Design Patterns II

Technion – Institute of Technology
236700
Spring 2017

Author: Gal Lalouche
Visitor: Motivation

Problem: Account management application

- We have several types of accounts
  - Guest account
  - Basic account
  - Gold/VIP Account
  - Developer Account
  - Etc.
- All accounts extend a common base class or implement a common interface
- Each type of account has different properties associated with it:
  - CSS/Theme
  - Access to special areas in the application
  - Display of ads
  - Etc.
Visitor: Alternatives I

- Add the methods, e.g., `adsDisplayed()`, `getTheme()` to the base class/interface, and implement in children

- Pros:
  - Basic use of polymorphism – this is what OOP is for!
  - Client code is very simple

- Cons:
  - As usual, increased coupling and reduced cohesion
    - Should an account really know the theme associated with it, or what it can access?
    - Can get ridiculously large as new features are added
  - Adds unnecessary edges to the **Web of Objects**
    - i.e., unnecessary dependencies to `Account`
  - Need to modify the classes (parent and children)
Visitor:
Language Envy I

• If we had Multiple Dispatch (AKA MultiMethods)...

```java
interface AccountHandler {
    public void handle(Account m);
}

class ThemeChanger implements AccountHandler {
    public void handle(BasicAccount a) { ... }
    public void handle(GoldAccount a) { ... }
    public void handle(DevAccount a) { ... }
}

class AdDisplayer implements AccountHandler {
    public void handle(BasicAccount a) { ... }
    public void handle(GoldAccount a) { ... }
    public void handle(DevAccount a) { ... }
}
```
Visitor: Language Envy II

- If we had Algebraic Data Types and Pattern Matching

```haskell
data account =
  BasicAccount { ... } |
  GoldAccount { ... } |
  DevAccount { ... } |
  ...
```

-- the compiler can check if we missed anything
displayAds (BasicAccount ...) = ...
displayAds (GoldAccount ...) = ...
displayAdds (DevAccount ...) = ...

Visitor: Alternatives II

What about **faking** it?

```java
AdDisplayer displayer = new AdDisplayer();
if (a instanceof BasicAccount)
    displayer((BasicAccount)a);
else if (a instanceof GoldAccount)
    displayer((GoldAccount)a);
else if (a instanceof DevAccount)
    displayer((DevAccount)a);
else
    throw new RuntimeException("no match");
```

- Totally **not** typesafe
- Adding new handlers is **hard and error-prone**
- Forgetting an account type fails at **runtime** (or worse, **doesn't fail** at all!)
  - We can't check at compile time that we've covered all the cases, that we have no repeats, or even that all of our classes below to the same hierarchy!
  - Adding new types of accounts would break all existing code with **no warnings**
- Lots of **code duplication** on the client side
  - We could move this logic to the AdDisplayer, but that only hides the problem
Visitor: Solution

Double dispatch:

• A way to emulate Multiple Dispatch in Java, C++, etc.
• When a method of object \( A \) is called with parameter \( B \), it calls a method on \( B \) with itself as the parameter
  • Methods written in class \( A \) have more information about the type of \textit{this} than clients who use an interface

```java
class Base {
  // in this context, the static type of \textit{this} is Base
}
class Derived extends Base {
  // in this context, the static type of \textit{this} is Derived
}
```

• We will use a combination of \textit{dynamic dispatch} and \textit{static binding} to emulate \textit{multiple dispatch}
interface Account {
    void accept(Visitor v);
    // any other methods you may want
}

interface AccountVisitor {
    void visit(BasicAccount a);
    void visit(GoldAccount a);
    // repeat for all types of accounts
    // important: no visit(Account a) method! (why not?)
}

class BasicAccount implements Account {
    @Override public void accept(AccountVisitor v) {
        // the static type of this is BasicAccount
        // so the compiler knows (statically) which method to bind to
        v.visit(this);
    }
    // any other methods you may want
}

class BasicAccount implements Account {
    @Override public void accept(AccountVisitor v) {
        v.visit(this);
    }
    // any other methods you may want
}
class ThemeChhanger implements AccountVisitor {
    void visit(BasicAccount a) {
        // ThemeChhanger specific code for BasicAccount
    }
    void visit(GoldAccount a) {
        // ThemeChhanger specific code for GoldAccount
    }
    void visit(DevAccount a) {
        // ThemeChhanger specific code for DevAccount
    }
}

class AdDisplayer implements AccountVisitor {
    void visit(BasicAccount a) {
        // AdDisplayer specific code for BasicAccount
    } 
    void visit(GoldAccount a) {
        // AdDisplayer specific code for GoldAccount
    }
    // etc.
} 

