ArchJava Evaluation Report

Lin Liao liaolin@cs.washington.edu
Ke Zheng (Colin) kzheng@cs.washington.edu

Introduction

With the software system becomes more and more complex, software architecture tools become more and more important. However, most such tools decouple the implementation code from the architecture design, allowing inconsistencies. ArchJava is a backwards-compatible extension to Java that integrates architecture expressions seamlessly with Java implementation. ArchJava supports a flexible object-oriented programming style, allowing data sharing and supporting dynamic architectures where components are created and connected at run time. The unique feature of ArchJava is a type system that guarantees communication integrity between architecture and implementation, even in the presence of shared objects and run-time architecture configuration [ACN01] [ACN02].

ArchJava has been applied to several applications like Taprats, an Islamic tiledesign Application, and Aphyds, a circuit-design application. All these case studies on applications with certain degrees of complexity have suggested that ArchJava can express architectural structure effectively within an implementation, and thus aid in program understanding and software evolution [ACN01] [ACN02].

As a new tool, more case study is needed to exploit the whole expressiveness of ArchJava, but most of the times, the complexity of a program is not only measured in lines, but also in its intrinsic structure, the way objects interact with each other. And we think the latter is more important for architecture tools, including ArchJava. All these are exactly what design patterns provide and work for. (In fact, as we dig further in design patterns, we found it has great similarity in software architecture). So it would be an interesting idea to test those most commonly used patterns with ArchJava. Meanwhile, by testing the effectiveness and expressiveness of ArchJava, we have deepened our understanding in both software architecture and programming language.

Design patterns describe the design of objects as well as the communication between objects to solve a general problem in a particular context. It is the design of simple, but elegant, methods of communication that makes many design patterns so important.
So far, people have summarized a lot of design patterns from experience, including over 20 classical patterns, each of which has several known applications and have great impact on solving various problems. In [ERR+94], they are classified into three types: creational, structural and behavioral.

- **Creational patterns** are ones that create objects for you, rather than having you instantiate objects directly. This gives your program more flexibility in deciding which objects need to be created for a given case.
- **Structural patterns** help you compose groups of objects into larger structures, such as complex user interfaces or accounting data.
- **Behavioral patterns** help you define the communication between objects in your system and how the flow is controlled in a complex program.

As an attempt to test ArchJava in different design patterns, we select six common design patterns, two out of each type. Here is a list of design patterns we used. One point needs to be mentioned is that when we chose these patterns, we had no idea how ArchJava should support them. So our selection has no preference or prejudice against ArchJava.

Creational Patterns
- **Abstract Factory**
- **Singleton**

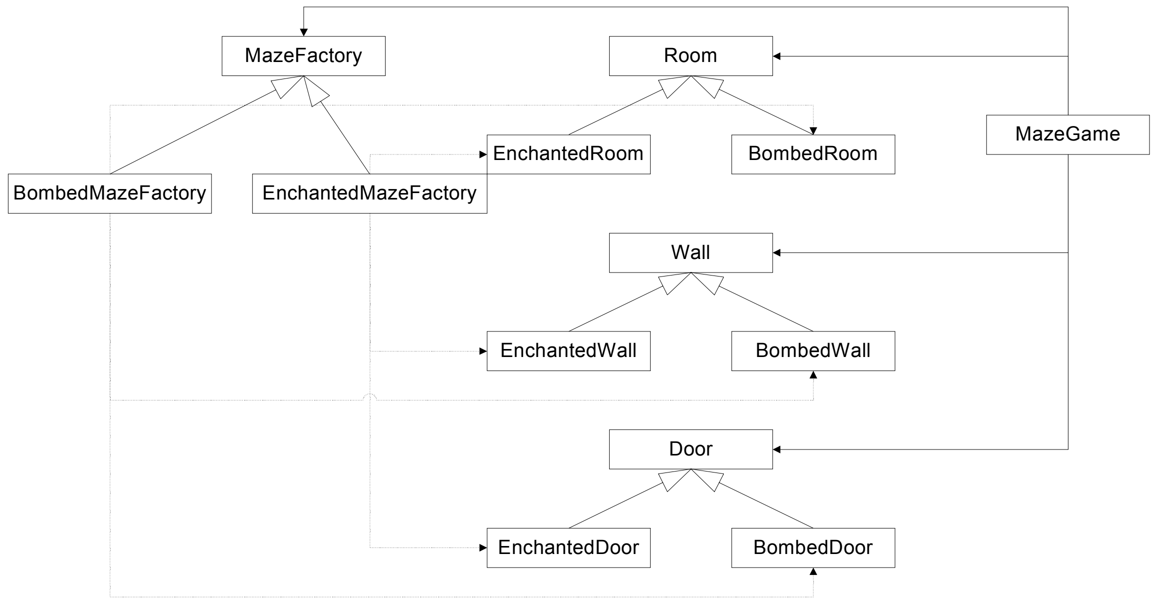
Structural Patterns
- **Composite**
- **Façade**

