ASPECTJ CODE IMPLEMENTATIONS-CROSS CUTTING CONCERNS

By Rami Natsheh

1. THE FIRST PART - Examining a Cross Cutting Concern

The cross-cutting concern that is being considered in this study is the synchronization concern among different concurrent objects or classes. It is common knowledge that concurrency among multiple threads or multiple objects that access the same resource or resources usually need some kind of mechanism to control this kind of access and regulate it (Milicia & Sassone, 2004) to achieve the expected behavior. On the other hand using inheritance in concurrent object oriented systems might introduce some problems in what is known as the inheritance anomaly (Milicia & Sassone, 2004).

Synchronization code is usually spread across multiple files or classes in large systems where multiple threads or objects are trying to get access to certain resource at the same time. It is quite uncommon for each thread or object to track the state of each resource on its own, so the state related information is contained within the resource itself. This eventually causes the synchronization code to exist in every shared resource across multiple classes or objects, hence introducing cross cutting concerns in such systems.

In order to synchronize access to different resources such as objects, methods …etc; the most used framework to handle such scenarios is to use the locking mechanism. The synchronized methods that are being accessed will be locked specifically to the current thread, until the thread releases the lock. To enhance the communication between Threads, Notify(), Notifyall() and Wait() commands are used, for example in Java (Milicia & Sassone, 2004). Even though this is a very efficient technique, it still keeps the Synchronization related code widespread across multiple files causing code tangling and making it difficult to be inherited, reused or maintained.

Utilizing Aspects in such systems provides a possible solution that will keep the code contained within one file. In order to demonstrate the benefits of using Aspects, a simple example will be used to show how Aspects can in fact improve the process of building systems that use synchronized threads or objects.

As seen in Figure 1, the program is designed to move 5 Robots randomly on a map. To guarantee that no Robot steps over any of the other Robots a synchronization mechanism is used. Two different approaches are discussed here. The first example “Robots” demonstrates the way the program was used to synchronize the movement without using Aspects; this is displayed in Figure 1. Detailed description of how to operate the programs is mentioned later in Part 1.

In order to synchronize the movement of the Robots, a method in the Map class, MoveRobotSynhcronized(), was added. This method is a synchronized method that allows one robot to access it at one time, and it returns a new random location that is one step around the current location of that accessing Robot. The original MoveRobotUnSynchronized() method was replaced with the new one. Detailed description of the two methods can be seen in the Map class in the Robot example that was attached to this report. Figure 2 shows a snippet of the code for the synchronized method.

