GoF patterns I

Week 8 Lecture
Sep 13, 2006

Agenda
- Iteration 2 requirements
- Adapter pattern
  - Delegation vs. inheritance
- Factory pattern
- Singleton pattern
- Strategy pattern

Iteration 2 Requirements
- POS project
  - Support for variations in third party external services
  - Complex pricing rules
  - A design to refresh a GUI window when the sale total changes
- Recap of iteration 1 requirements:
  - Implement a basic, key scenario of the Process Sale use case
  - Implement a Start Up use case as necessary to support the initialization needs of the iteration
  - No collaboration with external services
  - No complex pricing rules are applied

Iteration 2 Requirements
- Monopoly game project:
  - Basic, key scenario of the play Monopoly Game use case: players moving around the squares of the board. And as before, run the game as a simulation requiring no user input, other than the number of players.
  - Each player receives $1500 at the beginning of the game.
  - When a player lands on the Go square, the player receives $200
  - When a player lands on the Go-To-Jail square, they move to the Jail Square
  - When a player lands on the Income-Tax square, the player pays the minimum of $200 or 10% of their worth
What is design pattern

- Design patterns are tried and tested ideas for recurring design problem.
- Not readily coded solution, but rather the solution path to a common programming problem
- Design or implementation structure that achieves a particular purpose

When (not) to use design pattern

- Delay
- Don’t use design pattern prematurely
- First implement something that works then improve the weakness
- Design patterns may increase or decrease the understandability of a design or implementation
- The canonical design pattern reference is the “gang of four” book “Design Patterns: Elements of Reusable Object-Oriented Software” by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.

Four essential elements of a pattern

- The pattern name
- The problem
  - When to apply the pattern
- The solution
  - The elements that make up the design
- Consequence
  - The results and trade-offs of applying the pattern

Gang of Four Patterns (GoF)

- Famous and influential book about design patterns
- Recommended for students who wish to become experts
- We will cover the most widely-used patterns from the book
GoF design patterns

- 23 design patterns
  - Creational patterns
    - Abstract the instantiation process
    - Make a system independent of how its objects are created, composed and represented
  - Structural patterns
    - How classes and objects are composed to form larger structures.
  - Behavioral patterns
    - Concerns with algorithms and the assignment of responsibilities between objects

Adapter (structural pattern)

- Name
  - Adapter
- Problem
  - How to resolve incompatible interfaces, or provide a stable interface to similar components with different interfaces
- Solution
  - Convert the original interface of a component into another interface, through an intermediate adapter object

Adapter in POS project -- requirements

- NextGen PoS system needs to communicate with several external third-party services
  - Tax calculators, credit authorization services, inventory systems, accounting systems.
  - Each has a different API and can not be changed.

Adapter in POS project -- reality

- Consider the TaxCalculator services
- Suppose the POS system will be installed through out the country
  - Each state has its own way of calculating and collecting tax
    - California: 8.5% on almost everything
    - Mass: 5% on most items except grocery
  - Each state has its own TaxCalculator services
    - California API: List getTaxes(List allItem)
    - Mass API: Set computeTaxes(Set allItem)
- Tax laws change annually, at all government levels
- ID | Rule | Changeability | Source
  | RULE1 | Tax rules. Sales require added taxes. (POS domain) | High. Tax laws change annually, at all government levels | law
Adapter in POS project – responsibility assignment

- Which object should be responsible for collecting tax

```
method public void getTotal(){
  int tot = 0;
  for each SalesLineItem sli:
    tot = tot + sli.getSubtotal();
  return tot;
}
```

Adapter in POS project – solution path

- How to make the Sale object decoupled from detailed Tax Calculation services
  - Reason: Sale is interested in getting the tax for each taxable item, not how taxes are calculated in different states.
  - Solution:
    - Sale defines an interface to get results on taxes.
      - ITaxCalculatorAdapter
    - Any Tax Calculation Service that does not use this interface needs to find an interpreter (Adapter) to do the translation.
      - MassTaxAdapter, CATaxAdapter

Adapter in POS project – first attempt

- Novice design of Sale.getTaxes method

```
double getTaxes(){
  switch (state){
    case CA:
      CATax catax = new CATax();
      List<taxLineItem> = catax.getTaxes(lineitems);
      break;
    case MA:
      MATax matax = new MATax();
      Map<taxLineItem> = matax.computeTaxes(new HashSet(lineitems));
      break;
  }
}
```

Adapter in POS Project -- solution

- Process for making Sale object decoupled from detailed Tax Calculation services
  - Sale defines an interface to get results on taxes.
    - ITaxCalculatorAdapter
  - Any Tax Calculation Service that does not use this interface needs to find a interpreter (Adapter) to do the translation.
    - MassTaxAdapter, CATaxAdapter
Object discover during design

Sales

- dateTime
- ... contains

- Sales LineItem
- quantity

Tax LineItem

description, percentage, amount

A general example of Adapter -- problem

Adaptation to a standard interface

Target (Shape)

- defines the domain-specific interface that Client uses.

