
HUST
TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
ONE LOVE. ONE FUTURE.

1


SOICT
School of Information and Communication Technology
ONE LOVE. ONE FUTURE.

2

TRƯỜNG ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

IT3180 – Introduction to Software Engineering

15 – Reuse and Design Patterns

ONE LOVE. ONE FUTURE.

3

3

Software Reuse

It is good to design a program to reuse existing components. This can lead to better software at lower cost.

Potential benefits of reuse

- Reduced development time and cost
- Improved reliability of mature components
- Shared maintenance cost

Potential disadvantages of reuse

- Difficulty in finding appropriate components
- Components may be a poor fit for application
- Quality control and security may be unknown

4

4

Evaluating Software

- It is impossible to remove all bugs from software, even a well-established software
- Maintenance
  - **Is the software supported by an organization that will continue maintenance over the long term?**

5

Design For Change: Replacement of Components (1)

The software design should anticipate possible changes in the system over its life-cycle

New vendor or new technology

- Components are replaced because its supplier goes out of business
- Components from other source provide better functionality, support, pricing, etc.
- This can apply to either **open source** or **vendor-supplied** components

6

Design For Change: Replacement of Components (2)

The software design should anticipate possible changes in the system over its life-cycle

New Implementation

- The original implementation may be **problematic**
  - Poor performance
  - Inadequate backup and recovery
  - Unable to support growth and new features added to the system

Design For Change: Replacement of Components (3)

The software design should anticipate possible changes in the system over its life-cycle

Additions to the requirements

- When the system goes into production, it is usual to reveal both **weakness** and **opportunities** for **extra functionality** and **enhancement** to the user interface design
- For example, in data-driven application, it is almost certain that there will be requests for extra reports and ways of analyzing the data

Request for enhancements are often the sign of a successful system. Clients recognize latent possibilities

Design For Change: Replacement of Components (4)

The software design should anticipate possible changes in the system over its life-cycle

Changes in the application domain

- Most application domains change continually
  - Because of business opportunities
  - External changes (such as new laws)
  - New group of users
  - New technology
- It is rarely feasible to implement a completely new system when the application domain changes

➔Existing system must be modified

➔This may involve **extensive restructuring**, but it is important to **reuse existing code** as **much** as **possible**

9

9

Design Patterns

- **Design Patterns** are template designs that can be used in a variety of systems
- They are particularly appropriate in situations where classes are likely to be reused in a **system that evolves over time**

Sources:

- E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994
- Wikipedia has good discussion of many design patterns, using UML and other notation, with code samples.

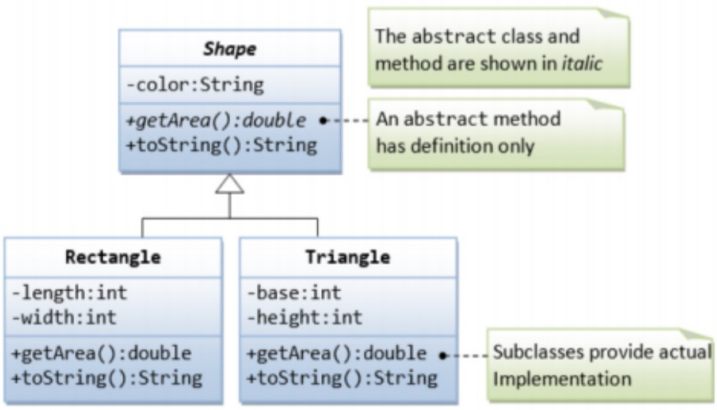
10

10

Inheritance and Abstract Class

- Design patterns make extensive use of **inheritance** and **abstract classes**
- Classes can be defined in terms of other classes through **inheritance**
  - Generalization classes – super classes
  - Specialization classes – subclasses
- Abstract classes
  - Super classes which contain abstract methods and are defined such that concrete subclasses extend them by implementing the abstract methods
  - May have not abstract methods, in this case, the intention is to prevent the creation of instances
  - Interface classes are abstract classes but for multi-inheritances and for specifying a standard protocols for all classes that realize them