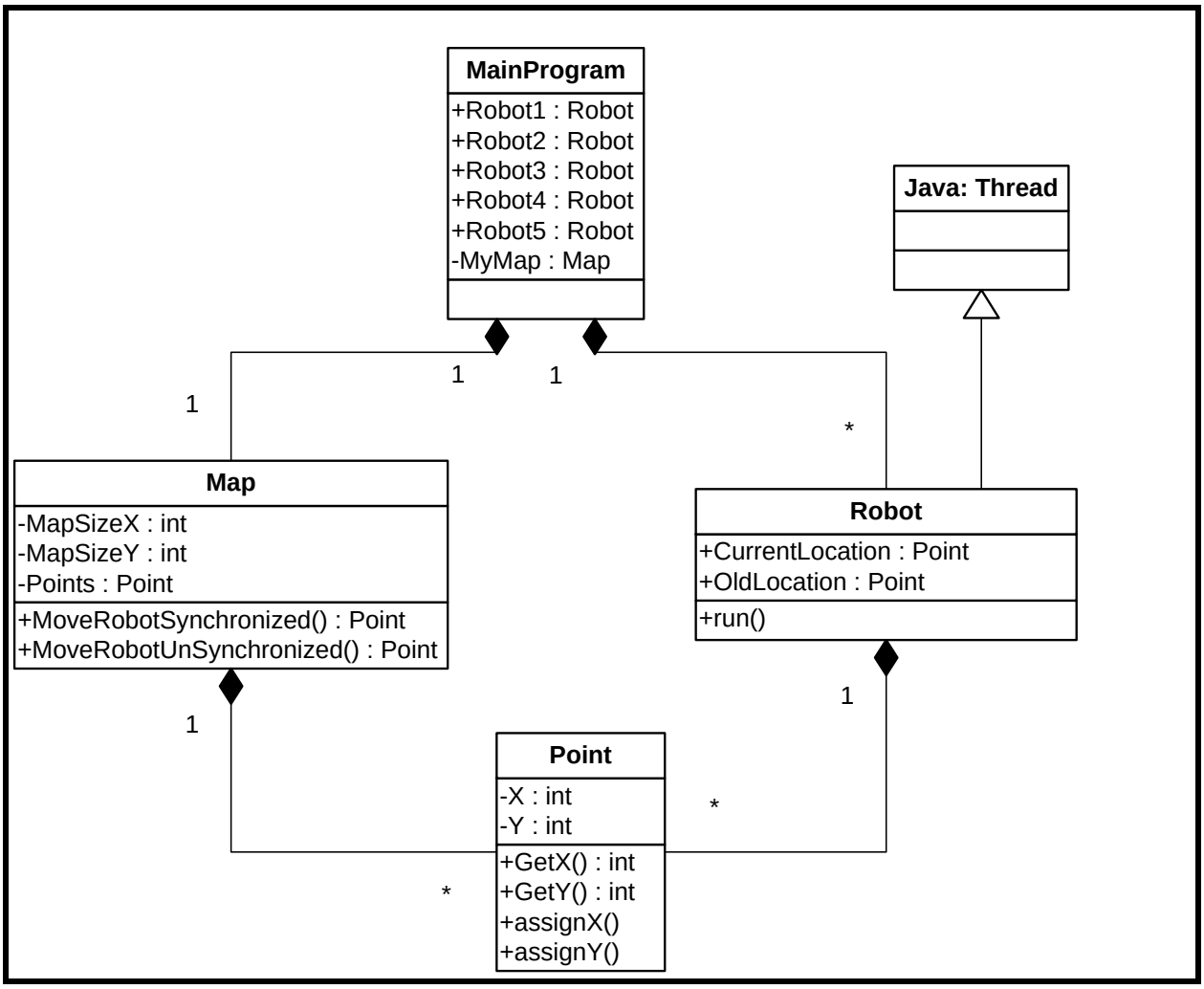

Figure 1 UML Diagram of the Robot example

As can be seen from the code, in order to guarantee that each point can be occupied by only one Robot at the same time, the original Map class was altered. In such a small program, such implementation is possible and will not be very costly. As the number of classes that require synchronization increases or when the internal implementation of the classes are not open to change –such as imported libraries- this approach becomes a much harder exercise and will increase the cost of integration (Pressman, 2004), since multiple classes will be affected.

```
public synchronized Point MoveRobotSynchronized(Point
CurrentLocation, Point OldLocation)
      {
            boolean status = true;
            while(status)
            {
                  // Find the new Coordinates
                  int X = GetNextXCoordinates(CurrentLocation);
                  int Y = GetNextYCoordinates(CurrentLocation);

                  // if the Point is used find another random location
                  // keep trying until it is successful
                  if (Points[X][Y].Locked)
                  {
                        status = true;
                  }
                  else
                  {
                        Points[X][Y].Locked = true;

      Points[OldLocation.getX()][OldLocation.getY()].Locked = false;
                        CurrentLocation.assignX(X);
                        CurrentLocation.assignY(Y);
                        status = false;

                  }
            }
            return CurrentLocation;
}
```

Figure 2 the Synchronized method in the Robot example without Aspects

On the other hand, if a new requirement is needed to change some of the implementation details, for example to move the robots by four steps instead of one step at a time, it will become difficult to propagate the changes to every single class in a bigger system. Also the original methods or classes need to be changed resulting in an increase in costs and the introduction of unexpected errors.

Using Aspects for such a small program is not going to save a lot of testing time, or improve performance. On the contrary, it was not the easy approach. But, if this program is to be modified and maintained and expanded, Aspects will be the better option (Noda & Kishi, 2001). Figure 3 shows the UML diagram of how Aspects was introduced in such an example.

In Figure 3, the newly defined Aspect, SynchronizeRobots is introduced. The major pointcut is defined to point the original MoveRobotUnSynchronized method. Also a new method is introduced within the Aspect, MoveSynchedRobots that will make sure that the robot moves to an unoccupied location. Figure 4 shows a snippet of the SynchronizeRobots Aspect for the pointcut and the new method.