Behavioral Patterns
- **Observer**
- **Interpreter**

Pattern1: Abstract Factory

Factory pattern is often used in complex Java systems. For example, AWT uses Abstract Factory to generate all the required peer components for the specific platform being used. EJB (Enterprise Java Bean) uses Factory pattern to generate all the beans. The basic intention of such a pattern is to provide an interface for creating families of related or dependent objects without specifying their concrete classes. Besides this, the Abstract Factory is also a good example to test the support of polymorphism, which is one of the most significant characteristics of OO language.

We use a common example—building a maze for a computer game—to evaluate ArchJava. And we just focus on how mazes get created. We define a maze as a set of rooms. A room knows its neighbors; possible neighbors are another room, a wall, or a door to another room. The classes Room, Door, and Wall define the components of the maze used in all our examples. We have two sets of Room, Door and Wall------enchanted and bombed. And we provide two factory classes: EnchantedMazeFactory and BombedMazeFactory, both inherit from abstract factory MazeFactory. The following diagram shows the relationships between these classes:



We decided to change class MazeGame (the main program), MazeFactory, EnchantedMazeFactory, BombedMazeFactory to components. The ports and their relationships are shown in the following diagram.
To implement such a structure, ArchJava has to support component inheritance and port override.
Our first idea was using abstract port, such as

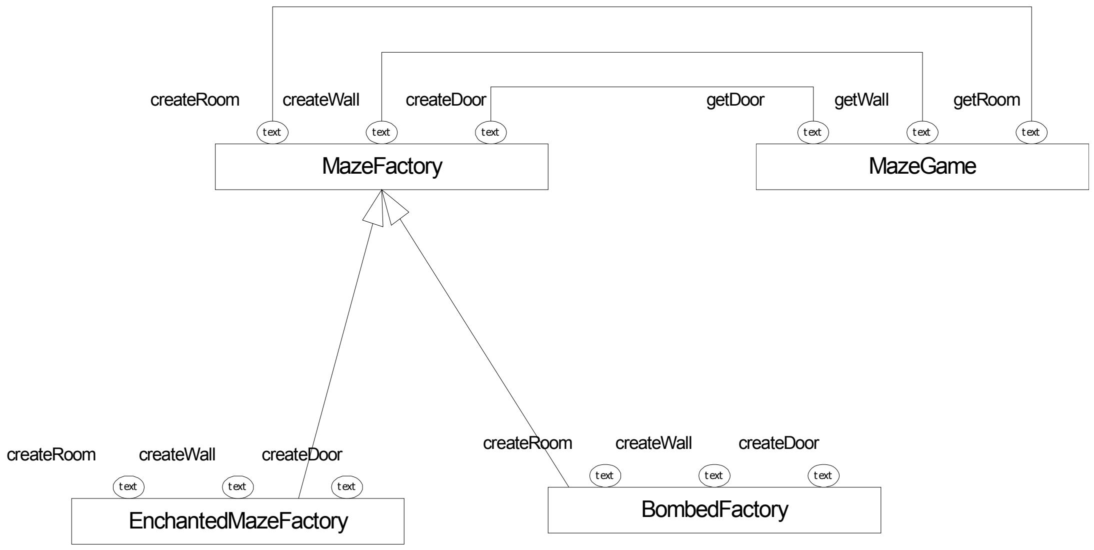
```
        public abstract port createRoom { …}
        ……
```

But there is a compilation error, meaning the current version doesn't support abstract port yet.