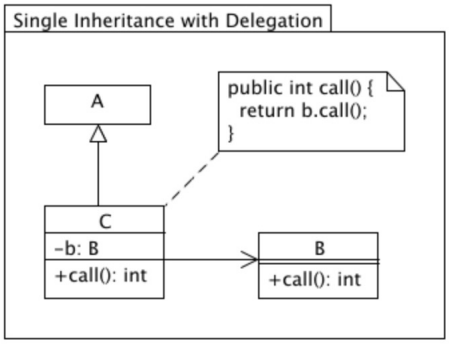
Inheritance and Abstract Class (2) - Example



Delegation

- A class is said to **delegate** to another class if it implements an operation by **resending a message** to another class.
- Delegation is an alternative to **inheritance** that can be used when reuse is anticipated.

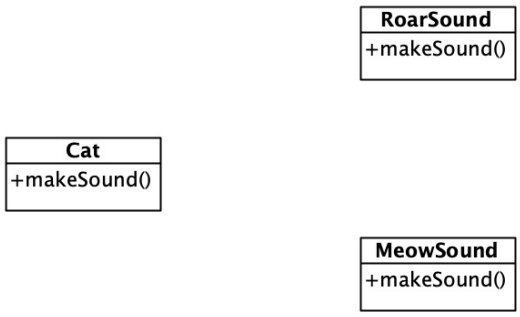
Delegation (2)



Delegation is like inheritance done manually through object composition

Delegation (3) - Example

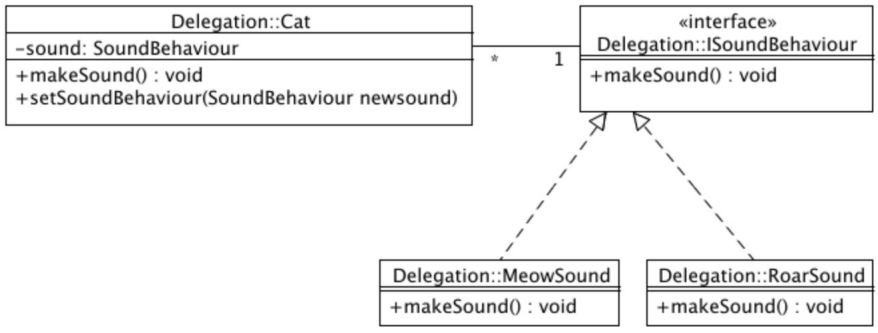
- *Case study: a cat's sound behavior - "meow" and "roar"*



- *How to compose the behavior of Cat at runtime?*
  Inheritance *or* ***Delegation****?*

Delegation (4) - Example

- Delegation makes it easy to compose behaviors at runtime



Delegation (5) – Example Source Code

```
public interface ISoundBehaviour {

        public void makeSound();
}

public class MeowSound implements ISoundBehaviour {

        public void makeSound() {
                System.out.println("Meow");
        }
}

public class RoarSound implements ISoundBehaviour {

        public void makeSound() {
                System.out.println("Roar!");
        }
}
```

Delegation (6) – Example Source Code

```
public class Cat {
  private ISoundBehaviour sound = new MeowSound();

  public void makeSound() {
    this.sound.makeSound();
  }

  public void setSoundBehaviour(ISoundBehaviour newsound) {
        this.sound = newsound;
  }
}
```

```
public class Main {
        public static void main(String args[]) {
                Cat c = new Cat();
                // Delegation
                c.makeSound();          // Output: Meow
                // now to change the sound it makes
                ISoundBehaviour newsound = new RoarSound();
                c.setSoundBehaviour(newsound);
                // Delegation
                c.makeSound();          // Output: Roar!
        }
}
```

Notation

ClassName	class name in italic indicates an abstract class
	dependency
	delegation
	inheritance