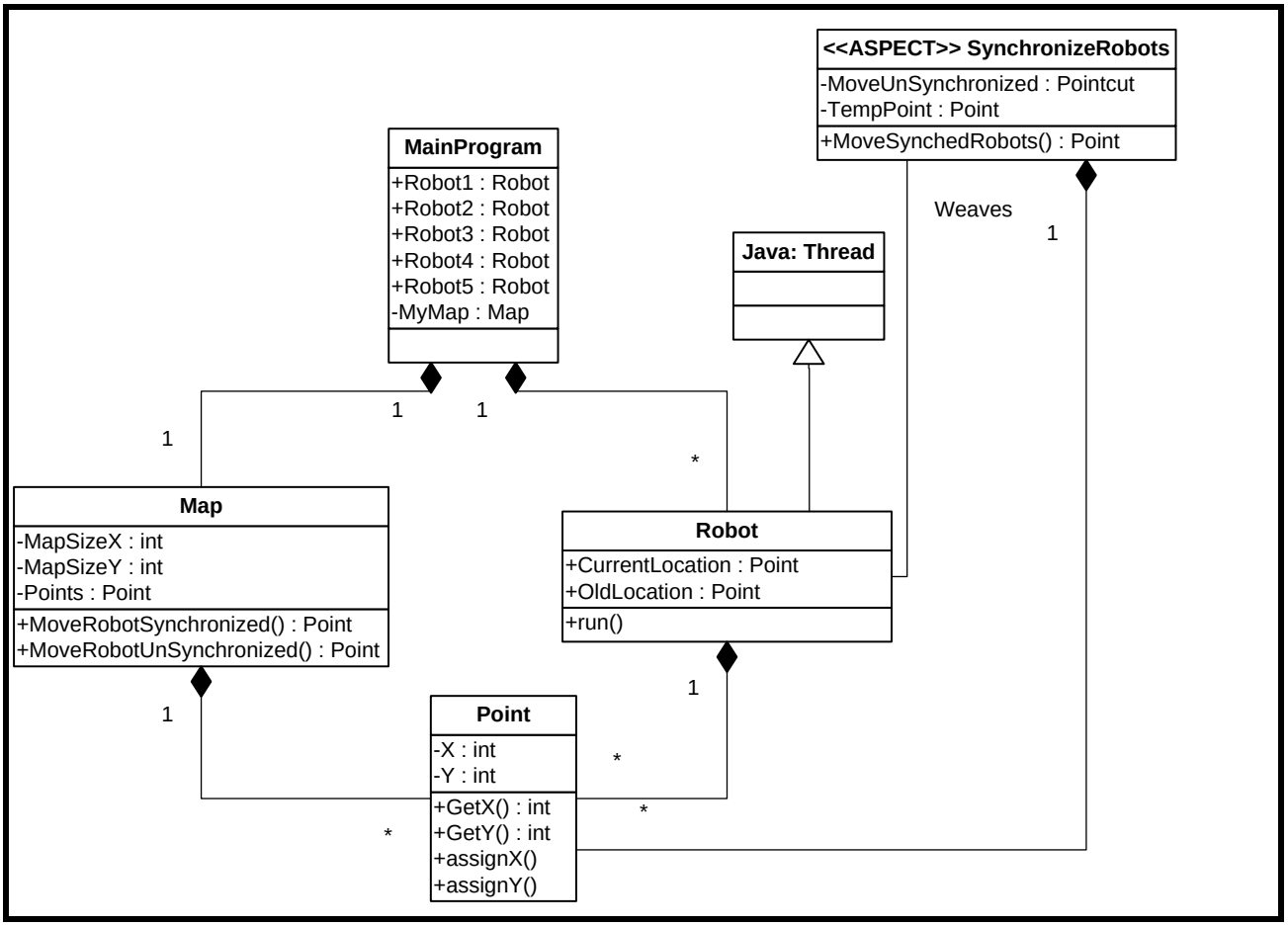

Figure 3 UML Diagram of Robot_Aspect example

```
      pointcut MoveUnSynchronized(Point CurrentLocation):
      call (Point MoveRobot*(Point)) && args(CurrentLocation);

      Point around(Point CurrentLocation):
MoveUnSynchronized(CurrentLocation) {

            TempPoint = MoveSynchedRobots(CurrentLocation);
            return TempPoint;
}

      public synchronized Point MoveSynchedRobots (Point CurrentLocation)
      {
            Point OldLocation = new Point(0,0);
            OldLocation = CurrentLocation;
            boolean status = true;
            while(status)
            {
                  // Find the new Coordinates
                  int X =
MainProgram.MyMap.GetNextXCoordinates(CurrentLocation);
                  int Y =
MainProgram.MyMap.GetNextYCoordinates(CurrentLocation);

                  // if the Point is used find another random location
                  // keep trying until it is successful
                  if (MainProgram.MyMap.Points[X][Y].Locked)
                  {
                        status = true;
                  }
                  else
                  {
                        MainProgram.MyMap.Points[X][Y].Locked = true;

      MainProgram.MyMap.Points[OldLocation.getX()][OldLocation.getY()].Lo
cked = false;
                        CurrentLocation.assignX(X);
                        CurrentLocation.assignY(Y);
                        status = false;
                  }
            }
            return CurrentLocation;
```

Figure 4 Using an Aspect in the Robot_Aspect example

Introducing a Synchronization Aspect in this program will make it easier to change the way the Robots move by adding several other lines of code without the need to change the original method or class. For example, if the previously mentioned requirement that the Robot should move four steps at a time instead of one, then the changes shown in Figure 5 will implement the requirement with very minimal changes in one file, that is the SynchronizeRobots Aspect.

```
      Point around(Point CurrentLocation):
MoveUnSynchronized(CurrentLocation) {
                        //TempPoint = MoveSynchedRobots(CurrentLocation);
            TempPoint = MoveSynchedRobotFourSteps(CurrentLocation, 4);
            return TempPoint;


      public synchronized Point MoveSynchedRobotsFourSteps (Point
CurrentLocation, int Steps)
      {
            Point OldLocation = new Point(0,0);
            OldLocation = CurrentLocation;

            for (int temp =0; temp < Steps; temp++)
            {
                  boolean status = true;
                  while(status)
                  {
                        // Find the new Coordinates
                        int X =
MainProgram.MyMap.GetNextXCoordinates(CurrentLocation);
                        int Y =
MainProgram.MyMap.GetNextYCoordinates(CurrentLocation);

                        if (MainProgram.MyMap.Points[X][Y].Locked)
                        {
                              status = true;
                        }
                        else
                        {
                              MainProgram.MyMap.Points[X][Y].Locked = true;

      MainProgram.MyMap.Points[OldLocation.getX()][OldLocation.getY()].Locked
= false;
                              CurrentLocation.assignX(X);
                              CurrentLocation.assignY(Y);
                              status = false;
                        }
```

Figure 5 Using an Aspect to introduce changes in the Robot_Aspect example

As can be seen from Figure 5, the original method MoveSynchedRobots was changed to MoveSynchedRobotsFourSteps. The changes will reflect in all threads and all methods that access the move function in the program, hence it is much easier to integrate, test and maintain.

Overall the whole Aspectization of the Synchronization is actually very useful in large programs that span large number of classes that contain many concurrent threads that do need the Synchronization mechanism when accessing shared resources. The benefits were demonstrated using the moving Robots example; where with Aspects the programmer can easily contain the whole cross cutting concern of synchronization in one aspect, can easily maintain the code and can easily add new changes or requirements by just changing one aspect rather than dealing with multiple classes that contain part of the Synchronization code.

Other researchers have found similar results in terms of how Aspects can be beneficial in such systems. In (Milicia & Sassone, 2004), it was found that Synchronization and Inheritance can be handled in a better way with Aspects compared to regular OO techniques. This was quite obvious when considering the system for expansion.