Adapter (Circle)

- adapts the interface to the Target interface.

Adaptee (XXCircle)

- defines an existing interface that needs adapting.

Client (ShapeApp)

- collaborates with objects conforming to the Target interface.

A general example of Adapter -- solution

Target (Shape)

- ... contains

Adapter (Circle)

- ... contains

Adaptee (XXCircle)

- ... contains

Client (ShapeApp)

- ... contains

Adapter patter: structure

- Target (Shape)
  - defines the domain-specific interface that Client uses.

- Adapter (Circle)
  - adapts the interface to the Target interface.

- Adaptee (XXCircle)
  - defines an existing interface that needs adapting.

- Client (ShapeApp)
  - collaborates with objects conforming to the Target interface.
Different implementations of Adapter

Object Adapter and Class Adapter

Object Adapter
- Relies on object composition to achieve adapter

Class Adapter
- Relies on class inheritance to achieve adapter

Two reuse mechanisms

Inheritance and Delegation
- Inheritance: reuse by subclassing;
  - "is-a" relationship
- Delegation: reuse by composition;
  - "has-a" relationship

Rule of thumb – design principles #1
- Favor object composition over class inheritance
  [GOF book pg 18-21]

Inheritance approach

Import Java.util.Hashtable;
class AddressBook extends Hashtable{
  String put(String name, String address){
    return (String) super.put (name, address);
  }
}

// Problem with the inheritance approach
public static void main(String[] argv){
  AddressBook contact = new AddressBook();
  contact.put("comp5028", new Exception());
}
Delegation approach

```java
class AddressBook {
    private Hashtable<String, String> table;

    AddressBook() {
        table = new Hashtable();
    }

    String put(String name, String address) {
        return table.put(name, address);
    }
}
```

**Inheritance vs. Delegation**

- **Benefits of delegation**
  - **Extensibility:** we can change the internal representation of `AddressBook` to another class
  - **Subtyping:** `AddressBook` can not be substituted for a `Hashtable` in any of the client code.

**Abstract Factory (creational pattern)**

- **Name:** Abstract Factory
- **Problem:** who should be responsible for creating families of related objects.
- **Solution:** Use a factory object to handle all creation responsibility

**Abstract Factory (creational pattern)**

- **Name:** Abstract Factory
- **Problem:** who should be responsible for creating families of related objects.
- **Solution:** Use a factory object to handle all creation responsibility

**Factory in POS project**

- **Problem:**
  - Now we have a series of adapters for all sorts of different external services
  - Who should be responsible for creating the correct set?

```java
class AddressBook {
    private Hashtable<String, String> table;

    AddressBook() {
        table = new Hashtable();
    }

    String put(String name, String address) {
        return table.put(name, address);
    }
}
```
Factory in POS project

- Suppose if this POS is deployed in some store of MA, we’ll need
  - MATaxAdapter,
  - GreatNorthernAccountingAdapter, ...
- If it is deployed in CA, we’ll need
  - CATaxAdapter, SAPAccountingAdapter

Factory pattern in POS project

- We need several factory objects each will be responsible for creating a set of objects.
  - A MAFactory which will create MATaxAdapter, GreatNorthernAccountingAdapter and so on
  - A CAFactory which will create CATaxAdapter, SAPAccountingAdapter and so on
  - Naturally we’ll have an abstraction which is an Abstract Factory

