Computer Product Online Ordering System



Patrick Cronin 0662143

Patrick Russell 0734004

Module ID CS4135

Module Name Software Architecture

Table of Contents

Blank Marking Scheme 3

Brief Outline 5

Discussion of Patterns Selected 6

Observer Design Pattern 6

State Design Pattern 7

Composite Design Pattern 7

Decorator Pattern 8

Discussion of creational pattern 9

Runtime Diagram 10

Code 11

Testing 22

Problems Encountered 22

Critique 22

Architectural use cases 23

Availability 23

Security 23

Performances 24

Vendor Offerings 24

References 24

# Blank Marking Scheme

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **CS4135: Software Architecture**  **Assignment 1: Semester 1, 2010-2011** | | | | | | |
| Name1:  Name2: | | | ID1:  ID2: | | | |
|  | **Item** | **Detailed Description** | | **Marks Allocated** | | **Marks**  **Awarded** |
| Sub-total | Total |
| 1 | Presentation | * General Presentation * Adherence to guidelines i.e. front cover sheet, blank marking scheme, table of contents | | 1  1 | **2** |  |
| 2 | Discussion of patterns selected | For each of the 5 requisite patterns   * Context and forces * Consequences   with respect to concept presented in narrative | | 0.25  0.25 | **2** |  |
| 3 | Discussion of creational pattern |  | |  | **1** |  |
| 4 | Diagram | * Programming to interfaces, not implementation * Correct application of design patterns * All patterns integrated | | 1  1  1 | **3** |  |
| 5 | Code | * Matches/Supports/Realises diagrams * Object-Oriented * Exposes intent, facilitated through supporting documentation in the form of comments. Naming conventions clearly identify design patterns used | | 2  1  2 | **5** |  |
| 6 | Testing |  | |  | **2** |  |
| 7 | Critique | Is it the case that the design patterns selected achieve their objectives? If not, why not? Any alternatives? | |  | **2** |  |
| 8 | Demo | * Competent code inspection * Working demo | | 1.0  1.0 | **2** |  |
| 10 | Runtime Infrastructure / technology pipeline | * Diagram * Clearly identified architectural use cases * Choice of vendor offerings * Discussion of reasons for selection from offerings targeted at quality attributes | | 1.0  2.0  1.0  1.0 | **5** |  |
| 9 | References |  | |  | **1** |  |
|  | **SUB-TOTAL (A)** | | | | **25** |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **PENALTIES** | | | |
|  | **Description** | **Member 1** | **Member 2** |
| 1 | Late Submission |  |  |
| 2 | Failure to contribute to coding effort |  |  |
| 3 | Failure to contribute to writing of report |  |  |
| 4 | Failure to report problems with team dynamics |  |  |
| 5 | Failure to contribute to demo |  |  |
|  | **Sub-total (B)** |  |  |

|  |  |  |
| --- | --- | --- |
| **FINAL MARKS AWARDED** | | |
|  | **Member 1** | **Member 2** |
| **(A-B)** |  |  |

# Brief Outline

A computer manufacturer wants to create a framework to support client orders and we want to have the framework with the capacity to expandable to other products if the computer manufacturer ever wants to. For maintainability and modifiability we have a number of design patterns to support problems that this type of framework might arise with. We used Java as the language for the framework as there is already a number of application servers that the language offers to use with.

The company requires a program which must incorporate the composite design structure to maintain a tree structure of the components of a computer.

For example a chassis can contain a motherboard which can contain RAM.

The framework must also use the state design pattern in order to calculate the VAT for an appropriate region.

The framework must have the support for adding additional hard drives and or graphics cards using the decorator pattern.

Product instantiation must be supported by a creational pattern such as the Factory method.

Customers must be able to update the amount of products they require which would incorporate the observer design pattern to make sure that the VAT amount is up to date should they add another product.