19

Strategy Pattern

Encapsulating Algorithms

20

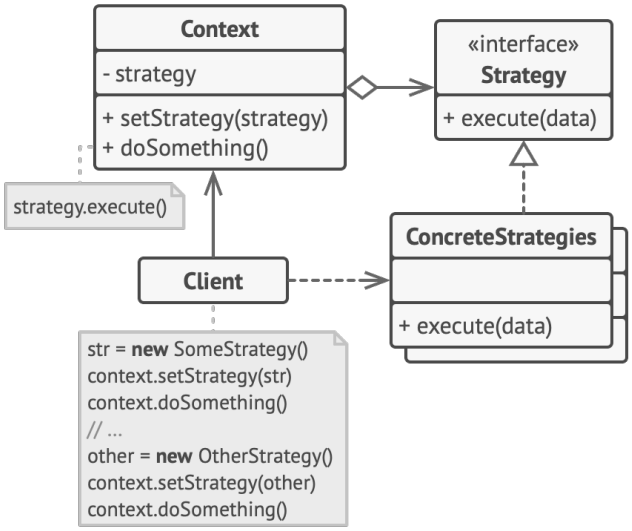
Problematic

- To solve a specific problem, there may be a **family of algorithms**
- Each algorithm is separated in a class called **strategy**
- The client will decide which strategy will be selected

21

21

Strategy Pattern - Structure



22

22

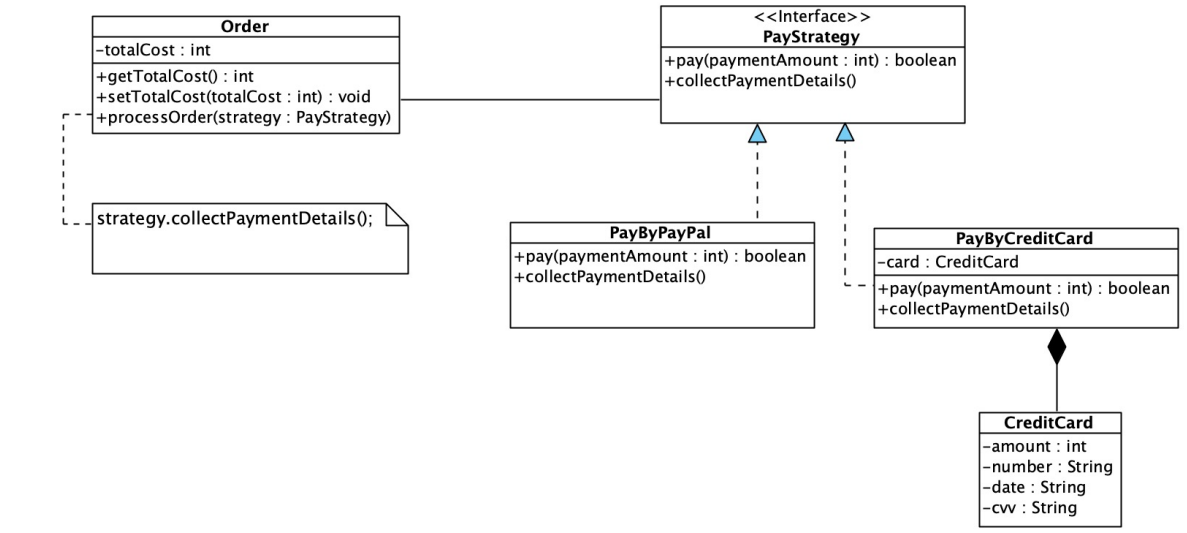
Strategy Pattern - Components

- The **Context** maintains a reference to one of the concrete strategies through the strategy interface
- The **Strategy** interface is common to all concrete strategies. It declares the methods which are used by the Context
- **Concrete Strategies** implement different variations of an algorithm the Context uses
- The **Client** creates a specific strategy object and passes it to the Context

Case Study in many software projects

- Payment method in an e-commerce application
- There are various payment methods in an e-commerce application. After selecting a product to purchase, a customer picks a payment method: either Paypal or Credit Card.
- MoMo and ZaloPay are considered in the future

