely by the AOSD community [5]. This feature also proves to be very useful in our case. Checking the parameters separately from the core business logic of the main method offers a more cohesive main method, thus preventing tangling of other concerns in the main method.

In our implementation we employ an around advice. If proper parameters are not fed by the user, first correct usage is printed out, and then program control is transferred to the main method with default parameters.

An AspectJ code example of this idiom is shown in Figure 14.

```
public pointcut captureMain(String[] args) :
      execution(void *.main(String[])) && args(args);


void around(String[] args) : captureMain(args) {
    if (args.length < 2 || args.length > 3) {
        args = new String[] { "someString", "anotherString" };
        System.err.println(CORRECTUSAGE);
    }
    proceed(args);
}
```

Figure 14. Idiom for Validating Parameters Passed to a Main Method

4.3. Enforcement Concerns

Asking simulation modelers to comply with a set of constraints is a very noticeable tip that tells us to employ enforcement aspects. Using enforcement aspects, we not only get a more robust design but also get rid of the tangling concerns we would have otherwise.

4.3.1. Imposing Constraints on Users for State Variables

Each state variable in Simkit models is implemented by an instance variable with protected access and should have public getter methods but setter methods, so that only events can change state variables by calls to firePropertyChange(propertyName, oldValue, newValue) methods. However, state variables are still susceptible to illegal modifications inside the classes in which they are declared or by setter methods defined by careless simulation modelers.

For the first Simkit usage constraint, where the users are asked not to define setter methods in order to prevent illegal state variable sets, we offer a solution in which first we define a pointcut that locates undesired set accesses and then by using a static declaration we notify the user by a compilation warning.

An AspectJ code example of this approach is demonstrated in Figure 15.

```
pointcut illegalSetAccesses():
    ! within(somePackage) && set(protected * someClass.someVariable)
    && !(withincode(public void aMethod()) ||
         withincode(public void anotherMethod(..))) ;

declare error : illegalSetAccesses() : "ERROR MESSAGE ";
```

Figure 15. AspectJ Code for Restricting Set Accesses to State Variables

Another constraint on simulation modelers with regard to state variables is as follows: For the first arguments of each property change method, there must be a matching state variable declaration.

For this concern, we offer a solution that makes use of Java Collection feature together with aspects. We use Java Reflection to keep a record of all the declared state variables. Then, we employ a pointcut to get the first argument of each property change methods and by a before advice we compare this argument with the previously kept record of state variables.

An AspectJ code example for this solution is demonstrated in Figure 16.

```
public aspect PolicyEnforcementForStateVariables2 {

    List<String> declaredStateVariables = new ArrayList<String>();

    pointcut locateASimulationClass():
        execution(simkit.SimEntityBase+.new(..)) && !within(simkit..*);

    before():locateASimulationClass(){
        // Related code:
        // collect context for state variables by Java Reflection
    }

    pointcut propertyChangeMethods(String propertyName):
        call(* simkit.BasicSimEntity.firePropertyChange(String, ..))
        && ! within(simkit..*) && args(propertyName, ..);

    before(String propertyName) :
        propertyChangeMethods(propertyName) {
        if (!declaredStateVariables.contains(propertyName)) {
            System.err.println("ERROR MESSAGE ");
            simkit.Schedule.stopSimulation();
            System.exit(1);
        }
    }
}
```

Figure 16. AspectJ for Checking State Variables Declarations

4.3.2. Imposing Constraints on User for Calling "do" Methods

Each event in a Simkit model is implemented as user-defined "do" methods. And events are scheduled by calls to waitDelay(string, delay) functions. Simkit uses Java"s reflection to determine the corresponding method for scheduled events. Normally, the programmer does not have to deal with this method. The Simkit library implements a method to invoke the related "do" method indicated by the programmer using waitDelay(string, delay) function.

First of all, when there do not exist corresponding instance methods, for example, such as doRun() and

doArrival() for events such as Run and Arrival respectively, the simulation will run incorrectly. This problem might be caused by either misnaming of event names given as arguments in the waitDelay(string, delay) constructs or by misnaming in the event handling methods.

