Build A PC

Briefing Document

William Laffan – 09004758
Ciaran Curley – 09005110
Eoin Joy - 09004456

Contents

Narrative

In Software design and architecture, design patterns are a set of common solutions to common problems. They are notably very reusable. This is in part because they do not have a concrete software implementation; they are just ideas or templates of solutions to common problems. Design Patterns are usually defined by the interactions between classes and do not specify the concrete implementations. The Gang of Four (GoF) popularised the concept of Design Patterns to software.
The goal of software design patterns, other than solving common problems, is to promote good software development and non-functional requirements such as portability, extensibility, interoperability, etc. These requirements are unsurprisingly called the "-ilities."

We were tasked with producing a framework using some of the GoF's design patterns from their text – "Design Patterns: Elements of Reusable Object-Oriented Software" This document is the result. The framework is a system which will allows users to order a PC online that has been custom built from the user's specifications. We were instructed to implement at least six design patterns; at least one from each family of patterns (creational, structural, behavioural) and at least one self researched pattern. The patterns we chose were Builder (creational), Factory (creational), Observer (behavioural), State (behavioural) and two self researched ones: Composite (structural) and Flyweight (structural). If you look at GoF design pattern relationships you can notice that other than the isolated Factory and Observer, the patterns all connect to each other.

Before we started writing any code we spent substantial time deciding on patterns, designing the structure of the framework and examining how the patterns interact with each other. For example, we folded the Leaves and Composites of the Composite pattern into the shared and unshared Flyweights of the Flyweight Pattern. We used Microsoft Visual Studio 2010's UML modelling tools.
Refactoring, of course took place throughout the project to better meet non-functional requirements and more appropriately implement the design patterns. We did this using Extract method, Encapsulate fields and Type generalisation with interfaces.

Use Cases

Make a PC: User can make a new pc from list of available components.

- User is presented with a list of available components
- Optional branch: Pick a template PC.
- User selects the components they want
- User submits selections.
- An Order is built containing an instance of the PC and an Order no.

Choose a template PC: User can pick a pre-built pc. Branching Use Case from Make a PC.

- User chooses a template PC
- A set of components are selected for the user

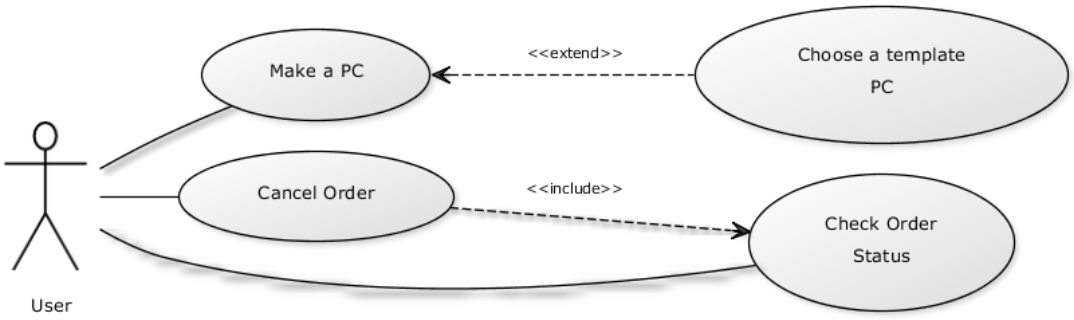
Check Order Status: Users can look up the status of their order from an order no.

- User enters their Order Number

- The state of their order is output
- User is asked if they wish to cancel. If state is advanced enough, user will be notified of a cancellation fee
- If yes, they are asked to confirm.

Cancel Order: User can cancel their order under certain conditions.

