

# OOSE200 Report

## Company Training Simulation

Connor Beardsmore - 15504319

### The Sacred Elements of the Faith

the holy  
origins

the holy  
structures

107	the holy behaviors						139	
FM Factory Method							A Adapter	
117	127					223	163	175
PT Prototype	S Singleton					CR Chain of Responsibility	CP Composite	D Decorator
87	325	233	273	293	243	207	185	
AF Abstract Factory	TM Template Method	CD Command	MD Mediator	O Observer	IN Interpreter	PX Proxy	FA Façade	
97	315	283	305	257	331	195	151	
BU Builder	SR Strategy	MM Memento	ST State	IT Iterator	V Vistor	FL Flyweight	BR Bridge	

## “Company Training Simulation”

### Polymorphism

Throughout the Company Simulator, polymorphism is extensively utilized to both generalise and decouple code, leading to increased testability. To allow for the use of polymorphism, both implementation inheritance and interface inheritance has been employed.

- Property - kept in map, polymorphically call calcProfit() via strategy
- Events + Plan - both use strategy so can call run() on parent class
- WageObserver list allows ANY class to become an observer if it implements

### Design Pattern Implemented

#### Factory Method Pattern

A Factory was employed to encapsulate object instantiation for both the Event and Plan subclasses.

#### Dependency Injection Pattern

The Dependency Injection pattern worked to remove all hard-coded dependencies, with the primary injector code being located in the main method.

#### Model View Controller Pattern

The MVC pattern was utilized for the overall layout of the system, due to its flexibility and its strong separation of concerns.

#### Observer Pattern

An observer was set up for WageEvents, allowing all relevant Property's to be updated easily by the notify method

#### Composite Pattern

company owns other companies, profit tree structure

#### Template Method Pattern

The Template Method pattern was used in the reading of the files. The common code for opening and closing files was kept in the superclass, with the subclasses implementing the protected abstract processLine() method.

#### Strategy Pattern

The run() method located in both Event and Plan subclasses is a form of the strategy pattern, with each subclass implementing this method differently. Also utilizing the Strategy pattern is the calcProfit() method in Property subclasses, as all Properties calculate profit differently.

### **Miscellaneous Patterns**

The use of for each loops throughout the system illustrate a form of the simplistic Iterator pattern. The objects used for file reading utilize the Decorator pattern, however these classes were used from the Java API.

### **Testability**

- Test cases!! sample outputs to clear up order ambiguity
- Factory + Dependency Injection allow for easy mocking of objects, low coupling
- Mad toString() and debug output methods
- clear and consise exception handling
- tested on heaps of invalid file types for all 3 input files

### **Alternative Design Choices**

Despite the design having a high level of testability and maintainability, there are alternative design choices that could have been employed.

- Iterators instead of for loops
- Controllers are easy to switch in and out, could have used one bunta controller
- could have used scanner instead of BufferedReader