Package Design in Practice

Week 10 Lecture
Agenda

- FACTORY pattern
- Payroll package design
- Assignment 1 review
- Assignment 2 questions
FACTORY pattern

- What does DIP tell us?
- Violation of DIP?
  - Circle c = new Circle (origin, 1);
  - Harmless if the concrete class is very unlikely to change
  - In early development stage of an application, many concrete classes that are very volatile.
- FACTORY pattern allows us to create instances of concrete classes while depending only on abstract interface.
FACTORY pattern (cont)

- Problematic scenario

Figure 10.1 An app that violates the DIP to create concrete classes
FACTORY pattern (cont)

- Applying the FACTORY pattern to SomeAPP

---

**ShapeFactory.java**

```
public interface ShapeFactory{
    public Shape makeCircle();
    public Shape makeSquare();
}
```

**ShapeFactoryImp.java**

```
public class ShapeFactoryImp implements ShapeFactory{
    public Shape makeCircle(){
        return new Circle();
    }
    public Shape makeSquare(){
        return new Square();
    }
}
```

---

Figure 10.2 Shape Factory

Figure 10.3 Shape factory code fragment
FACTORY pattern (cont)

- Problems?
  - Every time we add a new Shape implementation we have to add a method to the ShapeFactory interface!

- Possible solution
  - Sacrifice some type safety
  - Supply one make method which takes a String parameter
FACTORY pattern (cont)

- ShapeFactory.java
  
  ```java
  public interface ShapeFactory{
    public Shape make (String shapeName) throws Exception;
  }
  ```

- ShapeFactoryImp.java
  
  ```java
  public class ShapeFactoryImp implements ShapeFactory{
    public Shape make (String shapeName) throws Exception{
      if (shapeName.equals ("Circle"))
        return new Circle();
      else if (SHAPEName.equals ("Square"))
        return new Square();
      else
        throw new Exception("ShapeFactory cannot creat " + shapeName);
    }
  }
  ```
FACTORY pattern (cont)

- Substitutable Factories

![Substitutable factory diagram]

Figure 10.4 Substitutable factory
FACTORY pattern (cont)

- Using Factory for test fixtures

Problem

- We have a Payroll application that uses a database
- We wish to test the function of the Payroll module without using the database

Figure 10.5 PayrollTest spoofs Database
FACTORY pattern (cont)

- How does Payroll get the instance of PayrollTest it uses as the Database?

Figure 10.6 Spoofing the Factory
Payroll Package Structure

Figure 10.7 Initial Payroll system package design
Initial package design analysis

- Dependency is clear
- No cycles
- What happens if Classification package is modified?
- Transactions package do not share the same closure
- Isn’t PayrollApplication package too sensitive?
Applying the Common-Closure Principle

![Diagram of a closed package hierarchy for the Payroll application](image)

Figure 10.8 A closed package hierarchy for the Payroll application
Applying the Common-Closure Principle

- **PayrollDomain** package
  - Contains the essence of the whole system, yet depends on nothing
- **Classification** package
  - Changes are isolated
- The bulk of the executable code is in packages that have few of no dependents.
- Difference between first and second attempt
Applying the Reuse-Release Equivalency Principle

- What portions of the payroll application can be reused?

- Policy might be different
  - Not likely to be reused: Classifications, Methods, Schedules, Affiliations
  - Likely to be reused: PayrollDomain, PayrollApplication, Application, PayrollDatabase, PDImplementation

- Write software to analyze the current employee database
  - Likely to be reused: PayrollDomain, Classifications, Methods, Schedules, Affiliations, PayrollDatabase, PDImplementation
Applying the REP (cont)

- **Does** PayrollDomain **conform to CRP?**
  - Is Transaction an outlier in that package?
  - Still we want to separate the transactions from the elements that they manipulate
  - Move the Transaction into a new package TransactionApplication
  - PayrollApplication is no longer reusable.
Applying the REP (cont)

Figure 10.9 Updated Payroll Package Diagram
Coupling and Encapsulation

- Classification package
  - HourlyClassification
  - CommissionedClassification
  - SalaredClassification
  - TimeCard
  - SalesReceipt
- Which classes should be kept private?
Coupling and Encapsulation (cont)

- Isn't **TimeCard** created by **TimeCardTransaction**?
- Slight modification will remove the dependency.

Figure 10.10 Revision to TimeCardTransaction to protect TimeCard Privacy
Metrics

- You can’t manage what you can’t control, you can’t control what you don’t measure!
- (H) Relational Cohesion
  - $H = (R + 1)/N$
- (I) Instability measure
- (A) Abstractness measure
- (D) Distance from the main sequence
- (D’) Normalized distance from the main sequence
Metrics

- Tools that will calculate quality metrics
  - JDepend (Open source software)
    - [http://clarkware.com/software/JDepend.html](http://clarkware.com/software/JDepend.html)
    - JDepend traverses Java class file directories and generates design quality metrics for each Java package
      - Afferent Couplings (Ca)
      - Efferent Couplings (Ce)
      - Instability (I)
      - Abstractness (A)
      - Distance from the Main Sequence (D)
      - Package Dependency Cycle
  - Write Shell script to walk through source files and do the calculation
    - Sample shell script that walks through C++ source code
      - [http://www.objectmentor.com/resources/downloads/bin/depend.zip](http://www.objectmentor.com/resources/downloads/bin/depend.zip)
Metrics (cont)

Figure 10.11 Package Diagram with Metrics
Object factory

Figure 10.12 Object Factory for Transactions
Object factory – initializing the Factories

![Object Factory Diagram]

Figure 10.13 Static and dynamic structure of Main program and object factories
Rethinking the cohesion

- Should we separate Classification, Methods, Schedules, and Affiliations or should we merge them?
- Keep package diagram as simple as is practical.
The Final Package Structure

Figure 10.14 Final package structure