# Discussion of Patterns Selected

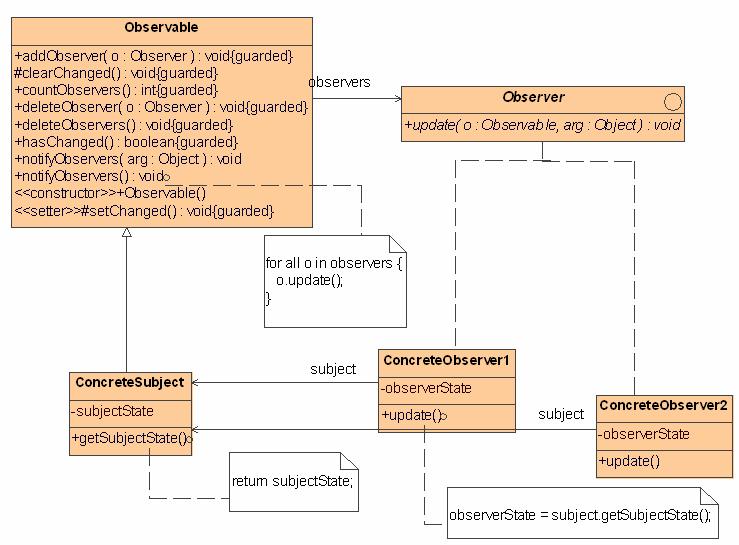
## Observer Design Pattern

The observer design pattern is generally used in software architectures where it is required that objects may need to be notified of a change to a specific object. This is done by the subject holding a list of its dependents (called observers) notifies these observers should there be a state change. As there already exists the java library “java.util.observer/observers” we used this instead of creating our own observer classes.

The observable class in our program is the ComponentComposite.java class.

The reason for this is that every time we add a new composite to our product notifyObservers() gets called. The observer which is the TransactionManager class then calls its update method which in turn causes the TransactionManager instance to recalculate its new total (Total price with vat).

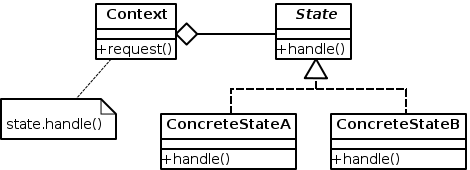
This implementation may seem a little obsessive as it calls notifyObservers() every time a new item is added to a tree which could happen thousands of times if you were to create a few hundred products however the specification stated that the system must just handle up to 100 computers and with present day computers having adamant resources these updates should not be a problem.

.

## State Design Pattern

The state design pattern in our system is incorporated in the vat calculation system where we set the state (Region) to either Ireland or UK. This is done as soon as the program starts, it is important to do this first as if we try to calculate the vat when the region state is null then we will get errors. As mentioned above when the observer is called and an update is made it passes the state variable which is an enum of regionEnum type and depending on its state “IRELAND” or “UK” we return the updated total to the total variable in the TransactionManager instance.

The framework is currently minimum with only two states but it supports adding additional states in future which makes the code more modifiable.

