Benjamin Palmer - Curtin University - 17743075

OOSE – COMP2003 Assignment Report

➔ “Training System for Company Directors”

Polymorphism - Discussion of Design Patterns

1. **Composite pattern** (tree-hierarchy of objects) – Company.
    a. This has been implemented on the Company class, in order to calculate profit over every class owned by that company, regardless of how many it owns. Because the "composite" class, Company, is only required to recursively search instances of itself, I saw it as redundant to have an interface or abstract class to represent each element.
    b. I have used a HashMap for general efficiency when you want to search for/remove a specific company/business unit by name or by the object type. These searches are also conducted in the same recursive manner as profit calculation. For example, it searches itself first for the required Property, otherwise recurses and does the same for every owned object until it has been found. This means that no matter how many levels deep the company or business unit to be found is, it will be found somewhere in the process, if it exists.

2. **Template method pattern** & generics (subclass has implementation; superclass knows when) - Reader.
    a. This is implemented in my TrainingFileReader and the 3 reader classes. The TrainingFileReader doesn’t need to know the type of file that it’s reading, as the calling class instantiates the correct type, indicated by the generic type. This class has a read method and a “protected abstract parse(…)”. The read method initialises the file reading objects, and then passes the BufferedReader object and the list of objects to add to, to the parse method. Parse is required to be implemented in any extending objects of TrainingFileReader. The parse methods all work in the same way, to read that specific file type and return the list containing objects of the indicated generic type. Each line in the reader files is also thoroughly error checked, for many possible error conditions.

3. **Factory method pattern** (strategy pattern with object creation delegated) – events.
    a. Once all files have been read, objects initialised and bank balances read for that year, various “events” can be performed on the different Properties in the simulation. Given that all 3 (6 if including +/-), events are similar in their implementations, especially Revenue/Value; I saw it as appropriate to implement them with the strategy pattern, using the IEvent interface and then have a factory method in Simulation. This factory method determines the type of event based on the event passed to it (a natural condition), and then creates the appropriate event to deal with that situation. The IEvent object then has the now-implemented “change” method called, and the calling method therefore needs to know no information about the object that it’s actually calling. This means other event types could easily be implemented as long as they implement IEvent, and the case for them to be used is also added to the factory method.

4. **Property and Subtypes** (not really a design pattern, but it is subtype polymorphism).
    a. Property represents the supertype of Company and BusinessUnit, containing the common functionality and class fields of each of them. Property is not all that useful by itself, but contains information critical to the subclasses. This means that other types of property could be added to the simulation without great changes, maybe an “InvestmentFirm” or “Government”.
    b. Other than this, this generic Property type means an operation can just be performed on a “Property”, it can then do a type check (instanceof) to determine exactly what needs to happen for the underlying subclass.

Testability

Following the SOLID model, and observability:

1. **Single Responsibility –** I've attempted to separate the responsibility of each method and class as much as a possible in the design of my assignment. As a few examples, the main method only calls other objects in order to run the program, and display of CompanyReport/s is delegated to DisplayReport. If a class only has a single responsibility, it is easier to determine what the correct outputs/results of that class will be.
2. **Open/Closed Principle –** As this is the first iteration of this piece of software, I don't find it too applicable to need to do this. As an example, I have confirmed the PropertyFileReader code does work, and in order to not make changes to this code, but still to extend functionality, new/different dependencies could be injected or the code "extend"ed, in order to do another task.
3. **Liskov Substitution Principle –** As I have done with the classes implementing the template method pattern (the three Readers), and also IEvent and the three event types, all of these different classes can have their methods called without the caller needing to know the exact object type. This makes it easy to mock dependencies, without changing expected behaviour. This allows individual methods to be tested in isolation, independent from the rest of the class.
4. **Interface Segregation Principle –** As done with IEvent, this interface does as little functionality as required, with only one method, meaning that it is unlikely IEvent will ever need to be changed. This reduces the complexity of testing as a whole too, as only one method from IEvent has to be tested, instead of a combination of methods on implementations, which could, potentially, have other effects.
5. **Dependency Injection –** wherever possible, I've used dependency injection in order to make testing on objects with dependencies, easy. Injected objects are easy to mock, and then I can easily test that specific method or class. For example, the "eventFactory", which imports an event model, and on return should return the correct type of IEvent. The event model can be mocked to provide various types of events when it's "getEvent" function is called, leading to easy testing of that particular method. Also, as far as I can tell, I've restricted use of "new", within methods, to JDK classes, and only created objects where absolutely needed or nearby to main().
6. **Observability –** Whilst not used, some of the classes do/could have accessors for data which otherwise shouldn't be visible to the rest of the simulation, but is useful for testing. An example is Simulation, where I have commented out code to return the lists of Events, Plans and Properties. This code, if used in a production sense, violates data-hiding principles, but can be useful in a testing scenario to validate function operations. Others I have removed since testing but left those get methods there as examples.