HOW TO OPERATE THE PROGRAMS

The two programs that were mentioned earlier were coded using Java with AspectJ support using Eclipse SDK 3.4.0. The two programs utilize the “org.eclipse.swt” library that is attached as a “swt-3.4-win32-win32-x86.zip” file and can be imported as an existing project into eclipse. The other two programs are “Robots.zip” and “Robots_Aspects.zip” which also can be imported as existing programs. Both programs can be run from the MainProgram.java using the “Ctrl+F11” keys.

2. THE SECOND PART - Redesign a design pattern with aspects in mind

The pattern that is under consideration in this report is the Observer pattern (Bishop, 2007). The Observer pattern consists of two classes; the Subject that changes its state occasionally and the Observer that might choose to know the state of the Subject and would attach itself to the Subject in order to be notified of any changes in the state of the Subject. Figure 6 shows the UML diagram for the Observer pattern.

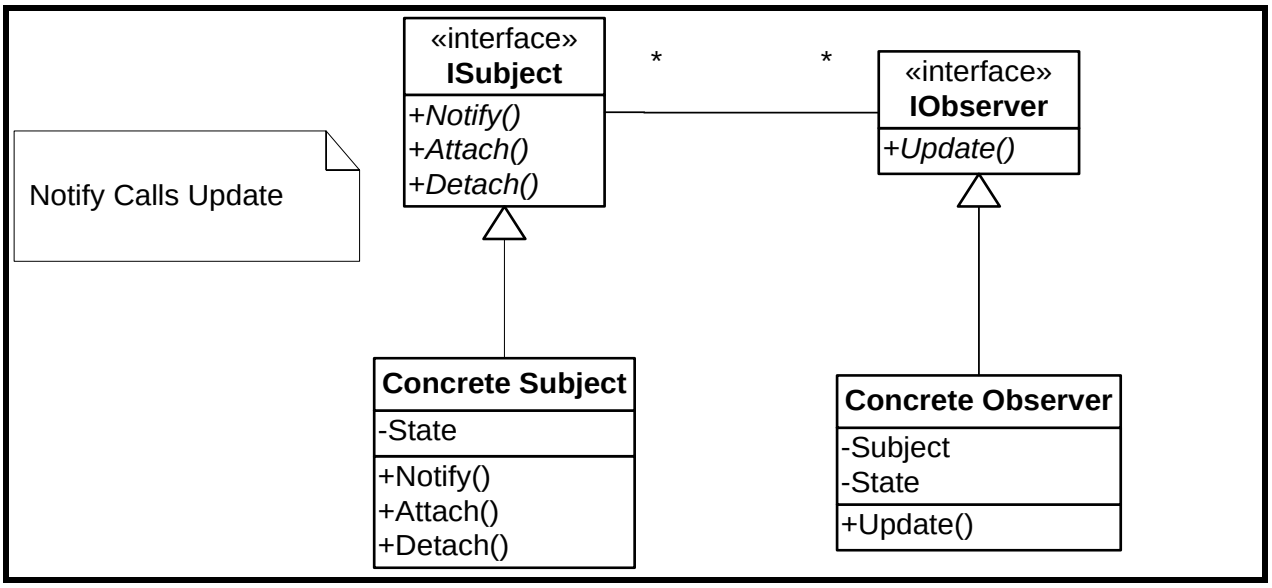

Figure 6 Observer Pattern

Implementing the Observer pattern in its known form that is shown above, will cause certain modifications to the concrete classes and will alter some behaviors leading to more coupling and less cohesion. For example the Concrete Subject and the Concrete Observer should implement new methods such as Notify(), Attach() and Detach() for the concrete Subject and the Update() method and Subject attribute for the concrete Observer. This unfortunately results in hard coding the changes in the system leading to difficulty in implementing changes in an evolving system (Noda & Kishi, 2001). Aspect Object Programming (AOP) on the other hand, can help in separating the system’s design patterns and in implementing them as a single unit of abstraction (Noda N., Kishi T. (2001). Figure 7 shows the UML diagram for the first suggested redesign of the Observer pattern with aspects in mind.