Abstract Factory: structure

- AbstractFactory (ServiceFactory) declares an interface for operations that create abstract products.
- ConcreteFactory (MAFactory, CAFactory) implements the operations to create abstract product objects.
- AbstractProduct (ITaxCalculatorAdapter, IAccountingAdapter) declares an interface for a type of product object.
- Product (MATaxAdapter, CATaxAdapter, SAPAccountingAdapter, GreatNorthernAccountingAdapter) implements a product object to be created by the corresponding concrete factory implementing the AbstractProduct interface.
- Client (Store) uses interfaces declared by AbstractFactory and AbstractProduct objects.
Alternative solution using external configuration file

```java
ServicesFactory

accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter

getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter

note that the factory methods return objects typed to an interface rather than a class, so that the factory can return any implementation of the interface

if ( taxCalculatorAdapter == null )
{
    // a reflective or data-driven approach to finding the right class: read it from an external property.
    String className = System.getProperty( "taxcalculator.class.name" );
    taxCalculatorAdapter = (ITaxCalculatorAdapter) Class.forName( className ).newInstance();
}
return taxCalculatorAdapter;
```

### Singleton (creational pattern)

- **Name:** Singleton
- **Problem:** Exactly one instance of a class is allowed – it is a “singleton”. Objects need a global and single point of access
- **Solution**
  - Make the class itself responsible for keeping track of its sole instance.
  - Ensure that no other instance can be created.
  - Provide a way to access the instance.

### Singleton in POS project

- ServicesFactory keeps instances of ITaxCalculatorAdapter, IAccountingAdapter, ...
- We only need ONE instance of everything for the whole application, hence ONE instance of ServiceFactory.
- Lots of objects might require using those instances.
- How to manage the visibility?
Singleton in POS project

```
Singleton in POS project

// Singleton, i.e., can only have one accounting adapter instance
AccountingAdapter accountingAdapter = getAccountingAdapter();
```

Implementation and design issues

```
Lazy initialization might have problem in multithread application

public static synchronized ServicesFactory getInstance()
{
    if (instance == null)
    {
        // critical section if multithreaded application
        instance = new ServicesFactory();
    }
    return instance;
}
```

```
Eager initialization alternative

public class ServicesFactory
{
    private static ServicesFactory instance = new ServicesFactory();
    public static ServicesFactory getInstance()
    {
        return instance;
    }
}
```

```
Creation work is avoided if the instances is never actually accessed
```

```
The getInstance lazy initialization sometimes contain complex and conditional logic
```

```
However, not necessary in Java at all!
```

External services with Varying interfaces problem

```
Pattern as a shared design vocabulary
```

```
“To handle the problem of varying interfaces for external services, lets use Adapters generated from a Singleton Factory”
```
POS project requirement

- Supporting complex pricing logic
  - Store-wide discount for the day, senior citizen discount, and so on
- Pricing strategy for a sale can vary
  - 10% off all sales
  - $10 off if the sales total is greater than $200
  - Others

Strategy in POS

Strategy (behavioral pattern)

- Name: Strategy
- Problem: How to design for varying, but related, algorithms or policies? How to design for the ability to change these algorithms or policies
- Solution: Define each algorithm/policy/strategy in a separate class, with a common interface.
Strategy: generic structure

- Strategy (ISalePricingStrategy) declares an interface common to all supported algorithms, allowing users to select an algorithm defined by a ConcreteStrategy.
- ConcreteStrategy (PercentDiscountPricingStrategy, AbsoluteDiscountOverThresholdPricingStrategy) implement the algorithm using the Strategy interface.
- Context (Sale) is configured with a ConcreteStrategy object that may define an interface that lets Strategy access its data.

Creating a Strategy

- Reconsider the Tax Calculator problem.
- The abstract ITaxCalculatorAdapter interface and all concrete adapters represent a strategy pattern.
- Factory object is used to create different adapters.
- We can use Factory to create different strategies.

Strategy and Context

Creating a Strategy

No caching of concrete strategy in PricingStrategyFactory, as opposed to ServicesFactory.

```java
// PricingStrategyFactory

PricingStrategyFactory instance; // PricingStrategyFactory
getSalePricingStrategy(); // ISalePricingStrategy
getSeniorPricingStrategy(); // ISeniorPricingStrategy

// PricingStrategy

String className = System.getProperty("salepricingstrategy.class.name");
strategy = (ISalePricingStrategy) Class.forName(className).newInstance();
return strategy;
```
Creating a strategy

- Reading and initializing the percentage value
  - If percentage discount changes frequently, and the current discount is stored in some database, who should read the current store discount?
  - Use the factory object to read and initialize the percentage value

```
if percentage discount changes frequently, and the current discount is stored in some database, who should read the current store discount?
use the factory object to read and initialize the percentage value
```

```
Creating a strategy

if percentage discount changes frequently, and the current discount is stored in some database, who should read the current store discount?
use the factory object to read and initialize the percentage value
```