Case Study in many software projects



25

25

26

Problematic

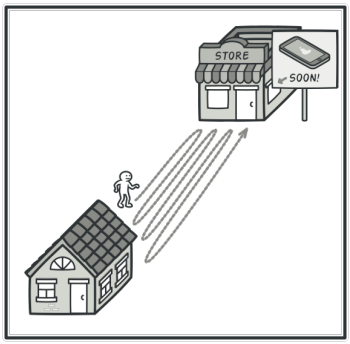
- To define a **subscription mechanism** to **notify** multiple objects about any **events** that happen to the object they are observing
- **Event-Subscriber**
- **Listener**

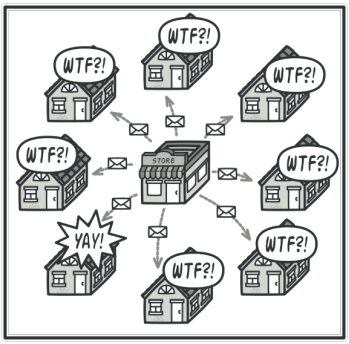
27

27

Case Study in many software projects

- The customer want to be **notified** about a new coming **product**
- **Notify** a set of customers about new **events**, new **vouchers**
- An admin want to associate a discount/coupon with multiple product. Any changes in discount/coupon must be **notified** to its associated products



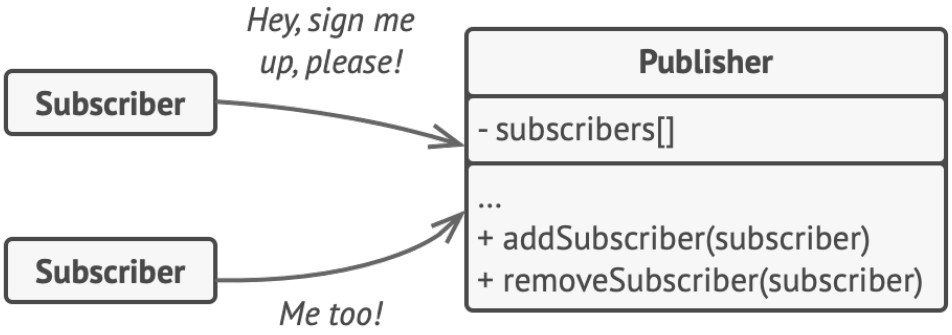


28

28

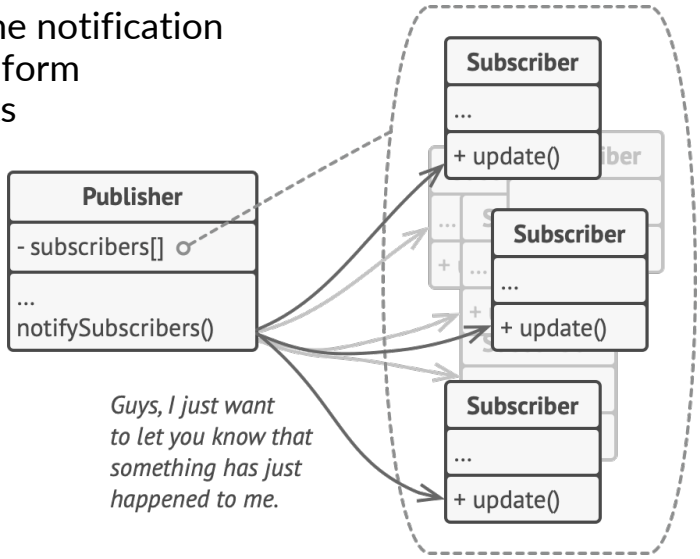
Observer Pattern Structure

- Add a subscription mechanism to the publisher so individual object can subscribe to or unsubscribe from a stream of events coming
- `subscribers`: a list of subscribers of the publisher

