MSO
Design Patterns: Adapter

Wouter Swierstra, Hans Philippi

September 12, 2018

Adapter pattern

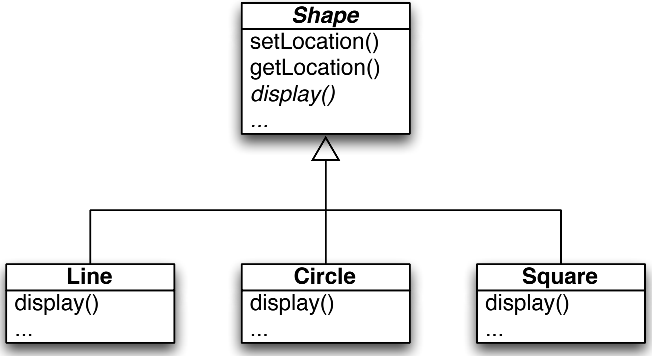
The Gang of Four state that the intent of the Adapter pattern is to:

> *Convert the interface of a class into another interface that the clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.*

Adapter: program to one uniform interface

Adapter example – shapes

Suppose you implement the following UML class diagram:



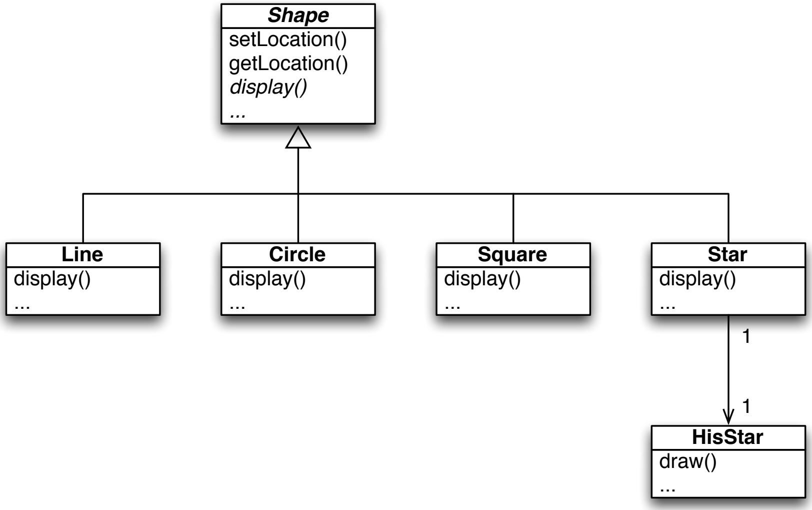
This encapsulates the exact implementation of individual types of shapes

Adapter example – adding stars

- Now suppose that you want to add a Star class ...
- And you have a friend that has already implemented a HerStar class
- But the HerStar class is not quite right:
  - the `display()` method is called `draw()`
  - it doesn't have a `location` attribute, but two attributes `pointX` and `pointY`
  - . . . and maybe there are a few other innocent differences
- I can't use the HerStar class in my system – it does not implement the necessary methods to be a subclass of Shape

Question: What can I do?

Adapter example



Implementation fragment

```
public class Star : Shape
{
    private HerStar star;

    public Star(...)
    {
        star = new HerStar(...);
    }

    public void Display()
    {
        star.draw();
    }

    ...
```

What this accomplishes?

- We can use both our original code and our friend's implementation
- We don't need to modify any existing code, but simply add a new class Star, that provides the desired interface to the HerStar class
- The rest of our code can continue to use *polymorphism* to hide the implementation details of individual shapes

Adapter pattern

- **Intent:** Match an existing object beyond your control to a particular interface
- **Problem:** A system has the right data and behaviour but the wrong interface
- **Solution:** The Adapter provides a wrapper with the desired interface
- **Consequences:** The Adapter pattern allows for preexisting objects to fit into new class structures without being limited by their interface
- **Implementation:** Contain the existing class in another class; have the containing class match the required interface and call the methods of the contained class

Taking a step back…

- Objects have *responsibilities* – as was clear from Larman's GRASP principles.
- Let's have a look at some of the more general design principles that you see in object-oriented design
- We have seen examples of *encapsulation*, where information is hidden – in particular, you may want keep certain implementation details abstract

Types of encapsulation

- Use private attributes to hide *data*
- Use private methods to hide *computation*
- Hiding one object behind another – nothing knows about the HerStar class except the Star class
- Encapsulating types – the implementation of the Line, Circle, Square, and Star classes is all hidden behind the abstract class Shape