For this concern, we propose a solution which ensures that first parameters of each event scheduling methods such as waitDelay(eventName, delay) matches with the name of an event handling method without the „do" prefix, such as doEventName(). We again employ the methodology of using Java Reflection together with Aspects for this implementation.

An AspectJ code example for this solution is demonstrated in Figure 17.

```
public aspect PolicyEnforcementForEventHandling perthis(locateASimulationClass()){

    List<String> declaredDoMethods = new ArrayList<String>();

    pointcut locateASimulationClass():
        execution(simkit.SimEntityBase+.new(..)) && !within(simkit..*);

    before():locateASimulationClass(){
        // Related code:
        // collect context for declared event handling methodsby Java Reflection
    }

    pointcut schedulingEventMethods(String scheduledEventName):
        call(public simkit.SimEvent simkit.SimEntityBase+.waitDelay(String, ..))
        && ! within(simkit..*) && args(scheduledEventName, ..);

    before(String schedulingEventName) :
        schedulingEventMethods(schedulingEventName) {
        if (!declaredDoMethods.contains(schedulingEventName)) {
            System.err.println("ERROR MESSAGE ");
            simkit.Schedule.stopSimulation();
            System.exit(1);
        }
    }
}
```

Figure 17. AspectJ Code for Enforcing Policies on "do" Methods

Secondly, there is no restriction which prevents calling the "do" methods by the programmer. Invoke "do" methods outside the Simkit library may result in changes in state variables and event list causing the simulation to run incorrectly. Therefore, an enforcement aspect which prevents calling "do" methods by the programmer is required for these conditions. To do this, first the points where a "do" method is called outside the Simkit library should be identified by a pointcut. Then, these types of calls are prevented by declaring error at this pointcut.

An AspectJ code example of this approach is demonstrated in Figure 18.

```
pointcut illegalCallToFirePropertyChange():
    ! within(simkit..*) && within(simkit.SimEntityBase+)
    && call(public void firePropertyChange(..));


declare error : illegalCallToFirePropertyChange()
    : "ERROR MESSAGE";
```

Figure 18. AspectJ Code for calling firePropertyChange

4.3.3. Imposing Constraints on User for Firing Property Change Methods

As described before, changes in values of state variables are monitored by a core aspect. This aspect invokes corresponding fire property change functions. Calling these functions by the programmer cause execution of these functions more than one time may result in erroneous behavior of the simulation or inaccurate statistics. Therefore, the library needs an enforcement aspect which ensures that the fire property change functions are not invoked by the programmer.

To prevent calling firePropertyChange methods, first the points where a firePropertyChange method is called should be identified. Finally, these types of calls are prevented by declaring error at this pointcut.

An AspectJ code example of this approach is demonstrated in Figure 19.

```
pointcut illegalCallToDoFunctions():
    ! within(simkit..*) && call(public void do*(..));

declare error : illegalCallToDoFunctions() :
    "ERROR MESSAGE";
```

Figure 19. AspectJ Code for Calling "do" Methods

5. Discussion

In this paper, we have tried to demonstrate mostly simulation systems specific concerns that are crosscutting/tangling and tried to develop aspects for an Event Graph based DES tool i.e. Simkit.

There are a number of observations that deserves noting in our study.

First of them is, some concerns such as simulation termination rules, restoring a simulation run and creating random number generators are crosscutting over multiple modules inherently for simulation systems. This is mostly due to the complexities associated with these concerns. For instance, in order to restore a simulation run, at least three essential properties of a simulation run need to be remembered. These properties are; the event list, the simulation"s state and the random number generation objects. Since each of these features are usually implemented in separate components in a coherently and loosely coupled way, we are to interact with all of these modules to restore a simulation run.

Secondly, we have noticed that, using a high level programming language as a simulation tool that does not offer a graphical user interface, imposes many constraints on simulation tools users. Policies the modelers are asked to comply when creating event handling methods, scheduling events, declaring state variables and firing property change events are just a few to name to give examples. Such concerns can be overcome by enforcement aspects efficiently while reducing the tangling issues.