// client code doesn't care about the dynamic type
AccountVisitor visitor = new AdDisplayer();
account.accept(visitor);
Visitor: Follow up questions

- Why do we have to write the **exact same code** in all of the children? Why not implement it just **once** in the **parent**?
  - Java has **static** binding: method's signature is decided at **compile** time
    - Not to be confused with **dynamic dispatch**: the exact implementation of the bound method depends on the dynamic type
  - The parent class/interface doesn't know which method to invoke
    - In the context of the parent class, the **static** type of **this** is `Account`, but that's **not enough information** for the visitor
- Do we have to use **overloads**?
  - Overloads are just **syntactic sugar**
  - We could have named our methods `visitBasicAccount`, `visitGoldAccount`, etc.
  - While this naming pattern isn't common in Visitor implementations, it has the added bonus of being more **declarative**
  - It's also a way to implement the Visitor pattern in languages that don't support overloading, e.g., Python
Visitor: Return Values

It's easy to allow visitors to return values:

```java
interface AccountVisitor<? extends T> {
    T visit(BasicAccount a);
    T visit(GoldAccount a);
}
interface Account {
    <T> T accept(AccountVisitor<? extends T> v);
}
class BasicAccount {
    @Override public <T> T accept(AccountVisitor<? extends T> v) {
        return v.visit(this);
    }
}
// example usage
class ThemeChoose implements AccountVisitor<Theme> {
    Theme visit(BasicAccount a);
    Theme visit(GoldAccount a);
}
// only null is an instance of Void
class AdsDisplayer implements AccountVisitor<Void> {
    Void visit(BasicAccount a);
    Void visit(GoldAccount a);
}
```
Visitor: Applicability

Use the Visitor pattern when:

• You want to add **new virtual methods** to a class hierarchy **without** actually **modifying** any of the classes directly

• We want the classes to remain **cohesive** while **robust** and **extendible** (open-close principle)

• You don’t want the client to **check** (or even be **aware** of) the **dynamic type**

• **Obliviousness** once again: clients are oblivious to the **dynamic type**, visitables are oblivious to how they are **extended with visitors**.
Visitor: Implications

✓ Adding behaviours (visitors) to existing classes is easy
✓ Elements stays “untouched”
X There is quite a bit of boilerplate involved
  ✓ But not for the client
  ✓ Can be auto-generated by the IDE
X Adding a new account type is tedious
  X Need to modify all existing Visitors
  ✓ But it will be checked in compile time
X There is some code duplication in the Account classes
  ✓ But this code will never change
X Beware of programming against concrete classes instead of abstract (i.e., no default methods) interfaces!
X Subtyping and overloading can have surprising results
X For example, forgetting to override accept / calling the wrong visit method
Singleton: The Anti-Pattern

Problem:

• Sometimes it make sense to have only one instance of a class
  • Guice modules
  • File system
  • GUI widgets
  • Server connections

• We might want to have global functionality available throughout our program without resorting to passing objects around
  • Utility functions
  • Logs
Singleton: Motivation

Motivating example:

- We are writing a chat application
- There's a big screen (and only one screen) where we have to print message to
- Can support other functionality
  - clicking on it
  - changing font
  - changing color
Singleton: Static methods?

- Using `static` methods (The `class` object is actually a singleton)

```java
class Screen {
    public static void printMessage(String message) {...}
    public static void changeColor(Color c) {...}
}
```

- Problems:
  - Can’t be used as an object (i.e., passed as a parameter)
  - Can’t use inheritance
  - Can’t program against an interface (i.e., can’t `implement`)
  - All dependencies needs to be known at compile-time
    - Although it's possible to call an `init` method if you really, really need to
  - Testing is a pain, can't easily with mocks
  - **You should avoid public static methods except for object creation!**
Singleton: Take I (classic)

- Ensures only one instance of the class with a private constructor

```java
class Screen {
    private static Screen instance;

    private Screen() {...}

    public static Screen getInstance() {
        if (instance == null)
            instance = new Screen();
        return instance;
    }

    public void printMessage(String message) {...}
}
```

- Can use a `static init(...) method` to pass parameters and check that this method was called only once in the `getInstance()` method
- Obviously not typesafe
- What happens in a `multithreading` environment?
Singleton: Take II