What the adapter pattern achieves

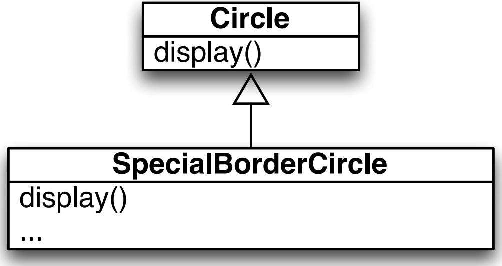
- Clients program using the abstract Shape class
- They are not aware of the issues related to the different kinds of shapes
- As a result, we can add new shapes *without having to worry about the impact on the client code*
- We have *encapsulated* the implementation of shapes using an abstract class

Beware of inheritance

- We use subclasses and inheritance to support encapsulation
- But handle inheritance with care ...
- ... remember: *Favour aggregation over inheritance?!*

Beware of inheritance

- Suppose I need to add a new type of circle with a special border
- I could just define a new subclass:



Favour aggregation over inheritance

Defining subclasses in this style has certain drawbacks:

- What if other Shapes also need different borders? It will be hard to reuse the SpecialBorderCircle classes methods.
- Suppose we add even more variation, like colour. Should we create coloured subclasses of all bordered Shapes? ...
- .. or the other way around?
- This lead to *weak cohesion* with respect to the code concerning shapes, borders and colors

Encapsulating variation

The Gang of Four write:

> *Consider what should be variable in your design. This approach is the opposite of focusing on the cause of redesign. Instead of considering what might force a change to a design, consider what you want to be able to change without redesign. The focus here is on encapsulating the concept that varies, a theme of many design patterns*

Example: encapsulating variation

Suppose I need to model different kinds of animals. My requirements might include:

- Each type of animal has a number of legs
- Different animals may have different kinds of movement (walking, flying, swimming, etc)
- Animal objects should be able to calculate how long it would take them to travel between two points over certain terrain

What is a good design?

A first approach

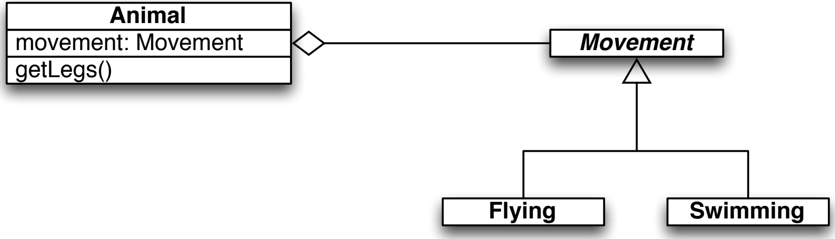
- Introduce attributes, getters and setters for the number of legs
- Animals also have different types of movement; we should add a Movement attribute to the animal class . . .

-
  ```
  const String WALKING = "Walking";
  const String FLYING = "Flying";
  ```
- But then calculating traveling time, the Animal class needs to switch on which movement an animal supports; adding new movement types means updating this function
- And then I want to add a new variation: carnivores vs herbivores?
- What about ducks? They can swim, walk or fly

A second approach

- So we should introduce an abstract class *Animal*
- And then add new subclasses for WalkingAnimal, FlyingAnimal, etc.
- But how should I add new variation: carnivores vs herbivores? Or cold-blooded vs warm-blooded?
- What about ducks? They can swim, walk or fly

A better alternative



Why is this better?

- We can introduce new kinds of movement, without having to adapt existing code
- We can introduce other kinds of properties of animals, such as diet, without distorting the subclass hierarchy
- Our code is more robust to changing requirements

What are we really doing?

- When coming up with a design, we focused on what different animals have in common (a number of legs) and what distinguishes different kinds of animals (their movement)
- This kind of analysis is sometimes called Commonality and Variability Analysis (CVA)
- CVA is a valuable tool for designers
- Implement an abstract class capturing the common interface
- Use aggregation for several non trivial properties
- Allow subclasses to implement the variation in those properties
- This is much more refined than the noun-verb analysis we have seen previously

CVA to design

When defining	**ask yourself**
An abstract class	What interface is needed to handle the responsibilities of this class?
Derived classes	How can I implement this particular flavour or variation of the superclass?

What do you think?

- Why might a CVA help think about ways to modify a system?
- Is it right to worry about variations at the beginning of your design?

Waterfall all over again . . . ??

If applying design patterns requires substantial thought to go into a design, how is this different from the waterfall model? Aren't we losing all the nice ideas about iterative software development? Agile/Scrum focus on writing code soon and *not* doing design up front.

Are we doing up front design?

STALLONE
CLIFFHANGER
"The best non-stop action picture of the year!"
—Vernon Scott, UPI
"An avalanche of thrills."
— Jami Bernard, NEW YORK POST

Material covered

Design Patterns Explained. Chapters 3–8.