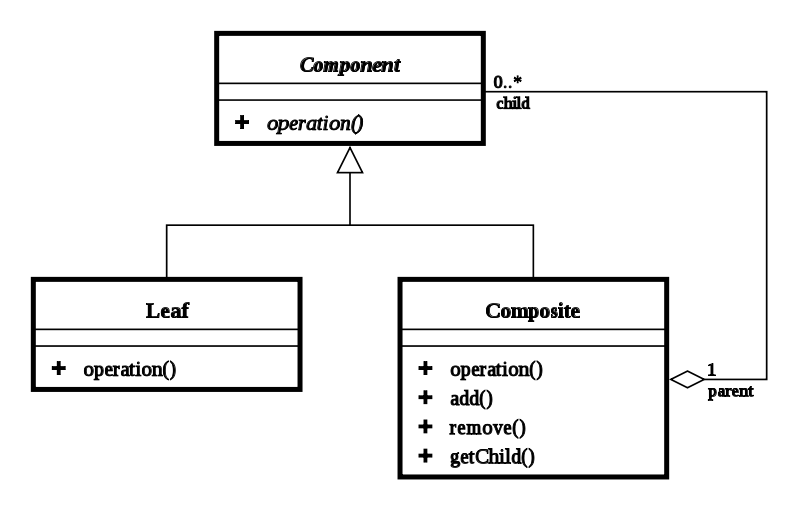


## 

## Composite Design Pattern

We incorporated the composite pattern in our system to hold components which we add to our product upon creation of the product in our factory. Once the factory creates a new product it then adds a composite to this product, we can then add further composites to this composite as well as extra components such as RAM. Our system currently has a hardcoded list of composites and composite components to be created however this may be changed in future to add additional components or additional composites making our framework more flexible than a project which didn’t incorporate this design pattern. The advantage of using this pattern in this framework is that it makes programing additional components easier and less tedious since we can treat composite components and composites equally just referring to the root node if we want something like a price for the whole product for instance. Using this pattern also makes our framework more modifiable as adding and removing new components, composites involves changing little code.

We opted for transparency over safety in our composite design pattern the reason for this is that it made coding easier and for a system the size of what was required for our frame work safety would have made it better but wasn’t significantly necessary. This may be a problem as if a company were to use our framework it introduces the possibility of a client trying to do meaningless things like trying to add and remove objects from leaf objects.



## Decorator Pattern

The intent of the decorator pattern is to dynamically attach responsibilities to an object. Its to give an alternative to subclassing.

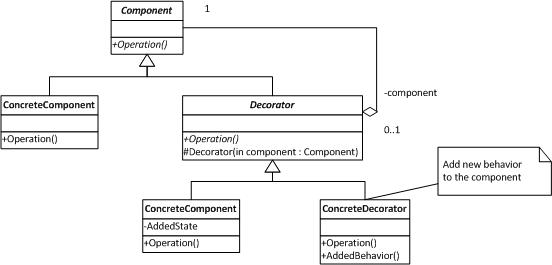
We used the decorator pattern to order extra items such as RAM and hard-drive for a particular computer the user would have selected while ordering. This should not be confused with the composite pattern in our project which simply adds a set list of items to a composite and you never add more than one composite component to a composite.

Decorating a Product involved traversing the tree using an iterator for searching the leaf nodes and comparing their String description in order to locate a particular node.

This raised a problem, once we found a node we needed to remove it in order to decorate it and add it back into the tree where we removed it from.

We solved this problem by adding a reference to the parent in each of the composite components. Once we found the item we could remove it and replace the newly decorated node to the composite we removed it from.

The consequences of using this pattern is that it reduces the possibility of class explosion. If we were not to use this pattern there would have to be a great deal of classes for a computer with extra ram and graphics card and one with just extra RAM etc. The decorator overcomes this problem by just locating the particular project to increase and decorating that object. This also means we only need to talk to the outermost component in the decorator to calculate the total cost of the additional items.



# Discussion of creational pattern

A Creational Pattern is a pattern that deals with object creation. There are a number of Creational Patterns including Factory Method, Abstract Factory Method, Builder, Singleton and Prototype. We chose to use the Abstract Factory method in our project as we wanted to leave the possibility of adding additional factories in future. A product factory in our system creates a computer factory in our project which then creates a new computer of the selected type. The benefit of doing this it makes it more possible to use our project as a framework in future for other product types. All we would need to do is add another factory from the product factory interface. So if our customer ever wanted to do printers or flowerpots instead it could be done. The trade-off is that we have to make it more abstract by introducing an extra interface. So instead of just having a Computer Factory we had to have a product factory as well.