- User enters order number
- Their order's state is output. Cancellation fee is calculated and output.
- Failure condition: If state is too far advanced, order may not be cancelled
- User is prompted to confirm cancellation and agreement to cancellation fee



Non-Functional Requirements

Framework must allow for extensibility in case the service expands its market to PC peripherals.
Framework must be deployable on multiple platforms. The framework is intended to deploy online and will likely need to expand to accommodate emerging web technologies
Framework must support common interoperability protocols

Supporting Non-functional Requirements

Extensibility: "Programming towards interfaces and not implementation" achieves extensibility. In the context of this framework, we have allowed for the extension of new Factories to handle new Products. The use of the State pattern allows us to easily modify and update the order tracking process.

Portability: It is safe to assume that the major modern web browsers support Java and contain a JVM and the Java Class Library. Java is also supported by modern major operating systems like Linux and Windows.

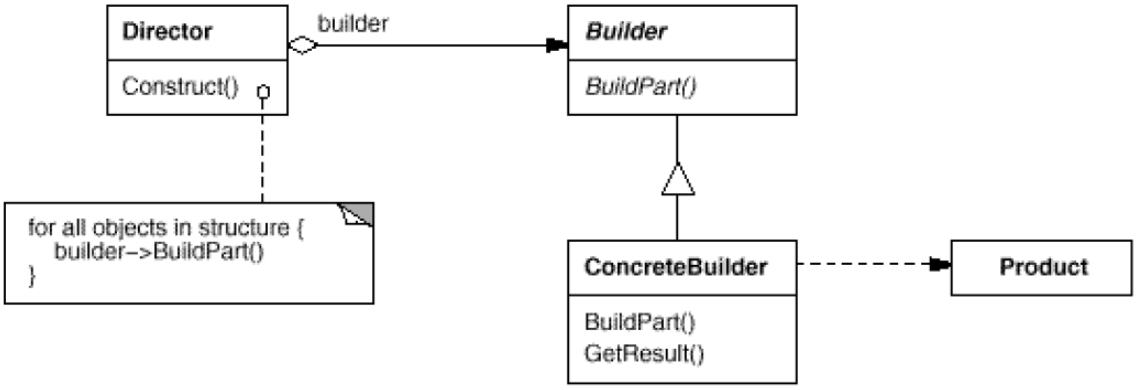
Interoperability: Interoperability in software is most commonly achieved by using common protocols and file formats. This ties in with portability quite well. Java is well known for being deployable on many platforms as well as being well supported by most

major browsers and operating systems. As such, its pervasiveness makes it very likely to work well with other systems.

Design Patterns

Builder

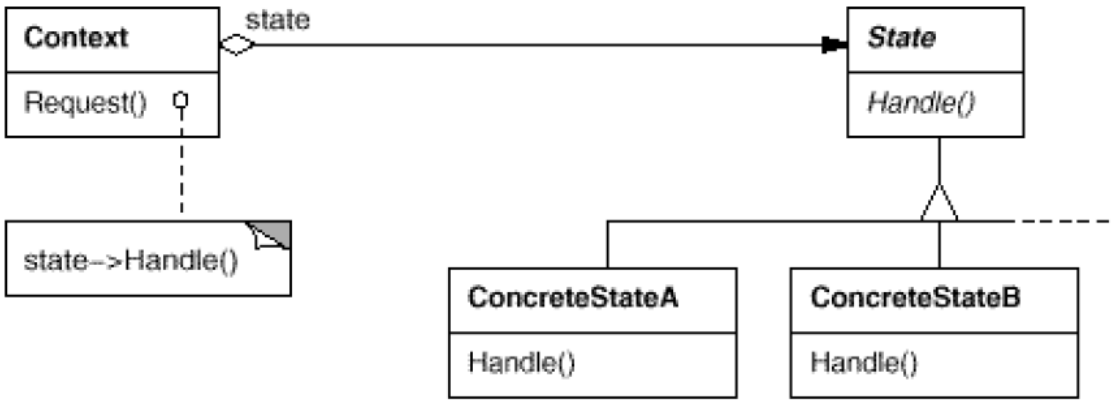
The Builder pattern is a creational pattern which is often used in conjunction with the Composite design pattern. It is used to separate the creation of complex objects from its representation so that the same process can create different representations.
In the context of this system, we use it to compile all the PC components into a PC for the user.
The builder uses an interface that hides the internal structure of the product, in this case, a PC.



