CS 430 - Lecture 19 - Design Patterns

Collin Roberts

December 14, 2023

Outline

1. Design Patterns
   1. Introduction
   2. Adapter Design Pattern (§8.6.2)
   3. Bridge Design Pattern (§8.6.3)
   4. Iterator Design Pattern (§8.6.4)
   5. Abstract Factory Design Pattern (§8.6.5)
   6. Categories of Design Patterns (§8.7)
   7. Strengths/Weaknesses of Design Patterns (§8.8)
2. Re-Use During Post-Delivery Maintenance

Unlike Library (Toolkit) and Application Framework from last lecture, Design patterns **assume the OO paradigm**.

Definition 1

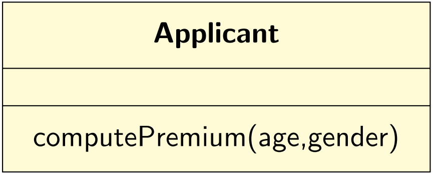
A **design pattern** *is a solution to a general design problem, in the form of a set of interacting classes that have to be customized to create a specific design.*

1. **What is Re-Used:** relationships among classes (usually expressed as a class diagram)
2. **What is New:** details within each class (usually a new class diagram, with the generic classes from the previous diagram replaced by classes tailored to the specific problem to be solved)

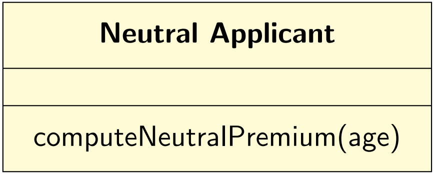
Motivation: FLIC Example (§8.6.1)

1. Until recently, premiums at Flintstock Life Insurance Company (FLIC) depended on both the age and the gender of the applicant for coverage.
2. FLIC has recently decided that some policies will now be gender-neutral. That is, the premiums for those policies will depend solely on the age of the applicant.

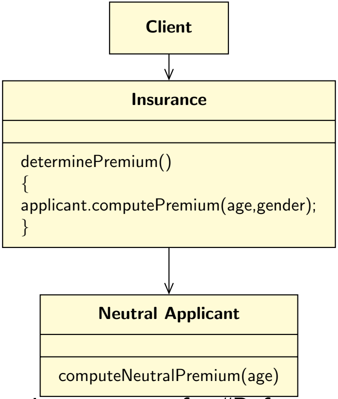
3. The old computation of premiums used this class:



4. The new computation of premiums will use this class:



5. However there has not been enough time to change the entire system. The situation is displayed in the following figure (Fig 8.4 in the text).



Client

Insurance

```
determinePremium()
{
applicant.computePremium(age,gender);
}
```

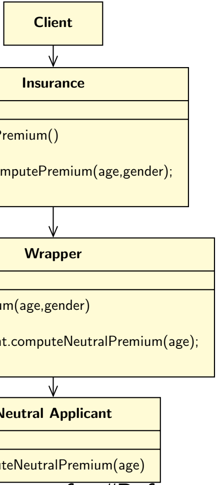
Neutral Applicant

computeNeutralPremium(age)

Notation: → for “References”.

6. Note the three interface problems with the bottom reference in the above diagram:
    1. Insurance calls the Applicant class instead of the NeutralApplicant class.
    2. Insurance calls the computePremium method instead of the computeNeutralPremium method.
    3. The parameters passed are age and gender, instead of age alone.

7. To solve these problems, we interpose the `Wrapper` class, as shown in this diagram (Figure 8.5 in the text):



Client

Insurance

```
determinePremium()
{
wrapper.computePremium(age,gender);
}
```

Wrapper

```
computePremium(age,gender)
{
neutralApplicant.computeNeutralPremium(age);
}
```

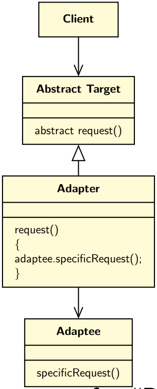
Neutral Applicant

```
computeNeutralPremium(age)
```

Notation: ⟶ for "References".

The Adapter Design Pattern

1. Generalizing the **Wrapper** construction above leads to the Adapter Design Pattern (Figure 8.6 in the text):



Notation: ⟶ for “References”.

Definition 2

An **abstract class** *is a class which cannot be instantiated, but which can be used as a base class for inheritance.*

Example: `Abstract Target` in the Adapter Design Pattern is an abstract class.

Definition 3

An ***abstract method*** *is a method which has an interface, but which does not have an implementation.*

Example: In the Adapter Design Pattern, `Abstract Target` class, `request()` is an abstract method. Usually abstract methods live inside of abstract classes.

2. Abstract methods are implemented in subclasses of the abstract class.
3. The abstract `request` method from `Abstract Target` is implemented in the (concrete) subclass `Adapter`, to invoke the `specificRequest` method in `Adaptee`.
4. This solves the interfacing problems from earlier. This is the raison d'être for the Adpater design pattern.

5. But the pattern is more powerful than that. It provides a way for an object to permit access to its internal implementation in such a way that clients are not coupled to the structure of that internal implementation. In other words, it provides the benefits of **information hiding** without having to actually hide the implementation details.

See the Lecture Notes.

See the Lecture Notes.

1. Creational, e.g. Abstract Factory
2. Structural, e.g. Adapter, Bridge
3. Behavioural, e.g. Iterator, Mediator

See Figure 8.12 in the text for the complete list of 23 documented by Gamma, Helm, Johnson and Vlissides.

Strengths

1. promote re-use by solving a general design problem,
2. provide high-level documentation of the design, because patterns specify design abstractions,
3. may already have implementations written, and
4. make maintenance easier for programmers who are familiar with the patterns.

Weaknesses

1. lack a systematic way to determine when patterns should be applied,
2. often require multiple patterns together, which is complicated, and
3. are incompatible with the Classical paradigm.

1. As we have seen throughout the course, an improvement in S/W methodology has a bigger payoff in maintenance than it does in development. This is true for the technique of re-use also:
    1. Reusable components are well designed, thoroughly tested, well documented and independent. These are the features of low maintenance S/W.
    2. Reusable components do not cause problems during maintenance.