# Runtime Diagram

# 

# Code

**ComponetComposite.java**

package composite;

import java.util.\*;

public class ComponentComposite extends Observable implements Product {

ArrayList<Product> productComponents = new ArrayList<Product>();

String description;

String name;

Double price = 0.0;

Product parent;

public Product myProduct;

ArrayList <Product> currentRow = productComponents;

ArrayList <Product> traverse = new ArrayList <Product>();

public Product findItem(String toBeFound){

/\*Iterate through the composite structure and return a reference

\* to the product matching the string name

\*/

int i = currentRow.size();

int j = 0;

while (i!=j){

String compname = currentRow.get(j).getName();

Product product = currentRow.get(j);

if (compname.equals(toBeFound)){

return product;

}

else if (product.getName() == "composite")

traverse.add(product);

j++;

}

/\*

Iterator<Product> iterator = productComponents.iterator();

while (iterator.hasNext()){

iterator = compos.iterator();

Product product =

(Product)iterator.next();

String compname = product.getName();

System.out.println("Is" + compname + toBeFound + "?");

if (compname.equals(toBeFound))

return product;

if (product.getName() == "composite")

{

traverse.add(product);

}

}

\*/

Product product = traverse.get(0);

currentRow = product.getChildren();

traverse.remove(0);

Product found = findItem(toBeFound);

return found;

//return null;

}

public ArrayList<Product> getChildren() {

return productComponents;

}

public void removeComponent(Product product) {

productComponents.remove(product);

setChanged();

notifyObservers();

//pass a reference to a component and remove it from the array

}

public void addComponent(Product parent, Product node ) {

node.setName("composite");

productComponents.add(node);

node.setParent(parent);

setChanged();

notifyObservers();

/\*add a component to a composite, also add a refernce of the parent

\* to the newly created child node

\*/

}

public String getName() {

return name;

}

public double getPrice() {

ArrayList <Product> children = this.getChildren();

int i = children.size();

int j = 0;

while(i != 0){

Product child = children.get(j);

RAM ram = new RAM();

System.out.println(ram.getPrice() + child.getPrice());

price = price + child.getPrice();

j--;

i--;

return price;

}

return (Double) null;

}

@Override

public void setPrice(double price) {

}

@Override

public Product getParent() {

return parent;

}

@Override

public void setParent(Product product) {

parent = product;

}

public void setName(String named){

name = named;

}

}

**RAM.java**

package composite;

import java.util.ArrayList;

public class RAM implements RamInterface, Product {

Product parent;

public double getPrice() {

return 30.00;

}

public String getName(){

return "ram";

}

public Product getParent() {

return parent;

}

@Override

public void addComponent(Product parent, Product node ) {

}

@Override

public void removeComponent(Product product ) {

}

@Override

public Product findItem(String toBeFound) {

return null;

}

@Override

public void setPrice(double price) {

}

@Override

public void setParent(Product product) {

parent = product;

}

@Override

public void setName(String name) {

}

@Override

public ArrayList<Product> getChildren() {

// TODO Auto-generated method stub

return null;

}

}

**RAMDecorator.java**

**package** composite;

**import** java.util.ArrayList;

**public** **class** RamDecorator **implements** RamInterface {

Product product;

Product parent;

**int** numberOfRam;

**public** RamDecorator(Product product) {

**double** price = product.getPrice();

**double** newPrice = price + 30.0;

product.setPrice(newPrice);

}

@Override

**public** **double** getPrice() {

**return** 30.00 + product.getPrice();

}

@Override

**public** **void** addComponent(Product parent, Product node ) {

}

@Override

**public** **void** removeComponent(Product product ) {

}

@Override

**public** **void** setName(String name) {

// **TODO** Auto-generated method stub

}

@Override

**public** **void** setPrice(**double** price) {

// **TODO** Auto-generated method stub

}

@Override

**public** ArrayList<Product> getChildren() {

// **TODO** Auto-generated method stub

**return** **null**;

}

@Override

**public** Product getParent() {

// **TODO** Auto-generated method stub

**return** **null**;

}

@Override

**public** **void** setParent(Product product) {

// **TODO** Auto-generated method stub

}

@Override

**public** String getName() {

// **TODO** Auto-generated method stub

**return** **null**;

}

@Override

**public** Product findItem(String toBeFound) {

// **TODO** Auto-generated method stub

**return** **null**;

}

}

**RAMInterface.Java**

**package** composite;

**public** **interface** RamInterface **extends** Product {

}

**TransactionManager.java**

package controller;

import factory.\*;

import java.util.\*;

import region.\*;

import composite.\*;

import decorator.\*;

public class TransactionManager implements Observer {

public Region Region;

public ArrayList<Product> productList = new ArrayList<Product>();//List of products

private RegionEnum region;

public double subTotal = 1.0;

public double total = 1.0;