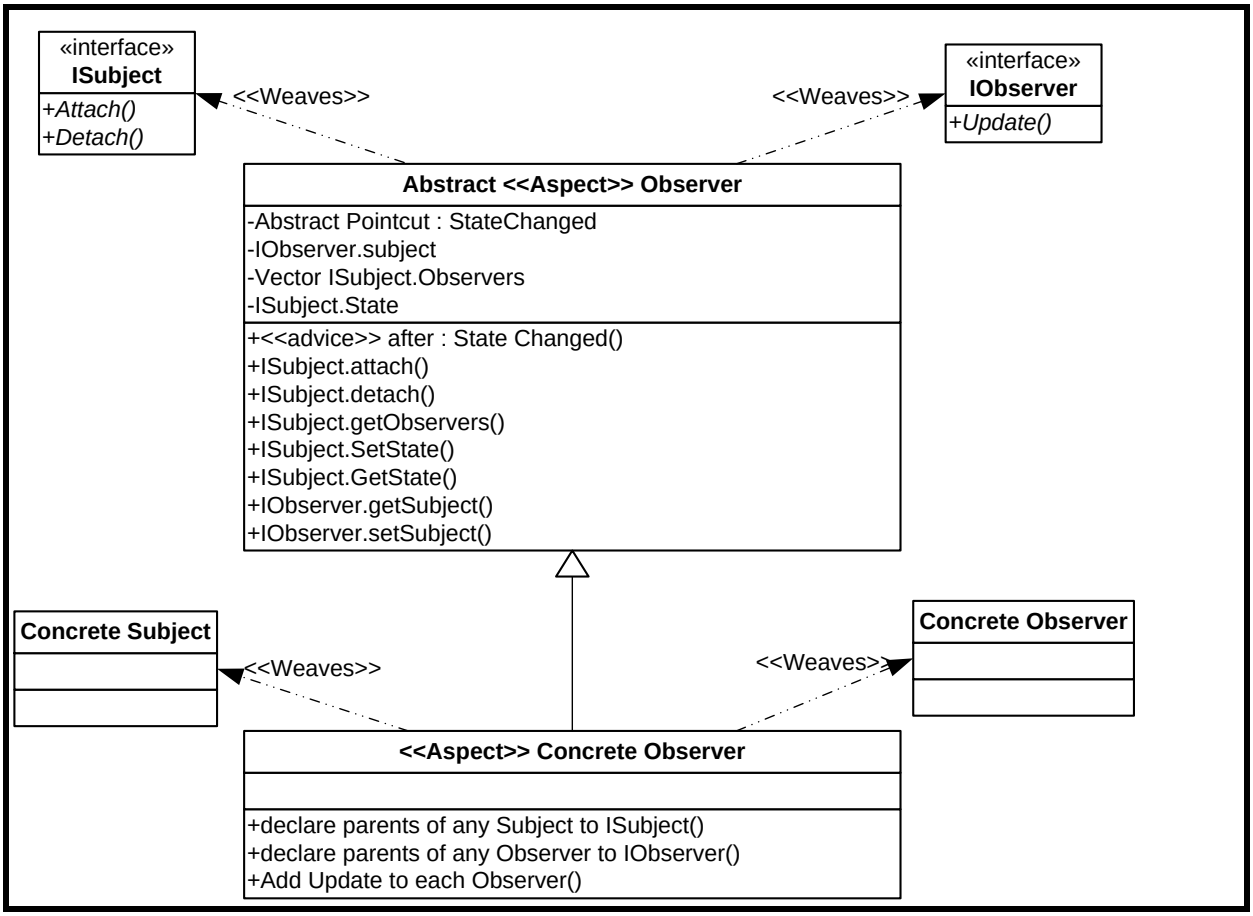

Figure 7 First suggested Observer Pattern using Aspects

As seen from Figure 7, a new Abstract Aspect “Observer” was defined which is inherited by a Concrete Aspect. Both of them weave certain changes in the interfaces and the Concrete Classes to achieve the required functionality of the pattern. The Abstract Aspect defines a StateChanged pointcut that each concrete aspect has to define. Also some attributes are defined for the ISubject and IObserver interfaces, such as the Subject in IObserver and the Vector in ISubject that will contain the attached Observers. Notice that the Notify method is removed form the ISubject interface since the notification functionality is being handled by the aspect in this pattern using the after advice.

The concrete aspect weaves into the Concrete Subject and Concrete Observer; it will add the required functionality of declaring the parents of the Subject to be ISubject and the Observer to be IObserver, and at the same time it will be adding the update mechanism to the Concrete Observers.

As seen from the new aspect oriented pattern defined, the concrete Subject and Observers are quite independent of the overall pattern. The pattern became reusable compared to the regular Object Oriented (OO) pattern definitions. There is no need to hard code the majority of the required methods and attributes in the aspect oriented pattern compared to the OO pattern. For example, there is no need to hard code the Attach(), Detach(), Notify(), and Update() methods in any of the concrete Subject and Observer. Also there is no need to declare any relations between the interfaces and the concrete classes within the concrete classes themselves. All of the above is done within the Aspect and the Concrete Aspect where most of the code related to the design pattern is localized to one file that is the Aspect.

In order to demonstrate the new first suggested aspectized pattern, a program (ObserverExample) was written to demonstrate how this new design pattern can be useful. Figure 8 shows the UML diagram of this program. The program contains two interfaces, ISubject and IObserver, one Abstract Aspect, AbstractObserverPattern, one Concrete Aspect, ObserverPattern, one Concrete Subject, and two Concrete Observers, Observer1 and Observer2.

The Subject class is actually a thread, the object created from that class in the MainProgram, Subject1 will run and change its state randomly. Two objects (Obs1, Obs2) based on Observer1 and Observer2 are created in the MainProgram and are attached to Subject1 using the attach() method in the MainProgram class. After the thread is started the state of Subject1 is changed randomly based on a random generator in the subject class to simulate a real scenario. When the State changes the Observers are notified and each will print its state as a reflection of the Subject state being changed. Using the defined pattern, the abstract and concrete aspects will weave the required code to make sure that the functionality required is implemented.

When examining the program the benefits of using the new pattern is noticed. For example, the Subject class does not contain any code that is relevant to the design pattern as well as the Observer class. Also the Subject class and the Observers classes can inherit any other class without taking into consideration its effects on the chosen design pattern.