1Builder Pattern Diagram

State

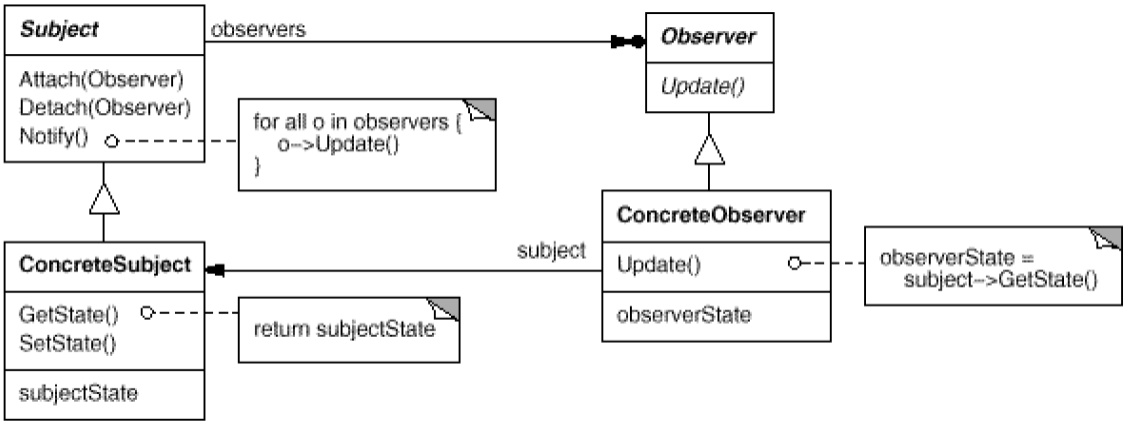
The State pattern is a commonly used behavioural pattern in which an object changes its behaviour based on its internal state. This is achieved with an abstract class for the current state and concrete ones for each of the possible states. In the context of this framework, the State behavioural pattern is used to handle how far along an order has been processed. An example of the changing behaviour is the ability of a user to cancel an order and how the system reacts to it.
When using the state pattern, it is important to decide what is in charge of changing states. If the states always change in the same order, it is safe for states to have the functionality to change themselves, as they'll always change to the next state in the chain. This framework represents a real factory building real PCs; the actual changing of states would then be handled by a real human at a computer terminal.



2State Pattern Diagram

Observer

The Observer pattern is a behavioural pattern in which one object is “Observed” many others. The Subject, that is, the object being observed, has a list of Objects, all using an Observer interface. When the subject changes state, it calls a method that let's all the observers know that the state has changed. The Observer can use either the “push” or “pull” model. In the push model, when the Subject updates the Observers, it also updates them with the new state. In the pull model, the Subject only alerts the Observer that it has changed; it is up to the Observer to then inspect the Subject.
In the context of this framework, the pattern is used so the user can see the state of the their order