Product found = null;

public RegionEnum getRegion(){

return region;

}

public void decorateProduct(Product product, ComponentEnum componentEnum){

DecoratorManager d = new DecoratorManager();

d.decorateProduct(product, componentEnum, 1);

}

public int getNumber() {

return productList.size(); //number if products in list

}

public void addProductToList(ProductTypeEnum productTypeEnum, int quantity) {

/\* this function creates a new computer factory and loops creating the specified

\* number of prodcuts.

\*/

ProductFactory productFactory = new ComputerFactory(this);

int i = quantity;

while (i > 0){

System.out.println(productTypeEnum);

Product newproduct = productFactory.createProduct(productTypeEnum);

productList.add(newproduct);

i--;

double newitemprice = newproduct.getPrice();

updateSubTotal(newitemprice);

}

}

public void updateSubTotal(double newitemprice){

subTotal += newitemprice;

}

public void updateTotal(RegionEnum regionEnum){

if (RegionEnum.IRELAND == regionEnum){

Region ireland = new Ireland();

total = ireland.getVat(subTotal);

} //passes in the prevat price and returns the total price including vat

else if(RegionEnum.UK == regionEnum){

Region uk = new UK();

total = uk.getVat(subTotal);

}

}

public void setRegion(RegionEnum newRegion) {

region = newRegion;

}

DecoratorInterface decoratorInterface;

ComponentEnum componentEnum;

public void upgradeProduct(Product product, ComponentEnum componentEnum, int quantity) {

//adds an additional component to a product using the decorator

decoratorInterface.decorateProduct(product, componentEnum, quantity);

}

public void update(Observable o, Object arg) {

updateTotal(region);

//when update is called by an observer we call this method to update the vat

}

}

**DecoratorInterface.java**

package decorator;

import composite.\*;

import controller.\*;

public interface DecoratorInterface {

public Product decorateProduct(Product product, ComponentEnum componentEnum, int quantity);

public Product addRam(Product product, int quantity);

public Product addHardDrive(Product product, int quantity);

public Product addGraphics(Product product, int quantity);

}

**DecoratorManager.java**

package decorator;

import composite.\*;

import controller.\*;

import java.util.Observable;

public class DecoratorManager extends Observable implements DecoratorInterface{

public Product decorateProduct(Product product, ComponentEnum componentEnum, int quantity){

if( ComponentEnum.RAM == componentEnum){

return addRam(product, quantity);

/\*find out what item is to be decorated then call add ram

to decorate the product

\*/

}

else if (ComponentEnum.HARDDRIVE == componentEnum){

return addHardDrive(product, quantity);

}

else if (ComponentEnum.GRAPHICSCARD == componentEnum){

return addGraphics(product, quantity);

}

return null;

}

public Product addHardDrive(Product product, int quantity) {

Product harditem = product.findItem("harddrive");

Product parent = harditem.getParent();

RamDecorator decor = new RamDecorator(parent);

product.removeComponent(harditem);

harditem.addComponent(parent, decor);

return harditem;

}

public Product addGraphics(Product product, int quantity) {

Product graph = product.findItem("graphicscard");

Product parent = graph.getParent();

RamDecorator decor = new RamDecorator(parent);

product.removeComponent(graph);

product.addComponent(parent, decor);

return product;

}

public Product addRam(Product product, int quantity) {

/\* when it is called we first find an item in the

composite tree matchin the name then we get its parent,

remove it, decorate it and place the newly decorated component

back into the tree.