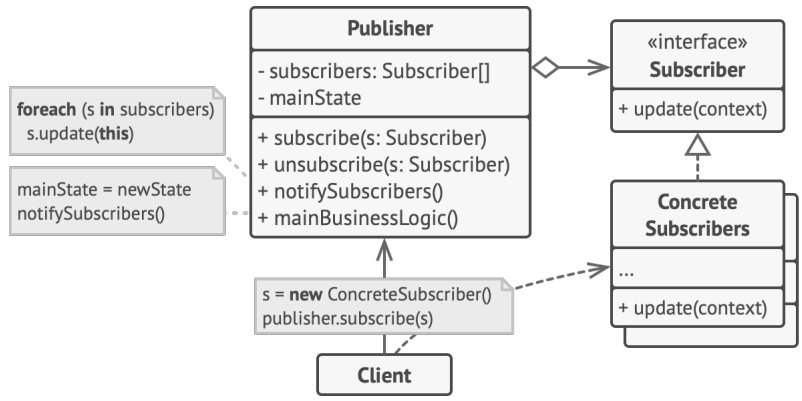


Observer Pattern Structure (2)

- `notifySubscribers()`: the notification mechanism of publisher to inform subscribers about new events



Observer Pattern Structure (3)



31

31

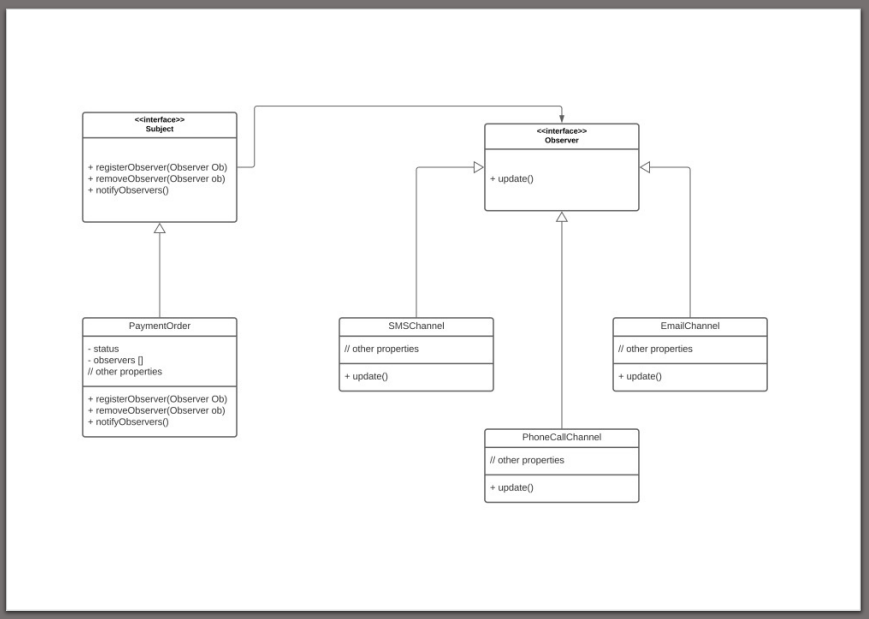
Case study: ecommerce

- Whenever an user make a new purchase, he or she will receive a notification about the order.
- Notification mechanisms: email, SMS, PhoneCall

- What plays the role of **Publisher**?
- What are **Observers**?

32

32



Solution

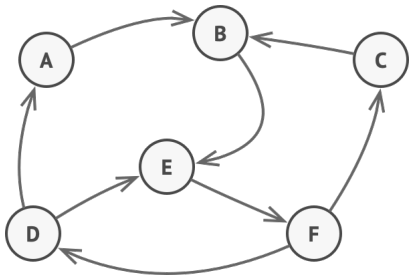
33

33

34

Problematic

- **State is a behavior pattern**
- Allow an object to alter its behavior when its internal state changes
- Closely related to the concept of **Finite State Machine**