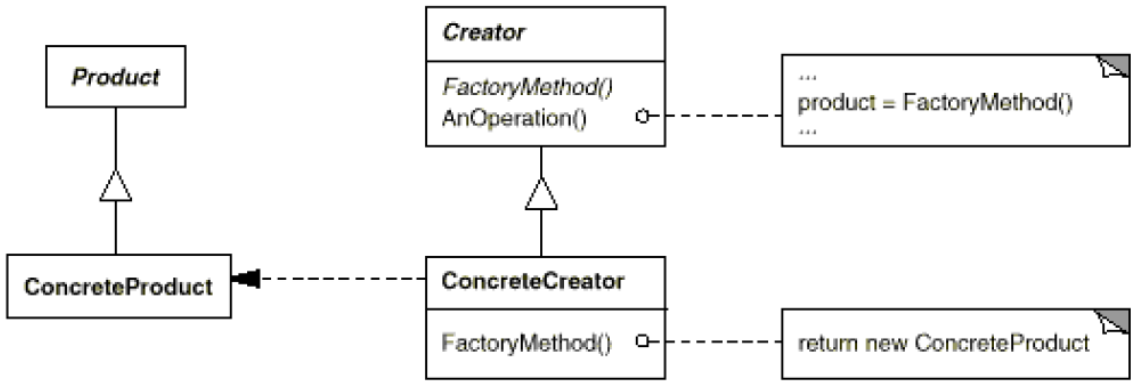


3Observer Pattern Diagram

Factory

The Factory pattern is a commonly used creational pattern in which an interface is designed to handle the creation of objects. This allows the classes to decide what objects to instantiate. It is useful when it is unknown which classes need to be instantiated. In this framework, we use the Factory method when the Flyweight needs to create an object that does not yet exist. The concreteFactory (in our case ComponentFactory) implements the Factory Interface (IFactory) and creates any object implementing the Product interface

(IProduct). If we wished to expand this framework, one such action could be to have a second concreteFactory to create Products such as computer peripherals.



4Factory Method Diagram

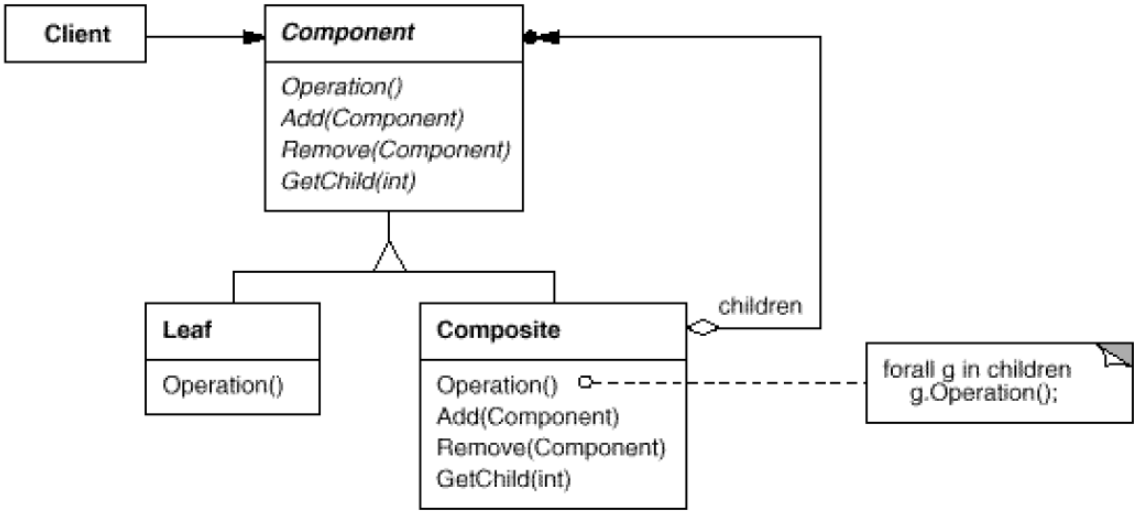
Our chosen Patterns

Composite

The Composite pattern is a structural pattern which lets clients treat individual objects and objects composed of smaller objects uniformly. In this system it represents PC components and the bays they occupy. For example, a motherboard is itself, a PC Component but it can also store a variety of possible graphics cards. So in this example, a motherboard is a Composite and a graphics card is a Leaf. The key to the Composite pattern is an abstract class that represents both primitives and their containers, in our case this is the "PC_Component" class. In our system, the Leaf subclass is called PC_Part and represents elementary components such as CPUs and GPUs, the Composite subclass represents components with interchangeable sub components like a motherboard, or a drive bay.

The advantage of this is that it allows us to represent the fairly complex PC as a relatively simple tree structure of composites and leaves.