\*/

Product ramitem = product.findItem("ram");

Product parent = ramitem.getParent();

RamDecorator decor = new RamDecorator(parent);

product.removeComponent(ramitem);

product.addComponent(parent, decor);

return product;

}

}

**ComputerFactory.java**

package factory;

import java.util.Observable;

import java.util.Observer;

import composite.\*;

import controller.\*;

public class ComputerFactory extends Observable implements ProductFactory {

public ProductTypeEnum newProduct;

private Observer transactionManagerRef;

public ComputerFactory(Observer observerRef){

transactionManagerRef = observerRef;

}

@Override

public Product createProduct(ProductTypeEnum productTypeEnum) {

if(ProductTypeEnum.DESKTOP == productTypeEnum){

return createDesktopComputer();

}

else if(ProductTypeEnum.LAPTOP == productTypeEnum){

return createLaptopComputer();

}

//find out what type of computer we need and create it using the factory

return null;

}

public Product createDesktopComputer() {

//construct the composite tree with our components

Product motherboard = new ComponentComposite();

Product ram = new RAM();

Product cpu = new CPU();

Product sound = new SoundCard();

Product screen = new Screen();

Product graphics = new GraphicsCard();

Product harddrive = new HardDrive();

((ComponentComposite) motherboard).addObserver(transactionManagerRef);

motherboard.addComponent(motherboard, ram);

motherboard.addComponent(motherboard, cpu);

motherboard.addComponent(motherboard, sound);

motherboard.addComponent(motherboard, graphics);

Product chassis = new ComponentComposite();

((ComponentComposite) chassis).addObserver(transactionManagerRef);

chassis.addComponent(chassis, motherboard);

chassis.addComponent(chassis, harddrive);

Product computer = new ComponentComposite();

((ComponentComposite) computer).addObserver(transactionManagerRef);

computer.addComponent(computer, chassis);

computer.addComponent(computer, screen);

return computer;

}

public Product createLaptopComputer() {

Product motherboard = new ComponentComposite();

Product ram = new RAM();

Product cpu = new CPU();

Product sound = new SoundCard();

Product screen = new Screen();

Product graphics = new GraphicsCard();

Product harddrive = new HardDrive();

((ComponentComposite) motherboard).addObserver(transactionManagerRef);

motherboard.addComponent(motherboard, ram);

motherboard.addComponent(motherboard, cpu);

motherboard.addComponent(motherboard, sound);

motherboard.addComponent(motherboard, graphics);

Product chassis = new ComponentComposite();

((ComponentComposite) chassis).addObserver(transactionManagerRef);

chassis.addComponent(chassis, motherboard);

chassis.addComponent(chassis, harddrive);

Product computer = new ComponentComposite();

((ComponentComposite) computer).addObserver(transactionManagerRef);

computer.addComponent(computer, chassis);

computer.addComponent(computer,screen);

return computer;

}

}

**productFactory.java**

package factory;

import composite.\*;

import controller.\*;

public interface ProductFactory {

public Product createProduct(ProductTypeEnum productTypeEnum);

}

**Ireland.java**

package region;

public class Ireland implements Region {

private double vatRate = 1.21;

public double getVat(double subTotal){

return subTotal \* vatRate;

}

}

**Region.java**

package region;

public interface Region

{

public double getVat(double subTotal);

}

**(note: some similar classes were left out of this hardcopy due to the size)**

# Testing

We used j-units for testing using Test Driven Development (TDD) in our project. One of the classes which we wrote a test for was the Transaction manager and here is a snippet from one of the j-unit files.

@Test

**public** **void** testTransactionManager() {

transManager = **new** TransactionManager();

*assertNotNull*(transManager);

}

@Test

**public** **void** testGetRegion() {

transManager = **new** TransactionManager();

*assertNotNull*(transManager.getRegion());

transManager.setRegion(RegionEnum.*IRELAND*);

*assertEquals*(RegionEnum.*IRELAND*, transManager.getRegion());

}

Here we are checking that if we instantiate a new transaction manager is its value not null if it isn’t its passed and this component in our project executes as expected.