Use lazy synchronization (also called double checking)

```java
class Screen {
    private static Screen instance;

    private Screen(){...}

    public static Screen getInstance() {
        // save synchronization overhead
        if (instance == null) {
            synchronized(Screen.class) {
                if (instance == null) {
                    instance = new Screen();
                }
            }
        }
        return instance;
    }
}
```
Singleton: Take III

- Use the static constructor:

  ```java
class Screen {
    // can replace with getInstance() if you want to
    public final static Screen instance = new Screen();

    private Screen(){...}
  }
```

- The instance is initialized during **class loading** in the **JVM** 😊
- Object might be loaded even if it is **not used** 😞
- Can’t pass parameters 😞
Singleton: **enum** alternative

- Since Java 1.5 there’s a simpler alternative using **enums**

```java
public enum Screen {
    INSTANCE;

    public void printMessage(String m) {...}
}
```

- **Pros:**
  - *Simpler* code
  - *Serialization* is free
  - Can be used as an object (i.e. passed as a parameter)
  - Can implement an interface

- **Cons:**
  - Can’t use *inheritance* (but we do get *default methods*)
  - Can’t pass parameters
Singleton: OOP Loophole

- Singleton looks and feels very “object-oriented”
  - We’re using *constructors*, *static* fields, *visibility modifier* and even *enums* to ensure correct behaviour
- However, we have an object that is accessible from anywhere in the code
- Essentially, a **Global Variable!**
  - Doesn’t get less object-oriented than that...
  - All the bad properties of global variables apply to singletons
- All of our design patterns revolved around *obliviousness*, *reduced coupling*, and *polymorphism*: Singleton goes against all of that
  - Clients *explicitly* invoke a Singleton's methods; they are *tightly coupled* to it; there can be no *polymorphism*
Singleton: How deep does the rabbit hole goes?

The Singleton pattern has many pitfalls:

• Lifecycle is **Global** – It’s not (necessarily) **lazy**, and it’s never **collected** by the Garbage Collector
• It opposes the principle that entities should exist in the smallest possible context
• Testing **nightmare** – tests are no longer **independent**
• Most implementations are impossible to sub-class (and rightfully so)
• Classes using Singletons basically **hide their dependencies**
  • Singletons actually make other classes harder to test!
  • Can be a **pain** to replace with mocks
• What happens if after a while you find out you **actually do need** 2 instances of the Singleton?
Singleton: How to fix it?

There are two rather obvious fixes to these problems:

- **Dependency injection**
  - This means that the singleton has to **implement an interface** (or extend a base class)
  - Client is **oblivious** to singleton-ness
  - Can be **replaced** later, can be **mocked**, all the good stuff

- **Immutability**
  - Global **functionality** is less dangerous than global **variability**
  - Unfortunately this can't always be achieved
    - In such cases, try to limit the number of **mutable interfaces** (e.g., using the **façade** pattern)
Singleton: Freebies

- There is a special case of singletons that are "free", i.e., don't have any pitfalls
- If your classes has no injected dependencies (i.e., only a default constructor) and is immutable, it might as well be a singleton
  - Note that this is unrelated to the singleton pattern: we don't have to limit ourselves to a single instance, we just can
- Possible examples:
  - Factories
  - Visitors
  - Observers (but not observables!)
- More common in languages with less verbose Singletons, e.g. Scala, Kotlin
- You should still inject these "free" singletons!
Design Patterns: Programming Languages

- Design patterns exist in the context of the **hosting language**
  - Usually, it is a feature **missing** in the language
  - We need **factories** because we can't **override constructors**
  - We need **visitors** because we don't have **multiple dispatch** or **algebraic data types**
- Some patterns "graduate" to become part of the language
  - **Kotlin** and **Scala**: flexible, easy to use Singletons called **objects**
  - **C#**: built in **Observers** called **events**
  - Originally **Iterator** was a design pattern, but now it's an integral part of any modern language
- Suggested Readings:
  - **Design Patterns by GoF** (the book that started it all)
  - **Wikipedia** has a lot of information, including java implementations
Design Patterns: The perils of over-design decadence

- All of our design patterns had something in common: reduce coupling, increase cohesion, use of polymorphism
- But design patterns usually have a cost associated with them
  - Reduced code locality, increase in number of classes, code is harder to reason about
  - TANSTAAFL
- If you're not going to need polymorphic functionality, then you shouldn't bother to implement it
  - i.e., YAGNI (or DTSTTCPW)
  - Waste of development time
  - Increased in code complexity
- FizzBuzz Enterprise edition