Then we override all the ports of MazeFactory in BombedMazeFactory and EnchantedMazeFactory, respectively. Then in MazeGame, we create a static connect as following:

```
public component class MazeGame {
        private final MazeFactory factory;
        connect factory.createRoom, this.getRoom;
        …
        public Maze createGame() {
                factory=new EnchantedMazeFactory();
                …
        }
        …
}
```

Although I expect the port is working as virtual like the methods in java, it isn't. Experiments showed that the ports are staticly bound. So we have to look for other alternatives.



The correct answer is connect expression. Our example shows that, using correct expression we can get late binding port and achieve polymorphism, then implement Abstract Factory. The change is in fact pretty straightforward when we were aware of this point.

```
public component class MazeGame {
        connect pattern MazeFactory.createRoom, MazeGame.getRoom;
        connect pattern MazeFactory.createWall, MazeGame.getWall;
        connect pattern MazeFactory.createDoor, MazeGame.getDoor;
        public port createRoom {
                requires Room makeRoom(int i);
        }
        public port createWall {
                requires Wall makeWall();
        }
        public port createDoor {
                requires Door makeDoor(Room r1, Room r2);
        }
```

```
    public Maze createGame() {
        final MazeFactory factory=new EnchantedMazeFactory();

        this.getRoom connection=connect(this.getRoom, factory.createRoom);

        Room r1 = connection.makeRoom(1);
        Room r2 = connection.makeRoom(2);
        ….
    }
    ….
}
```

This test shows that ArchJava can easily support Abstract Factory pattern. More important, it can support polymorphism and dynamically binding port. One insufficiency is the lack of abstract port, which will make the structure more clear and concise.

Pattern 2: Singleton

Singleton is also a very useful creational pattern. It ensures a class only has one instance, and provides a global point of access to it. For example, there should be only one file system and one window manager. And we only need one factory for a family of products. We still use the example of maze. This time we want to ensure only one MazeFactory instance is created.

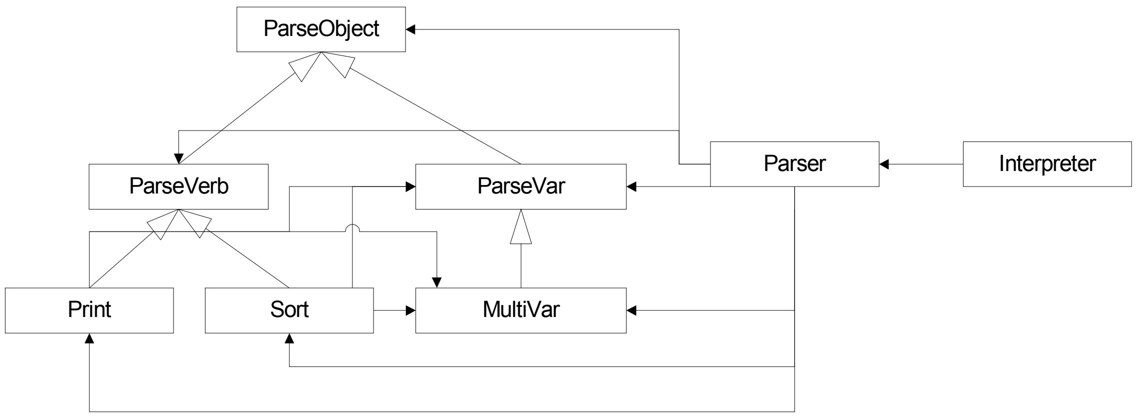
The most common way to implement Singleton is a static instance variable and a private constructor, shown below.

```
public abstract class MazeFactory {
    // The private reference to the one and only instance.
    private static MazeFactory uniqueInstance = null;

    // The MazeFactory constructor. Note that it is private!
    private MazeFactory() {}

    // Create the instance using the specified String name.
    public static MazeFactory instance(String name) {
        if(uniqueInstance == null)
            if (name.equals("enchanted"))
                uniqueInstance = new EnchantedMazeFactory();
            else if (name.equals("bombed"))
                uniqueInstance = new BombedMazeFactory();
        return uniqueInstance;
    }
    ….
}
```

Then we want to change MazeFactory to a component which provides a port to export the instance reference, say getInstance. But soon we recognize it seems impossible to implement such a pattern in ArchJava. The reason is in ArchJava, all the connects are constructed after the corresponding concrete objects are created. So we have to create the instance of MazeFactory before we can get the port getInstance. But how could we create the instance without getting the port? Since there is no concept of static port in ArchJava, we don't know how to implement such port getInstance so far. This doesn't mean we cannot use Singleton in ArchJava project. We can still change MazeFactory to component without package the static method instance() in a port.