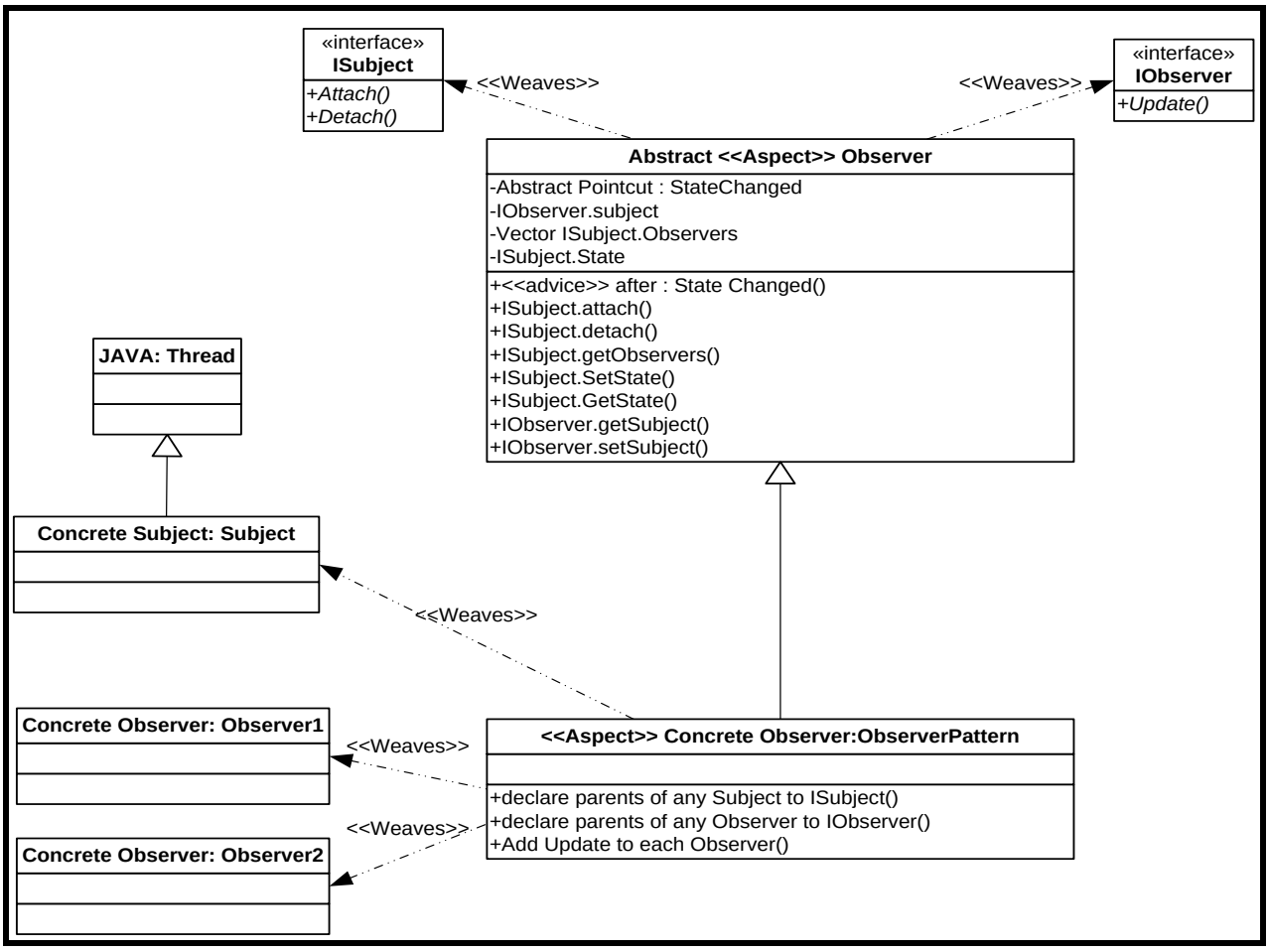

Figure 8 UML Diagram of the example program for the first Aspectized Observer pattern

One important advantage of such an approach is the fact that the application core becomes more reusable once it is independent from the design pattern. This eventually leads to better reusability and maintainability. For example, a different pattern can be implemented on the core application if the evolving systems need to do this (Noda N., Kishi T. (2001).

After the implementation of this system, the author realized a major disadvantage of such approach; that is the pattern is not easily expandable to more than one subject. If the user needs to add another Subject to the system that interacts with the same Observers the Abstract Aspect and the concrete Aspect need to be modified in a way that modifies both Subjects that might complicate the process. That is what triggered the author to look into other options.

Another implementation was considered when redesigning the pattern that is shown in Figure 9. In this pattern two aspects were defined, one for adding the functionality of updating observers, the other is to attach or detach the observers.