Moreover, our observation is that when working on legacy code for refactoring purposes, the feature of using Java Reflection feature and aspects in a combination might offer a convenient solution. The justification that lies behind this assertion is as such: (1) It may turn out to be very costly in terms of time to concoct other solutions. Usually, delving into somebody else"s code to understand the design and provide a solution requires a lot of time. (2) Even when there can be found AOSD solutions, crosscutting modules need to be identified by code mining. Instead, using both Java Reflection and aspects, we can find solutions to some problems, working only on one module or class. We have used this idiom successfully in two enforcement aspects developed for the Simkit.

Finally, one striking question comes to mind. What if we would like to implement all these concerns with another AOSD framework other than AspectJ. In order to make a comparison, we choose JBoss as another framework and focus on how aspects are implemented in these frameworks.

We can begin with pointcuts which are the heart of any AOP implementation. JBoss supports pointcuts defined in XML and by using Java 5 annotations. On the other hand, AspectJ supports language based pointcuts and Java 5 based annotations. AspectJ and JBoss become quite a bit different when it comes to the pointcut syntax. However, expression power seems to be similar.

Another point to discuss is joinpoints. JBoss operates joinpoints as much more like an event framework while AspectJ brings a new approach which is a little hard to learn for beginners. For java programmers, JBoss AOP is easier to implement any joinpoint.

Finally, we compare their build and deployment sides. It seems that it is simpler to make deployment in AspectJ; just compile your code with AspectJ compiler and put aspectjrt.jar in your classpath and you are good to go. This is true regardless of whether you"re developing a Java application or a Swing application. JBoss AOP deployment is a little more involved. The simplest means of deploying an aspect is to put the jboss-aop.xml in the META-INF folder of the Jar. However, if you are using an annotation, it needs extra configurations for the system. As a conclusion, AspectJ is handier for building and deployment of the system.

5. Conclusion

Simkit is a simulation modeling tool and like many simulation systems, it implements a number of concerns such as event scheduling, event handling, keeping a track of simulation"s state and creating random number generators. It appears that these concerns crosscut over multiple modules in the system. This increases the complexity and reduce the maintainability and ease of use of this tool. In this paper, we have identified the crosscutting concerns in Simkit and tried to refactor it with Aspect Oriented Software Design.

For the results of our study, first of all, we have proved one more time that using only object oriented paradigm is not sufficient to manage all the concerns, i.e. the crosscutting concerns.

Secondly, we have shown that, concerns such as simulation termination rules, restoring a simulation run (persistence) and creating random number generators are inherently crosscutting for simulation systems.

Thirdly, we have shown that, using a high level programming language that does not offer a graphical user interface, imposes many constraints on simulation tools users. Such concerns can be overcome by enforcement aspects smoothly.

Finally, we demonstrated that when working on legacy code for refactoring purposes, using Java Reflection and aspects hand in hand might prove to be very convenient and less costly. Thus it can be adopted as an idiom.

We leave it as a future work to explore the usability and efficacy of such an idiom

6. Acknowledgements

The authors would like to thank to Asst. Prof. Dr. Bedir Tekinerdogan, who has organized the "Third Turkish Aspect-Oriented Software Development Workshop", for his endless motivation in teaching us Aspect Oriented Programming in a graduate course at Bilkent University and generously sharing his comments with us to support writing this paper.

7. References

[1] R.E. Shannon, Systems Simulation – The Art and Science, Prentice Hall, Englewood Cliffs, 1975.

[2] B. Page, W Kreutzer, The Java Simulation Handbook – Simulating Discrete Event Systems with UML and Java, Shaker Verlag, Aachen, 2005.

[3] A. Boss, "Basic Event Graph Modeling", Simulation News Europe, ARGESIM and ASIM, Germany/Austria, April 2001.

[4] A. Boss, "Discrete Event Programming with Simkit", Simulation News Europe, ARGESIM and ASIM, Germany/Austria, November 2001.

[5] R. Miles, AspectJ Cookbook, O"Reilly, Cambridge, December 2005.