Pattern 3: Interpreter

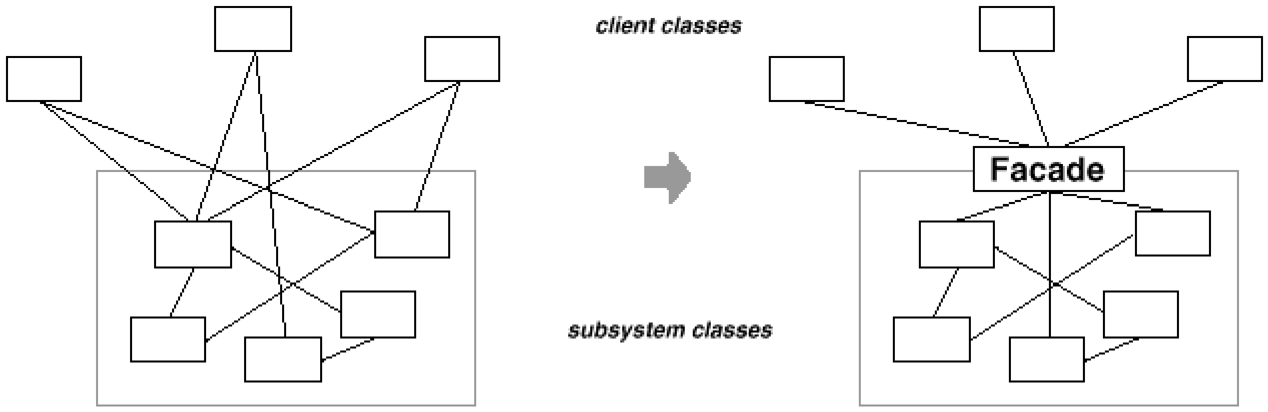
Many systems provide a command language to interact with users, such as macro, SQL, etc. The Interpreter pattern describes how to define a grammar for simple languages, represent sentences in the language, and interpret these sentences.
Our example is a simplified report generator. It support a very simple query grammar such as “print lname frname club time sortby club, time”. It supports 2 verbs as print and sortby, and 5 column names frname, lname, age, club and time. Using Interpreter pattern, we have to create object for each terminal and non-terminal. This makes the system fairly complex. The simplified diagram of structure is shown below.



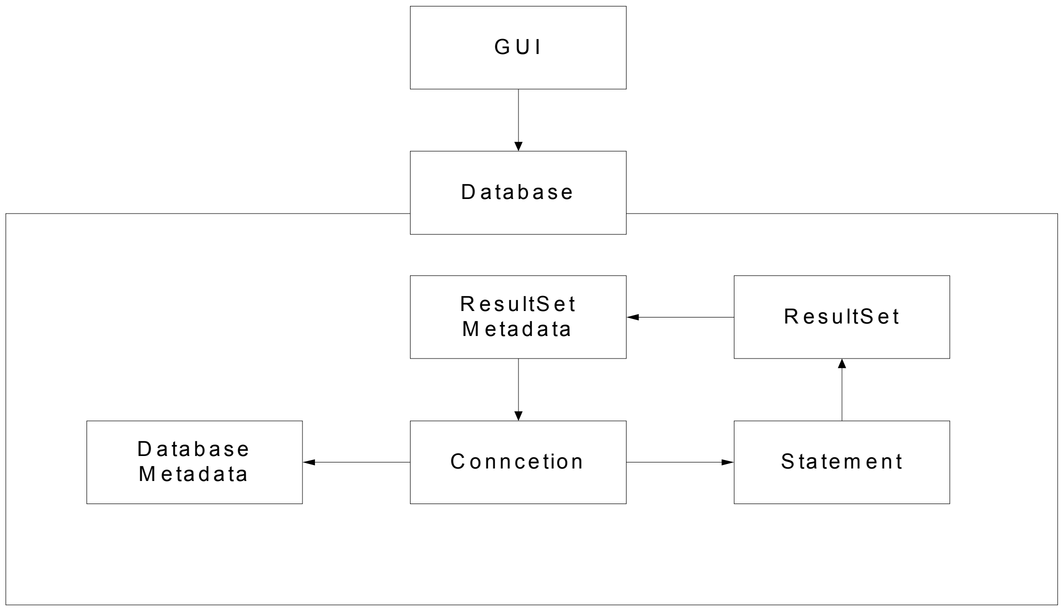
The key object is Parser, which implements the interpretation algorithm using a stack. A lot of communications exist between Parser and other objects and among ParseObject’s descendents.
We tried to express such as a structure in ArchJava. We spent at least 5 hours on such a conversion but made little progress. In the end we gave up. The main reason is that there exist a bunch of methods using components as parameters or return values. Since ArchJava doesn’t allowed components to be transmitted directly as parameter or return value, in order to support communication integrity, such a conversion becomes very difficult. Although we can use connect expression to make the communication work, as what we did in the Composite pattern, it seems a little complex and not natural. When component transmission happens frequently, as in the Interpreter pattern, the task becomes pretty tough.
So we believe for patterns as Interpreter and Iterator, which include many entity components that are frequently transmitted, it’s not easy to implement them in ArchJava.