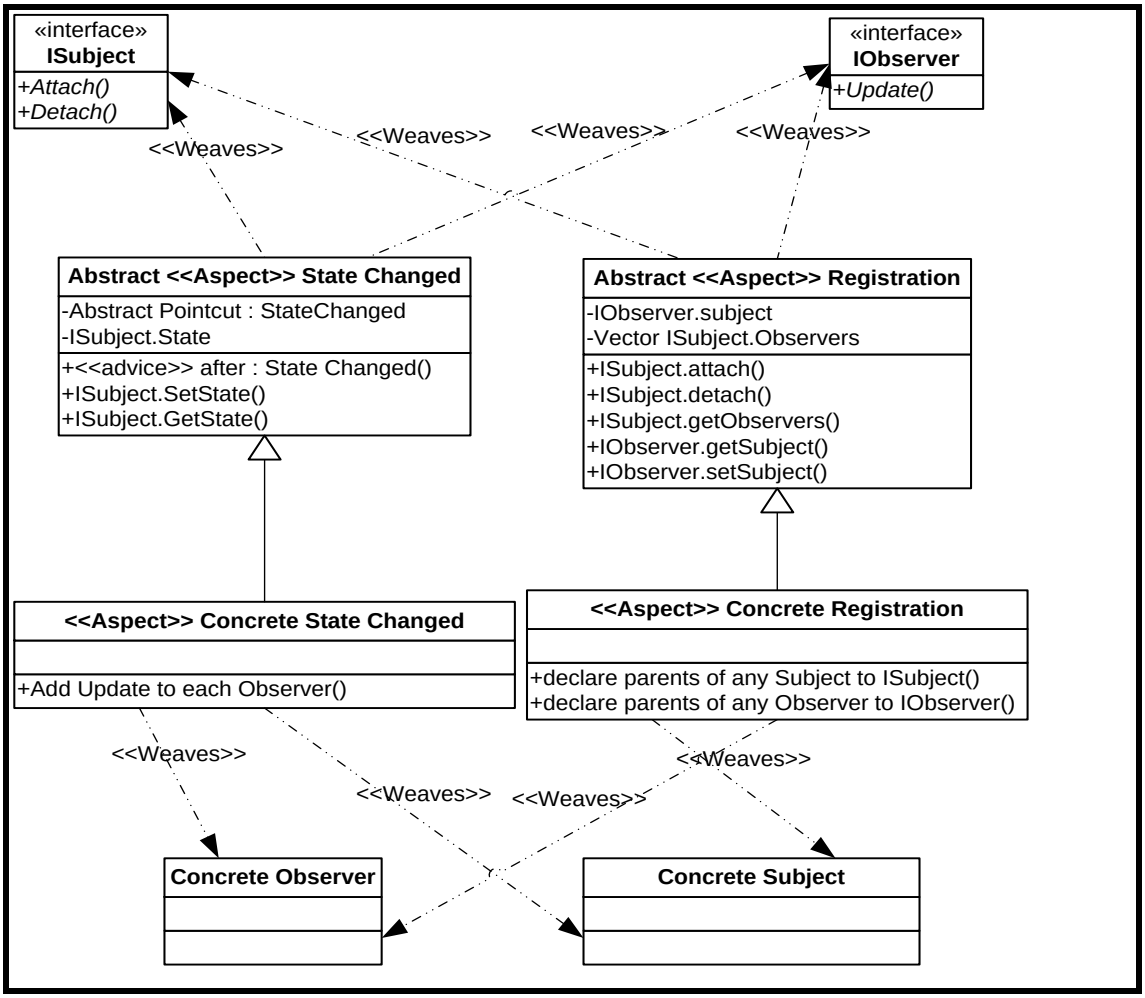

Figure 9 UML Diagram of another Aspectized Observer pattern

Separating the two independent processes for the Registration and the State changes might be useful, but in the author's point of view, it just added more complications to the design. Both aspects actually weave into both concrete classes leading to a more complicated code, also the previously mentioned flaw related to adding more Subjects is actually exaggerated here since its code will be distributed among the two Aspects. So this design was discarded.

The final redesign that the author considered for the Observer pattern is shown in Figure 10. In this redesign, it is clear that adding another Subject is easily done by defining another concrete Aspect for the Subject and its code will be separated. This allows for systems to evolve easily and probably will allow for a better reusability, in addition to the other benefits of separating the core application from the design pattern. This final pattern was applied and used to change the original program that was described earlier. The program is attached as “ObserverExample2”.

