

The need for design patterns

Lecture 13

Introduction to patterns

- Adapters are very commonly used. That's why so many variants has been proposed, and their advantages and disadvantages are so well studied.
- There are many other similar commonly used techniques for object-oriented program designs.
- In this lecture, we discuss how these commonly used techniques are specified, and what we can expect from a description of them.

References:

- Design Patterns: Section 1.1, Ch 4 Adapter (pp139–150)

- Adapters come from a problem that programmers face when trying to achieve re-usability, namely when interface of the existing classes are different from what the programmer needs.
- It is needed again and again when the problem comes again and again.
- There are many such problems when object-oriented programs are designed. Many of them are solved previously by creating classes that interact in various ways.
- These solutions are built on top of fundamental OO concepts like inheritance and encapsulation. New classes are made, using existing classes as data members or as base-classes.
- By examining how other programmers solve their problems, we can reuse their solutions when we meet similar problems, instead of going back to the fundamentals.

PMOOP(0396A)

PMOOP(0396A)-13.1

What's in a pattern

Each design pattern has the following elements to facilitate our study:

- **Pattern name:** for us to identify the pattern.
- **Problem:** describe when to apply the pattern. Includes where the problem occurs (**context**), what is the problem (**intent**), and what are the conditions for which the solution can be applied (**applicability**).
- **Solution:** describes how to solve the problem an abstract way. This enumerates the classes and objects needed, specifies their compile-time relationship (**structure**), and specifies how they interact when the program is executed (**collaboration**).
- **Consequences:** the result of the pattern and the cost, including the time and space requirement of the pattern, but more importantly the cost in managing the code after the pattern is applied. They allow us to easily evaluate a pattern.

PMOOP(0396A)-13.2

What we already know: Adapter

Adapter: Context and Intent

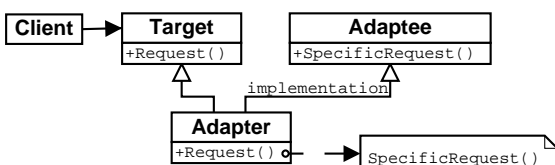
- A library class cannot be used directly just because the interface is different from what we want.
- We do not want to modify the existing class to suit our needs, since:
 - The source code of the class might be unavailable, or
 - The class is used in other projects. Modifying the class to suit the interface gives us two classes to maintain.
- The library class might also miss some functionalities that we need. We also need a place to implement these functionalities.

Solution: We create a class that *adapt* the library class to our interface.

PMOOP(0396A)-13.3

Structure: class adapter

The first type of adapter has the following structure:

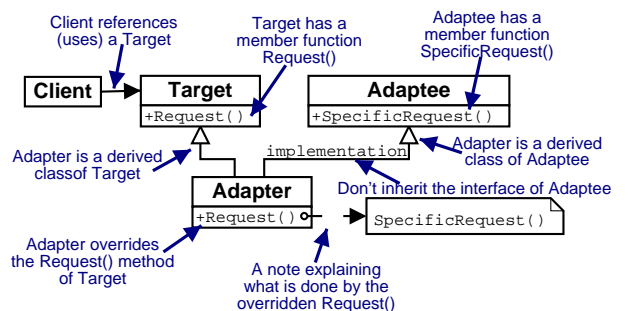


- **Target:** the class that the client needs (SoundOutput).
- **Client:** the objects or functions needing the target (main).
- **Adaptee:** the existing class that need adapting (OSS).
- **Adapter:** the adapted interface (OSSAdapter).

PMOOP(0396A)-13.4

How to read class structure diagrams?

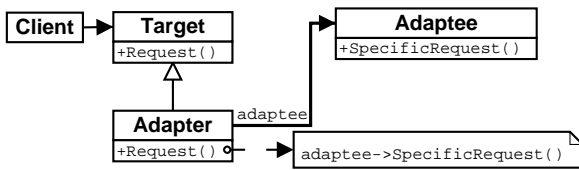
A class diagram shows the structure of the class in a graphical way. A lot of information can be found in a class diagram:



PMOOP(0396A)-13.5

The other type of adapters: object adapters

The second type of adapter has the following structure:



There is only one primary difference: now Adapter “references” an Adaptee, not derived from Adaptee.

The participants are exactly the same as class adapters.

PMOOP(0396A)-13.6

Consequences: the trade-off between the two

We have already noticed these:

Class adapter

- Commit to a concrete Adaptee class: subclasses cannot be adapted.
- Allow Adapter to override operations of Adaptee.
- Creates only one object, no pointer needed.

Object adapter

- One Adapter class can adapt all subclasses of Adaptee.
- More difficult to override operations in Adaptee.

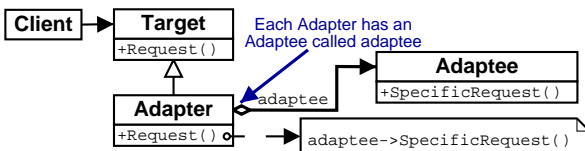
PMOOP(0396A)-13.7

Aggregation

Most of the class diagram constructs are illustrated in the class adapter. They include **class and methods, sub-classes and method overriding, references (i.e., acquaintance)** and **notes**.

Sometimes we want to emphasize that an object **A** is responsible for another object **B**, and **they have the same lifetime**. We usually say that **A contains B**. This is called an aggregation relationship.

We add a diamond shape symbol to emphasize these special references. For example, the following reflect our original design.



PMOOP(0396A)-13.8

Conclusion and Exercise

- Object-oriented concepts of encapsulation and inheritance allow reusability problems to be addressed.
- Instead of designing the solution everytime a reusability solution arises, we can understand the solutions of others and reuse them.
- Design patters are reusability problems together with their solutions and consequences.
- Solutions are presented with the assistance of class diagrams.
- Class diagrams show how classes are related by sub-class, reference and aggregate relations.

Exercise

- Read the command pattern (pp 233–242). (Needed in assignment 2.)

PMOOP(0396A)-13.9