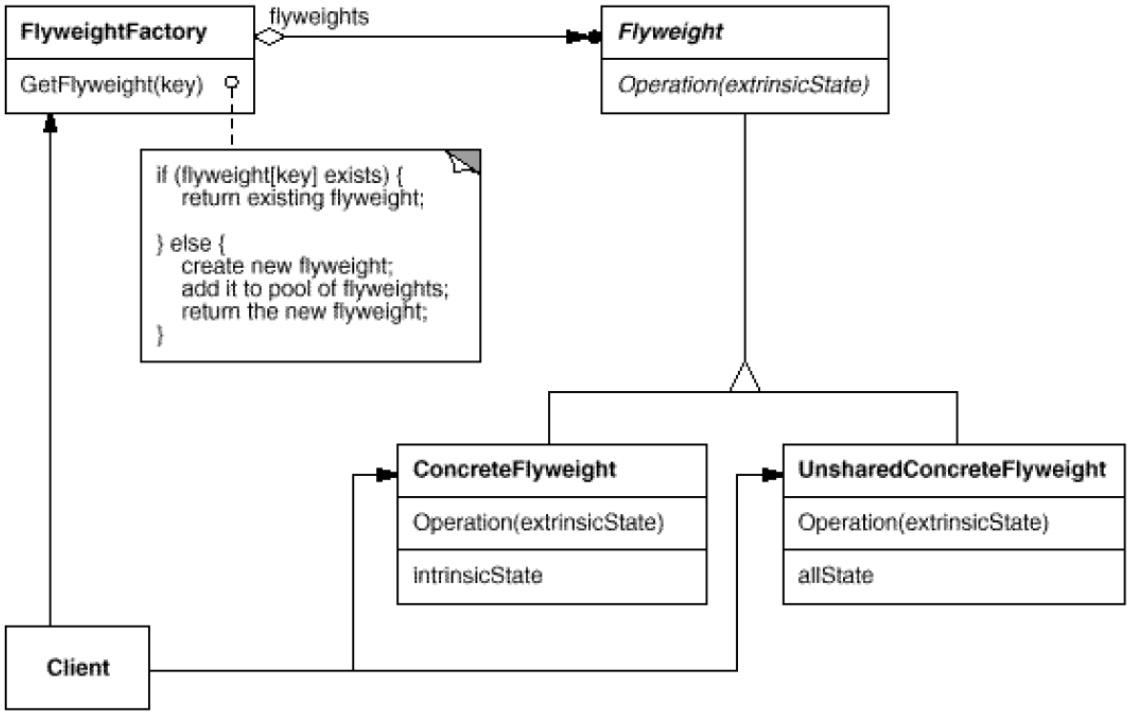
One disadvantage of the Composite pattern though, is that because leaves and composites are treated uniformly, you cannot restrict the components of a composite. For example, we know that a CPU logically cannot be a leaf of a drive bay however we cannot trust the system to enforce that; it requires checks at runtime.



5Composite Pattern Diagram

Flyweight

The Flyweight pattern is a structural pattern used to limit unnecessary memory usage when using large numbers of small objects. In the context of this framework, it is used to handle the large duplicates of pc components such as graphics cards, CPUs, power supplies etc. When used with the Composite pattern, the only shareable Flyweights are the Composite's Leaf nodes (In this case the PC_Part objects). The Containers are not shareable because Container could hold any number of combinations of different Leafs. In the most cases with the use of the Flyweight pattern, uniquely identifying attributes are externalized to an "extrinsic state." In the case of this framework, there are no extrinsic states as the Leaf (PC_Part) merely consists of a name and cost. Names of components are prefaced with a "C_" or a "P_" to designate whether the item is a Part or Container. Java's "instanceof" keyword could be used however, if an instance of the object didn't already exist, FlyweightFactory would have no way of knowing what object to instantiate.