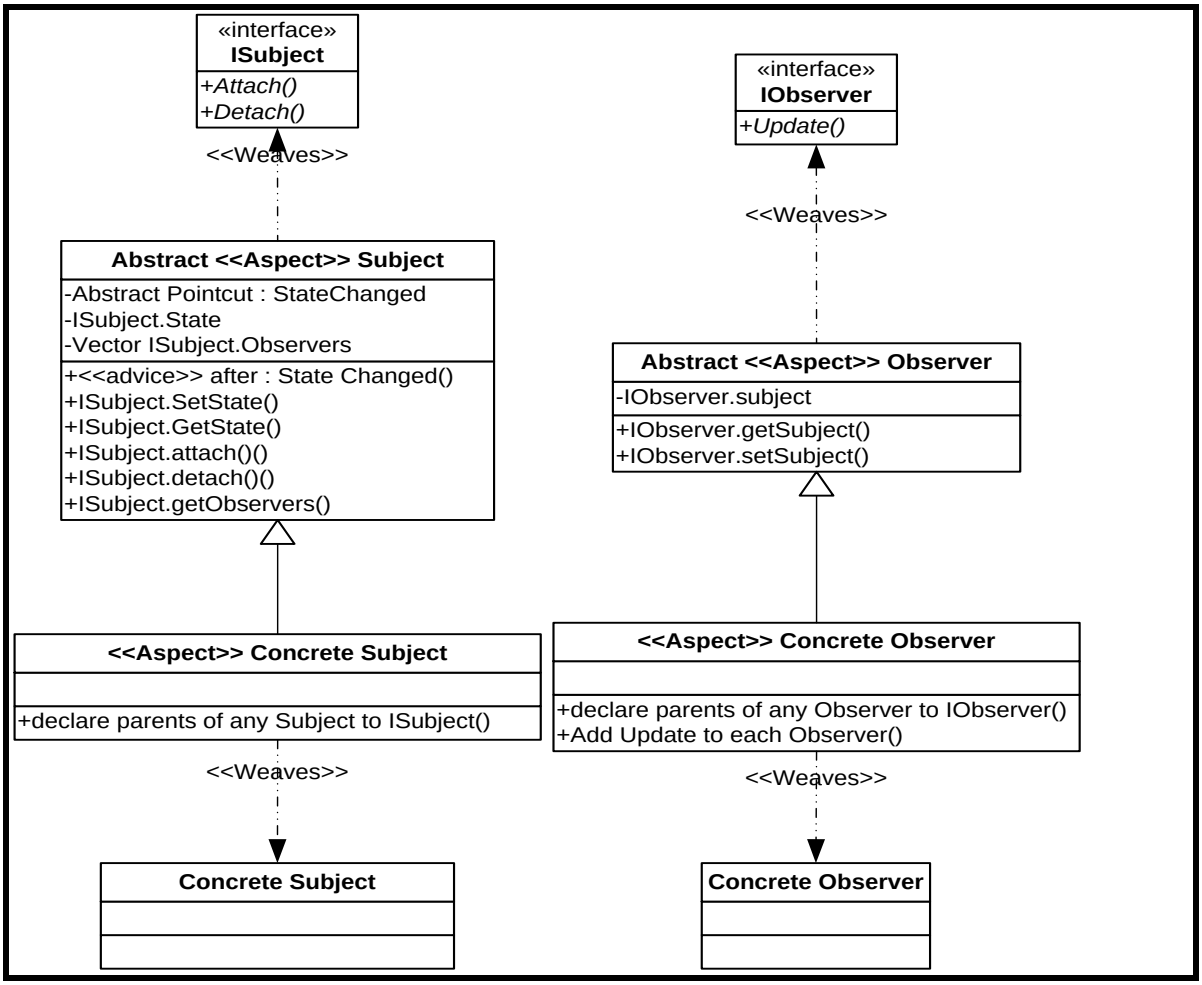

Figure 10 Final UML Diagram of an Aspectized Observer pattern

Considering aspects for the redesign of the Design patterns was addressed by several researchers. The design that is shown in Figure 7 is similar in structure to the one mentioned in (Piveta & Zancanella, 2003). On the other hand the design shown in Figure 10 is similar to the one mentioned in (Noda & Kishi, 2001).

After applying the new design pattern in Figure 10, it is obvious that the there is less interaction between the Subject and Observer Classes. That is the Notify() method in the Subject is not calling the Update() method in the Observer. This functionality is handled by the Aspect within this pattern. The Aspect will weave these changes during compilation time. Due to this fact it is noticed that the Cohesion within the Subject and the Observer classes has increased and that less coupling is occurring during to removing the tangled code from the class.

In (Garcia et al., 2005), several measures were used to evaluate the benefits of redesigning the design patterns using Aspects. One of the measures was separation of concerns, the other measures were Coupling, then Cohesion and finally Size. According to the study, and specifically related to the aspectized Observer pattern, it was found that the AOP approach provides better results in terms of the previously mentioned measures resulting in less coupling, more cohesion, less size and better separation of concerns.

Some difficulties were faced during the course of writing the programs and making sure that they are working. Using the Java editor caused some of the errors to be declared by Eclipse even though the aspects were defined correctly. One example was in the main program when attaching one of the observers to the subject, the Eclipse SDK did not recognize the Subject1.attach() method as a valid method. When the MainProgram was opened using the aspect editor the program compiled correctly. Also some intermittent issues were seen with the compiler, sometimes with the same code it declares errors and then after closing the projects and opening it again the issue is resolved.

HOW TO OPERATE THE PROGRAMS

The two programs that were mentioned earlier were coded using Java with AspectJ support using Eclipse SDK 3.4.0.The two programs are “ObserverExample.zip”, which was coded using the pattern mentioned in Figure 7, and “ObserverExample2.zip” which also can be imported as an existing program and was coded using the pattern mentioned in Figure 10. Both programs can be run from the MainProgram.java using the “Ctrl+F11” keys.

REFERENCES

Noda N., Kishi T. (2001). Implementing Design Patterns Using Advanced Separation of Concerns. In *Proceedings of OOPSLA 2001, Workshop on Advanced Separation of Concerns in Object-Oriented Systems*. Tampa Bay, FL

Piveta, E., Zancanella, L. (2003). Observer Pattern usingAspect-Oriented Programming. In *Proceedings of the 3rd Latin American Conferenceon Pattern Languages of Programming.* Porto de Galinhas, PE, Brazil.

Hannemann, J. , Kiczales, G.(2002). Design pattern implementation in Java and aspect. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* Seattle, Washington, USA

Milicia, G., Sassone, V. (2004). The inheritance anomaly: ten years after. In *SAC '04 Proceedings of the 2004 ACM Symposium on Applied Computing* (pp. 1267-1274). New York, NY, USA.

Holmes, D., Noble, J., Potter, J.(1997). Aspects of Synchronization. In *Proceedings of the Technology of Object-Oriented Languages and Systems - Tools-25*(pp. 24-28)

Pressman, Roger S. (2004). Software Engineering, A Practitioner's approach. NewYork, NY: McGraw-Hill .

Bishop, Judith (2007). C# 3.0 design patterns, First edition. North Sebastopol, CA: O'Reilly

Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., Von Staa, A.(2005). Modularizing design patterns with aspects: a quantitative study. In *Proceedings of the 4th international conference on Aspect-oriented software development* (pp. 3-14). Chicago, Illinois