Pattern 4: Façade

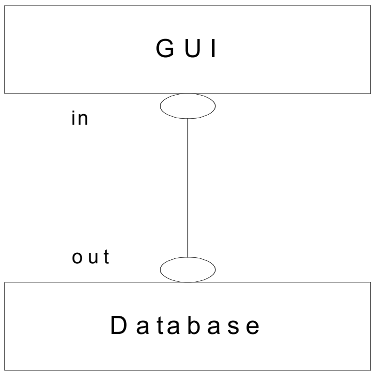
Frequently, as programs evolve and develop, they grow in complexity. As a result there may be a number of complicated subsystems, each of which has its own complex interface. The Façade pattern allows you to simplify this complexity by providing a simplified interface to these subsystems. The basic idea is shown in the figure.



Our evaluation is based on a JDBC subsystem, the structure of such a system is shown below.



Here Database work as a façade, it provides a simplified interface for JDBC operations. Using ArchJava to implement such a pattern is very straightforward: we only need to change Database and GUI as components and provide ports for their communications. The component diagram is shown below:



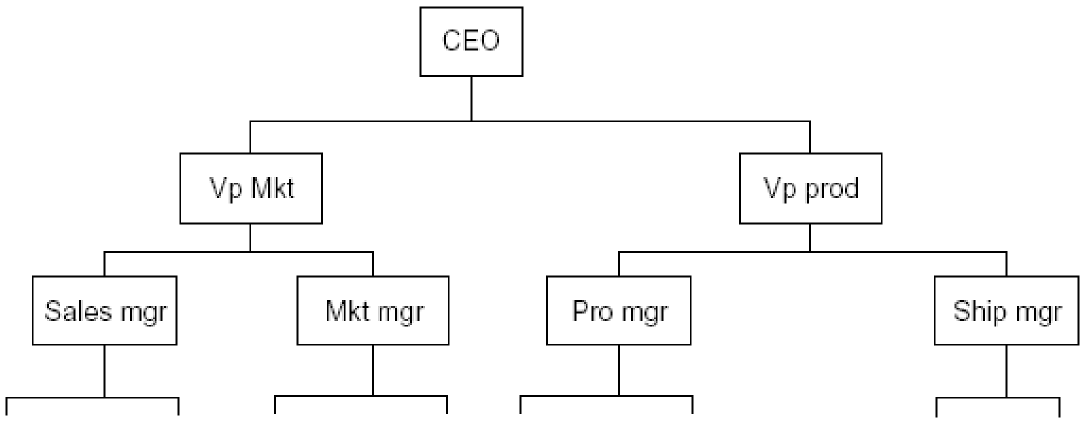
The port is designed as following:

```
public class Database
{
        public port out {
                provides String[] getTableNames();
                provides String[] getTableMetaData();
                provides String[] getColumnMetaData(String tablename);
                provides String[] getColumnNames(String table);
                provides String getColumnValue(String table, String columnName);
                provides String getNextValue(String columnName);
                provides resultSet Execute(String sql);
        }
        ….
}
public class GUI extends Frame  implements ActionListener, ItemListener
{
        public port in {
                requires String[] getTableNames();
                requires String[] getTableMetaData();
                requires String[] getColumnMetaData(String tablename);
                requires String[] getColumnNames(String table);
                requires String getColumnValue(String table, String columnName);
                requires String getNextValue(String columnName);
                requires resultSet Execute(String sql);
        }
        private final Database db;
        connect pattern Database.out,GUI.in;
        ….
}
```