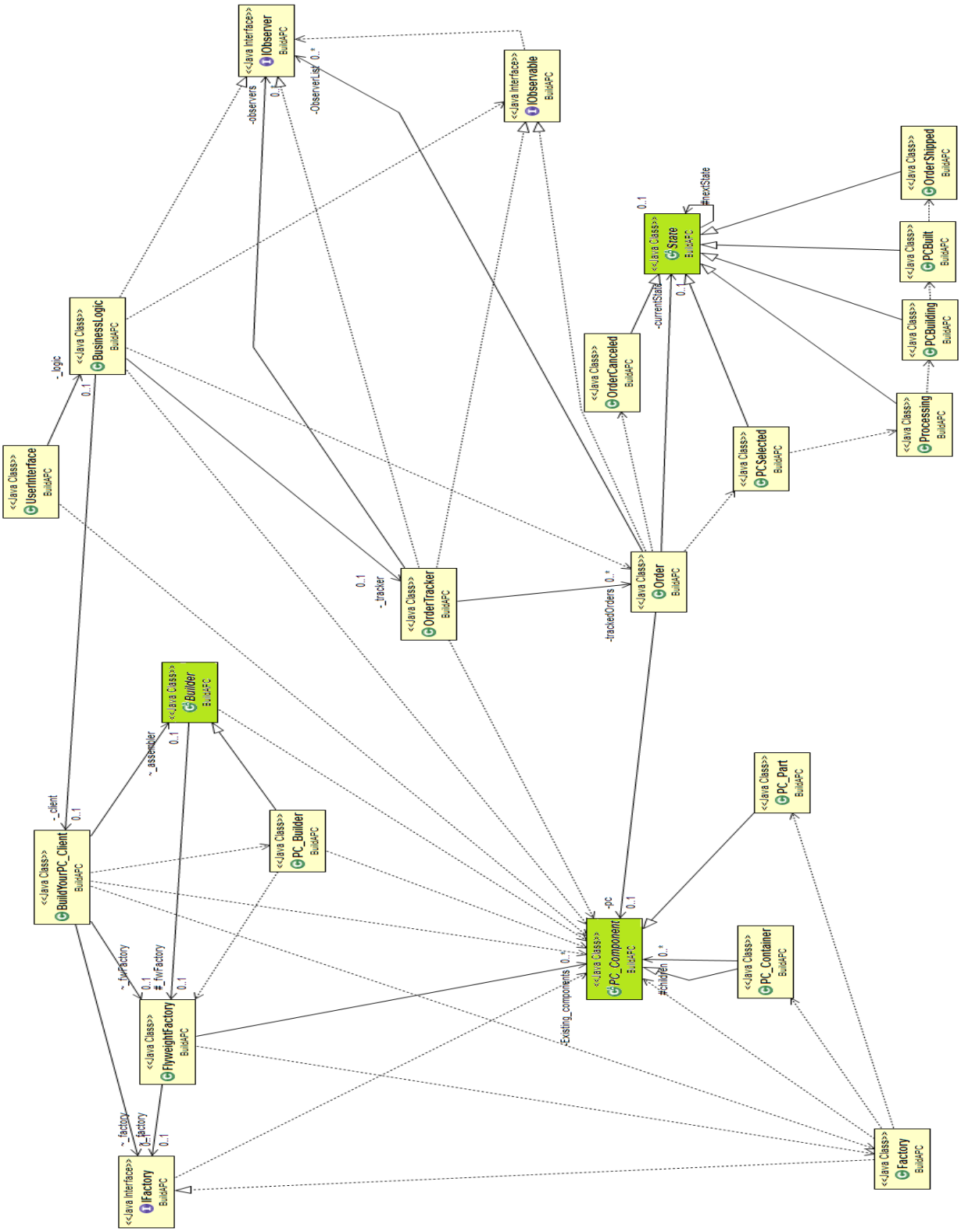


6Flyweight Factory Pattern Diagram

Architectural Pattern

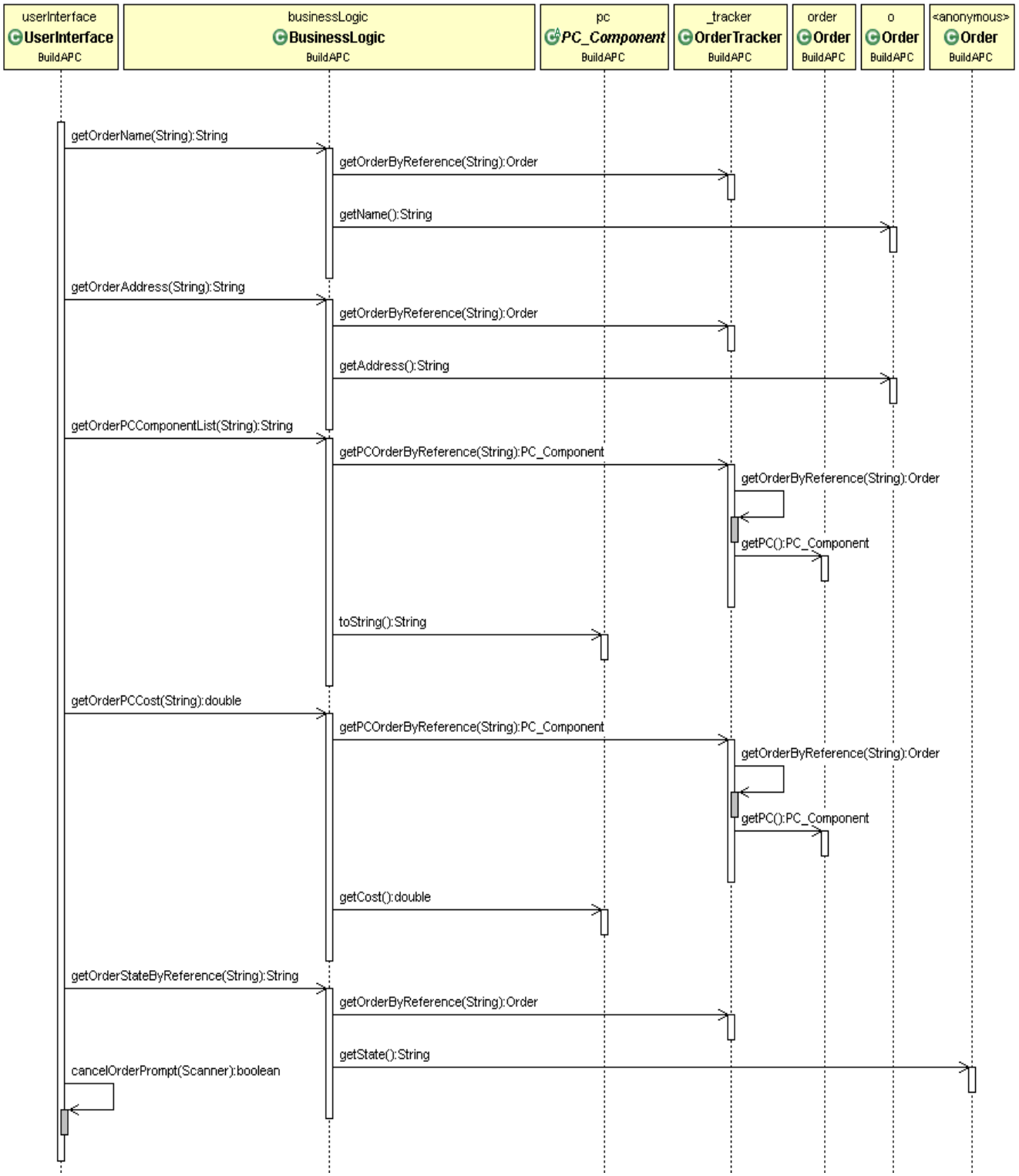
We investigated several of the architectural patterns from POSA1 (Buschmann 1996), POSA2 (Schmidt 2000), and POSA3 (Kircher 2004). One of the ones we investigated was the PAC (Presentation Acces Control(POSA2)). Unfortunately none of the architectural patterns we looked at would fit our requirements for the system, so no architectural pattern was integrated into the system.