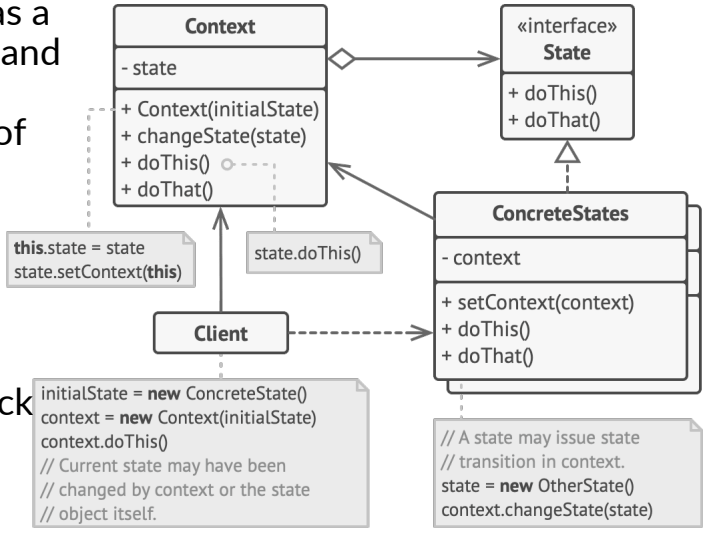


Case study in many projects

- When a new order is created, the users should view the state of the order
- When the state of an order is changed, some actions will be fired!

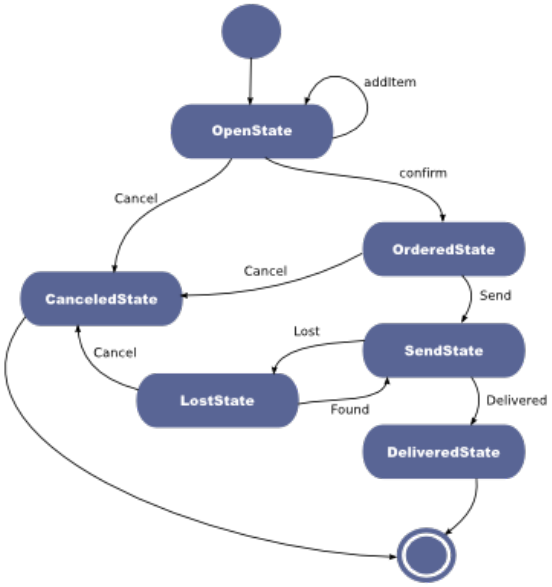
State Structure

- Context: the object which has a reference to one of its **state** and a mechanism to prceess whenever there is a change of its state (**changeState**)
- abstract State: state specific methods
- concrete States: **specific** implementation for **state methods**, has a reference back to the context

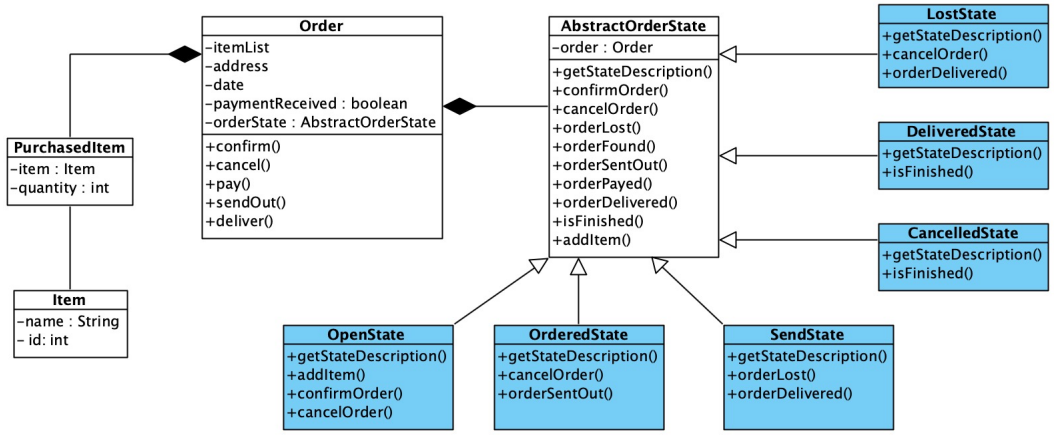


Case study

- Order states follow a finite-state machine diagram
- Which is the Context object?
- What are states objects?



Solution



39

39

15 – Reuse and Design Pattern

(end of lecture)

ONE LOVE. ONE FUTURE.

40

40