As you can see, however complicated the subsystem is, the Façade Pattern uses a simplified interface to solve the problem. And in turn, this makes the ArchJava implementation easy for adding only required ports.

Pattern 5: Composite

Composite pattern is designed to accommodate component as individual object or collection of objects. It's usually used to build part-whole hierarchies or to construct data representations of trees. And it is a perfect example to illustrate one of the important principles in OO programming: Favor object composition over inheritance.
An easy example we used here is to set up a small company, so it will have a hierarchical structure like this:



Each of these company members receives a salary, and we could at any time ask for the cost of any employee to the company. We define the cost as the salary of that person and those of all his subordinates. Here is an ideal example for a composite:

- ➢ The cost of an individual employee is simply his salary (and benefits).
- ➢ The cost of an employee who heads a department is his salary plus those of all his subordinates.

We would like a single interface that will produce the salary totals correctly whether the employee has subordinates or not.

```
public float getSalaries();
```

For this problem, there lies two technical points worth mentioning when apply ArchJava:

1. Dynamic Component.
   It's obvious that the Employee Class and EmployeeTree class are the components in our structure, but as we can dynamically instantiate Employee components in our EmployeeTree component, we'll use dynamic component creation.
2. Connection Pattern.
   As we have just mentioned, we have a single interface only for the employee component, so we end up with a special kind of port that actually shares the same interface. Further more, a single employee component sometimes participates in several connections using the same conceptual protocol, so we use port interface to describe this special case.

The code below shows exactly what we have just discussed.

Employee Component Class

The Employee component class stores the name and salary of each employee and allows us to fetch them as needed.

```
public component class Employee {
    public port interface up {
            provides float sumSalary();
  }
    public port interface down {
            requires float sumSalary();
  }
    public float sumSalary() {
       float sum = salary;
       if (left != null)
         sum += left.sumSalary();
       if (right != null)
         sum += right.sumSalary();
       return sum;
  }
    public void set(String n, float s, down l, down r) {
       name = n; salary = s; left = l; right = r;
  }
    private String name;
    private float salary;
    private down left;
    private down right;
}
```

EmployeeTree Component Class

This is the component class that we start by creating a CEO Employee and then add his subordinates and their subordinates.

```
public component class EmployeeTree {
  connect pattern Employee.up, Employee.down;
  private final Employee boss = new Employee();
  private final Employee marketVP = new Employee();
  private final Employee prodVP = new Employee();
     …
  void makeEmployee() {
    boss.down bossconnectL = connect(boss.down, marketVP.up);
    boss.down bossconnectR = connect(boss.down, prodVP.up);
    boss.set("CEO", 200000, bossconnectL, bossconnectR);
     …
  }
     …
}
```

More work could be done once we have constructed this Composite structure, such as load a visual JTree.

Actually, when we first work on this pattern, we have two great barriers:

1. How to represent recursion under ArchJava?

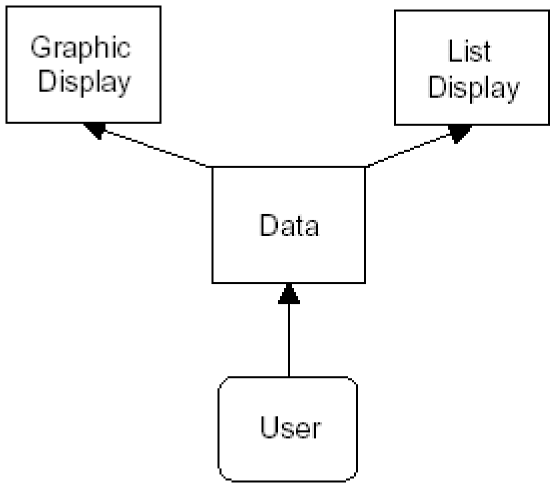
As the port communication between components here is different from common data flow.