The second test is testing the setting the region for a TransactionManager object. Again here we assert that the result of setting the region isn’t null and in doing this we also test the getRegion() method.

We found the use of J-units very useful for creating our project as it allowed us to test individual components to make sure that not only it changed a variable but that said variable had the expected value.

# Problems Encountered

As the decorator and composite have different philosophies in how objects should be structured. We found it hard trying to integrate the decorator pattern with the composite pattern and it was tricky. So the integration of the decorator pattern with the composite pattern was more or less forced on each other. Also as we hadn't really touched Java since first year there was a lot of cobwebs to get clear of.

# Critique

The state pattern which we incorporated into the part of our program which updates the VAT worked quite well for our project, it was a very simple implementation of state so it worked perfectly.

The only criticism I would have of using this pattern is that perhaps using state here is a little tedious as the state object can have only one of two states.

In an enterprise level system I would imagine state would be used for something which is a great deal more complicated such as UI views where the context would depend on what a UI system would put out.

The decorator pattern which we used in our system and I think it is a near perfect example of a good place to use this pattern. If we were to not use the decorator pattern here and we added lets say 100 extra RAM components there would be class explosion and it would just be a mess. This pattern makes it easier for us to just talk to one object which will maintain a total for us of all its decorated products.

The abstract factory pattern which we chose I would say for this system is probably not necessary. The advantage of it is that it makes it easier for somebody in the future to add an additional factory for instance an MP3 player factory and implement the same framework. For the project however this is probably unnecessary as we only need to create computers .

I believe the use of the composite design pattern in this system was probably a good idea as it makes it easier for us to do something like calculate the total cost of an item, we just talk to the composite at the root of the tree.

We also do not need to discriminate between a leaf node and a branch which made our code a lot less complex and error prone.

I believe the use of the observer pattern in our system was quite useful when we added new items and decorated products. It made sure that every time something was added to a product or a product was created that the total with the vat was kept up to date. It may have been unnecessary to use this pattern here as we only have one observer which is our transaction manager. The update is called quite a few times as components are added to our product but only one thing gets changed when our observer receives the update from the observees.

# Architectural use cases

## Availability

This framework is available to British Isles customers only and should be able to expand to other countries within the world.

## Security

The user should be able to find out the full amount of the product of how much they want to pay, before they login their details to pay for the product. The system should logout automatically when they have paid. If the system is left idle for 5 minutes in the middle of a transaction it should give a warning that it will logout automatically within 20 seconds if the user doesn’t react.

**Extensibility**

Our framework provides quite a good level of extensibility. The reason for this is that you can plug in almost any UI you want to the system to make it work. It can handle simple requests such as add a new product and since the factory does this for you the designer of a new plugin to our framework would not need to be concerned with how these objects are created.

## Performance

The performance of our system could be criticised as regards the observer pattern in our system. Every time we add a new component to a product we call an update, this may cause an overhead if run on a server with multiple simultaneous transactions. Although our framework my have some performance issues I think over all it is quite good as use of the decorator pattern means that we reduce the use of excess classes which cuts down on the amount of code which is important for network over-heads.

# Vendor Offerings

There are a number of Java Application Servers we narrowed them down to IBM’s Websphere Application Server, Mort Bay Consulting’s Jetty and Open Community’s Glassfish AS. We chose Glassfish as it had a large support base in the open community, it had Java EE and it is more efficient than Websphere. While Jetty we found was a good product we wouldn’t be able to upscale the product as it didn’t have Java EE Compatibility. While Websphere AS is a good product, to pay for the license would make it economically less sense to go with this one.

# References

Erich Gamma, 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1 Edition. Addison-Wesley Professional.  
  
Eric Freeman, 2004. *Head First Design Patterns*. Edition. O'Reilly/SPDe.