Alternative Design Choices

1. **Two possible alternative design decisions:**
    a. **Display company name and balances on every iteration** – unlike my implementation (explained it a bit inside the class itself), I could have printed each company report every time one was generated, meaning that a call to something like "printReport" would be made for every company, every year. How I've implemented this means that displaySimulation() is only called once, and doesn't lead to "junk"/invalid information being printed to the terminal when the program is in an exception state. I have, however implemented a way to still print these messages if requested – by changing the PRINT_ERROR constant in TrainingSystem.java to "true", which prints the received company reports until the exception condition.
    b. **Factory method pattern for file line error checking** – Another potential use of the factory method could have been to change the "errorCheck" functions, which I have in Plan/Event/PropertyReader.java and TrainingSystem.java. "errorCheck" imports an array of strings to perform many different error checking scenarios and throws an exception if it detects an issue.
        i. A few reasons why I could have implemented an "ErrorCheck": it would increase singular responsibility of the file reading classes, as they currently check for errors *and* populate a list based on data in the file. It would also make checking the functionality of the error checking methods, and separately the file reading methods, easier to do.
        ii. However, and the reason I didn't do this, is that I see it to be increasing design complexity by adding potentially 5/6 different classes, 1 for each file (4), one interface and maybe even a factory class – 5/6

more classes than I currently have. This also means many more objects have to be initialised, increasing memory usage and code complexity.

iii. My only other design consideration is that checking for errors, in a class that reads files, is critical anyway, so I believe that my decision to not use the factory pattern (delegating error checking to another class) is valid. To make sure the error checking method was easy to test, I didn't use a class field to store the current line of text to read, so I import that line into the errorCheck method.

2. **A few others I thought about too:**
   a. For the overall "architectural design pattern", I have decided to implement MVC, with an emphasis on being a non-interactive program, meaning the view will only display output to the user and receive no input other than the command line arguments, in a batch-processing way. The view contains the main and sets up the program. I **could have also used the basic IPO**(Input-Process-Output) model, however would have had to include the container/model classes with the process, and I don't see that as demonstrating good separation from container classes and (what would be) controller classes.
   b. Factory pattern for file reading – Instead of using the template method pattern to read files, an alternative would have been to use a factory, which determines the type of imported file, and then uses the created class to perform the same functionality as my current "parse" method. The reason I didn't do this is because I think it slightly increases design complexity, although the memory usage of my implementation would be marginally more; as every reader-type class in my implementation also has to create the read method every time a child class is initialised, where the factory pattern avoids this. I, however, believe factory increases complexity because it requires two additions to my implementation, that being an interface for the file readers, and a factory method in TrainingSystem.java or TrainingFileReader.java, or a class by itself.

Side note about calculating profits:

I saw it as a strange assignment specification choice that even the "primary companies" (companies without a named owner) have their profit split to the "unnamed owner", meaning that this applies to the *primary company too*, and on every year that the program runs, half the profits of the companies without a named owner, just disappear. I see this as an issue, and it can be fixed by implementing the following code to stop all "primary companies" from dropping half their profit on every year:

Lines 180 & 181 of Company.java (profitCalc()) would be replaced with:

```
// Determine how much of the profit is kept.
if(this.getOwner() == null)
{
     // This is the primary company.
     this.account.updateBalance(profit);
}
else
{
     /* This is a sub company, and profit is split to the owner too. */
     this.account.updateBalance(0.5*profit);
     profit *= 0.5;
}
```

Thanks for reading!