2. How to avoid method of a component class that returns a reference component? As ArchJava prohibit this kind of method, it takes long time to think of an alternative way that has same functionality. And seems it really brings great trouble if objects are referenced in this way.
   And we come to a conclusion that ArchJava works in certain granularity that it would express the architecture rather than data structure that has finer granularity.

In summary, a composite is a collection of objects, any one of which may be either a composite, or just a primitive object. The Composite pattern allows you to define a class hierarchy of simple objects and more complex composite objects so that they appear to be the same to the client program. Because of this simplicity, the client can be that much simpler, since nodes and leaves are handled in the same way.
And ArchJava is able to handle this pattern by incorporating dynamic component creation and port interface, although a little bit un-straight forward.

Pattern 6: Observer

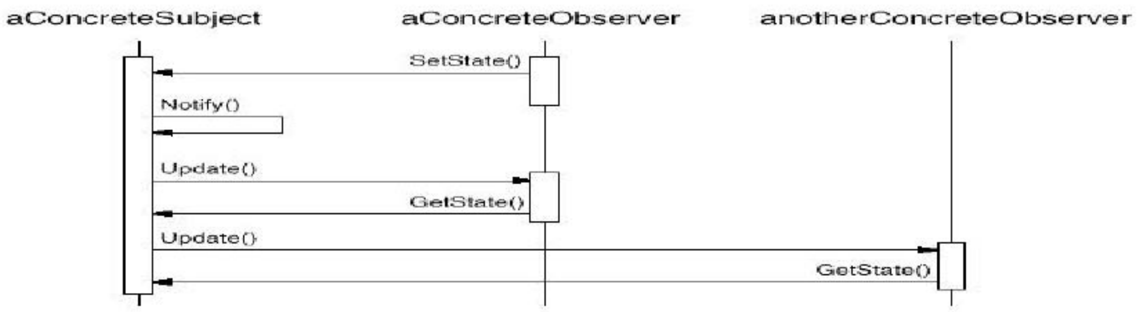
Probably it is the most important design pattern in the windows world, as we often would like to display data in more than one form at the same time and have all of the displays reflect any changes in that data. So Observer design pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
It’s simple to demonstrate as below and it’s also known as Model-View Design Pattern.



This is an extremely useful pattern and can be used in any of the following situations:

1. When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
2. When a change to one object requires changing others
3. When an object should be able to notify other objects without making assumptions about those objects

This pattern is easy to implement in Java, as Observable/Observer classes are built-in support for the Observer pattern. And as we have tested, ArchJava is able to express this pattern perfectly well.
A small example we used here is a simple project with a concrete Subject and two observers. As the subject changes, it notifies the two observers, thus changes the display. The basic relationship can be expressed below:



To make things easy, we simply have **component NameObserver** with only **port in**,

```
public component class NameObserver implements Observer {
        public port in {
                provides void update(Observable obj, Object arg);
                requires void setName(String name);
        }
…
}
```

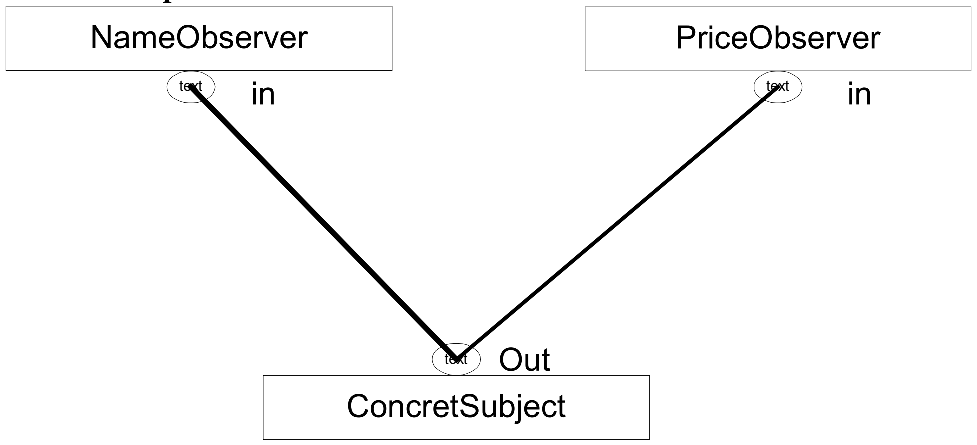
and **component PriceObserver** having a similar **port in**,

```
public component class PriceObserver implements Observer {
        public port in {
                provides void update(Observable obj, Object arg);
                requires void setPrice(float price);
        }
…
}
```