Class Diagram (UML)



7Full Class Diagram. Higher Resolution will be available digitally

Interaction Diagram



8Sequence Diagram of Use Case: Check Order Status. Higher Resolution will be available digitally

Testing

For this project, we used Test-Driven Development. As we wrote classes, we simultaneously wrote unit tests for them. We used Junit 4.11 to perform the automated tests.

Test driven development is related to what was originally an aspect of Extreme Programming. Test Driven Development is generally performed by writing tests, writing code to pass those tests and finally refactor the code.

The requirement to print each method call gave us a clear image of order of execution and quickly highlighted issues.

Deployment

The product is intended to be web applet. Another alternative considered would be Java Web Start.

We chose a web applet because the requirements to run it are much lighter than JWS. JWS requires JRE being installed on the client's machine and the browser set up for JWS, the server ideally runs JWS with its own protocol: JNLP. The applet however only requires that JRE be installed on the clients machine and the server only needs a basic HTTP server. One issue though is that the applet executes within the internet browser and must share its memory allocation with it while JWS has its own allocation. However we felt that we had taken appropriate steps to minimize memory usage and that this was a minor issue.

Alternative design patterns were considered during the requirements analysis phase. For instance, instead of the Builder design pattern, we considered the use of the Abstract Factory design pattern. The use of the Abstract Factory could have been used to create different PC Factories which would have created, for example, different versions of low-, medium-, and high-end PCs. However we decided that the added control over how a PC was built was a greater advantage so we used the Builder. Another alternative design pattern we considered was using the Strategy instead of the State. However we soon realised that we were tending towards little functionality in this possible state or strategy, and that the use of States, not Strategies was the obvious choice.

Issues

Late in the development we realised that the functionality to create a PC from base components would not be realised. This was due to not wishing to violate the encapsulation of the FlyweightFactory by passing its list of possible components up to where a user could view and select them.

It was also decided that a GUI would be infeasible in the time remaining and a simple CLI was the interface implemented.

An Architectural Pattern to fit our design could not be found and was not implemented in the design or realization of the system.

It was more difficult than we envisioned to connect our two hierarchies of connected patterns,.

Final Analysis

In retrospect, we believe that in a real world situation, we would not have made the same decisions we made in this system. We believe this to be caused in some extent by the way with which the project was structured. By putting the priority on realising a minimum number of patterns and keeping with the “ilities”, it was clear that the practical functionality of the framework would suffer.

The use of Test Driven Development driven by JUnit was a very good help in finding implementation problems quickly and efficiently. If such problems arose, it was easier refactor our code to solve problems because of how we tested our classes and their methods.

We refactored our original design to produce an implementation that was easier to understand from the third party point of view and managed dependencies and “bad code smells".

We found implementing the Composite and Flyweight very useful because in effect they shared their concrete components, and the Builder could use them efficiently by using the abstract supertype of these classes.

It was more difficult than we had envisioned to integrate the subsystems to something a user could use, and we feel that we should have given this more thought during the design phase of the project.

References

Buschmann, F., Rohnert, H., Stal, M. (1996) *Pattern-oriented software architecture. Volume 1, a system of patterns*, J. Wiley and Sons: Chichester; New York; Brisbane; Toronto; Singapore.

Gamma, E. (1995) *Design patterns : elements of reusable object-oriented software*, Addison-Wesley: Reading, Mass.

Schmidt, D.C. (2000) Pattern-oriented software architecture. patterns for concurrent and networked objects, volume 2 Volume 2 [online], available: http://www.books24x7.com/marc.asp?bookid=3002 [accessed 23 Nov 2012].