and **component ConcreteSubject** with only **port out**.

```
public component class ConcreteSubject extends Observable {
        public port out{
                provides void setName(String name) ;
                provides void setPrice(float price) ;
                requires void update(Observable obj, Object arg);
        }
…
}
```

And the connection among the observers and subjects is demonstrated in the **TestObserver component**.



```
public component class TestObservers {
        private final ConcreteSubject s = new ConcreteSubject("Corn Pops", 1.29f);
        private final NameObserver nameObs = new NameObserver();
        private final PriceObserver priceObs = new PriceObserver();

        connect s.out, nameObs.in, priceObs.in;
```

```
	public static void main(String args[]) {
		new TestObservers().test();
	}

	public void test()	{
		// Create the Subject and Observers.
		// Add those Observers!
		s.addObserver(nameObs);
		s.addObserver(priceObs);
		// Make changes to the Subject.
		s.setName("Frosted Flakes");
		s.setPrice(4.57f);
		s.setPrice(9.22f);
		s.setName("Surge Crispies");
	}
}
```

For observer pattern, we haven't encounterd much difficulty, except the connect. The port provided by ConcreteSubject contains all the methods with different Observers' ports. So the connection here is:

```
	connect s.out, nameObs.in, priceObs.in;
```

which is different from usual.

And look back at the case study the author has done, we can find that the two examples are both observer patterns, which in turns support our conclusion that ArchJava works well for observer patterns.

Discussion

- Can ArchJava express the architecture of a real program of significant complexity?
  Our answer is yes. In the six patterns we chose, we finished four of them. They are Abstract Factory, Composite, Façade and Observer. For Interpreter, we believe we can do it, although we gave up because of the too heavy workload. For Singleton, we don't know how to do it yet. Since a complex architecture can often take as simple patterns or can be divided into various comparatively simple patterns, the support to these patterns can reflect the expressiveness of ArchJava to some extent.

- How difficult is it to reengineer a Java program in order to express its architecture explicitly in ArchJava?
  Here is a table of our time spending on the project.

Items Done		Time Spend (in hour)
Paper Reading		6 (each person)
Example Code Studying		3 (each person)
Implement Patterns in ArchJava	Abstract Factory	6
	Singleton	2
	Façade	3
	Interpreter	5
	Composite	6
	Observer	3
Total		43

  We think these efforts are reasonable. Moreover, we started our programming without really understand how ArchJava works. So it would cost much less time if we were assigned the same amount of work at this moment.

- Does expressing a program's architecture in ArchJava help or hinder software evolution?
  Yes, definitely. While working with ArchJava, we are actually forced to think about the architecture first. And most of the times, we applied top down methodology to analyze the program. It's greatly helpful in deepening the understanding of program architecture. And for the software evolution, one of the most important reason that people develop design patterns is for software reuse. And as you can see, ArchJava support most of the patterns we tested, thus it can help in reusing software. Further more, it express architecture in such a clear way that it helps in software evolution.

- What's the most difficult thing in your work?
  To guarantee communication integrity, ArchJava has made many restrictions, such as only final field may be connected to a static connection, component cannot be used as a public method parameter, component constructor cannot have arguments of component type, etc. Our experience showed that for most these restrictions, developers can find a way around easily. But it's really a pain that it

doesn't allow component to be transmitted directly between ports. That's also the main reason we didn't finish the Interpreter pattern.

- Is the difficulty of reengineering has some relationships with the pattern classification?
  We don't think so. In our evaluation process, three most time consuming patterns, Factory, Interpreter and Composite, come from three different categories. And the two unfinished patterns also come from different categories.

Reference:

[ACN01] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. ICSE 2002.
[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural Reasoning in ArchJava. ECOOP 2002.
[ERR+94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. 1994
[J98] James W. Cooper. The Design Patterns: Java Companion. 1998