WASHINGTON STATE UNIVERSITY
EXTENSION

Cpts321 - Design Patterns

Guangbei Yi

Who Am I?

- Major in B.S. Software Engineering
  - Minor: Music
- Joined WSU in 2020
- B.S. Computational Mathematics
  - (Graduated 2017 – in China)
- UG Research Student at WSU, supervised by Dr. Haipeng Cai
- Other: I have 3 year working experience as a QA Engineer at Ubisoft

What are Design patterns?

- From book: Design Patterns - Elements of Reusable Object-Oriented Software
  - Author: Gang of Four(GOF)
  - First mentions the concept of design patterns in software development.
- Represents best practice.
- Includes solutions to common problems.
- These solutions are the result of trial and error by numerous software developers over a considerable period.
- Usually used by experienced object-oriented software developers.

WSU

Types of Design Patterns

There are 23 design patterns in total, can be divided into three main categories:

- Creational patterns:
  - Factory Pattern
  - Abstract Factory Pattern
  - Singleton Pattern
  - Builder Pattern
  - Prototype Pattern
- Structural patterns:
  - Adapter Pattern
  - Bridge Pattern
  - Filter/Criteria Pattern
  - Composite Pattern
  - Decorator Pattern
  - Facade Pattern
  - Flyweight Pattern
  - Proxy Pattern

Behavior patterns:
Chain of Responsibility Pattern
Command Pattern
Interpreter Pattern
Iterator Pattern
Mediator Pattern
Memento Pattern
Observer Pattern
State Pattern
Null Object Pattern
Strategy Pattern
Template Pattern
Visitor Pattern

vsu

However, there are other import patterns that are not in the book – for example J2EE Patterns

- MVC Pattern
- Business Delegate Pattern
- Composite Entity Pattern
- Data Access Object Pattern
- Front Controller Pattern
- Intercepting Filter Pattern
- Service Locator Pattern
- Transfer Object Pattern

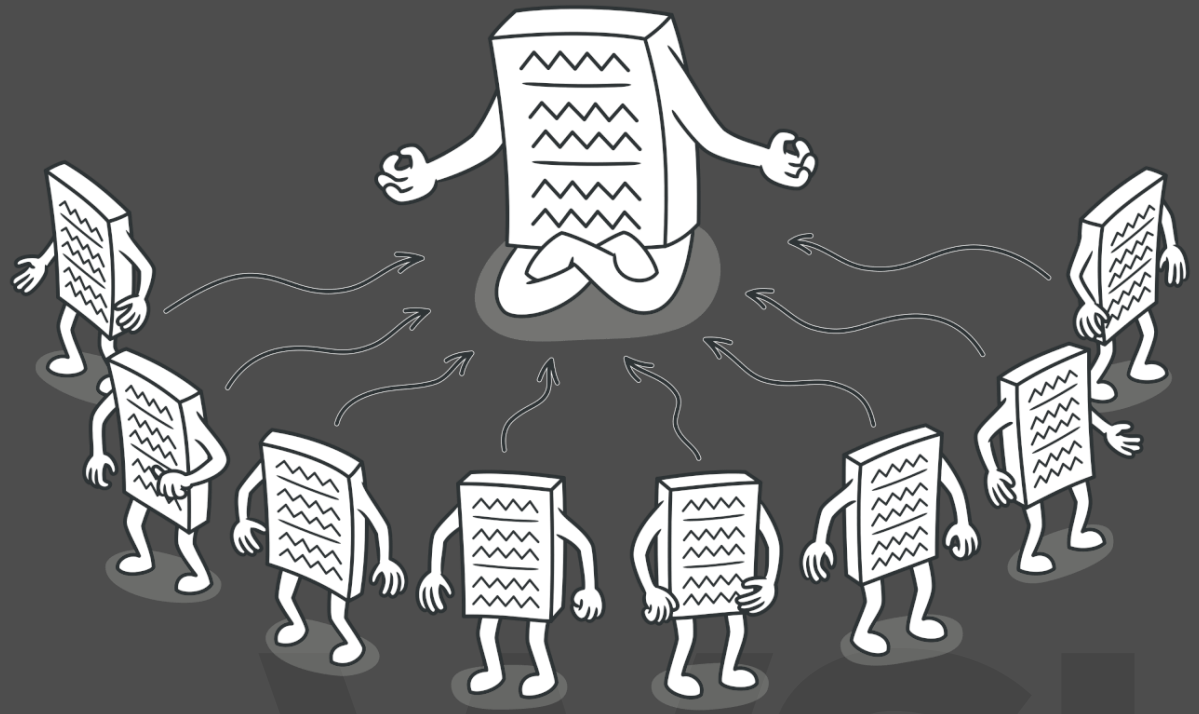
WSU

Design patterns examples

Singleton Pattern

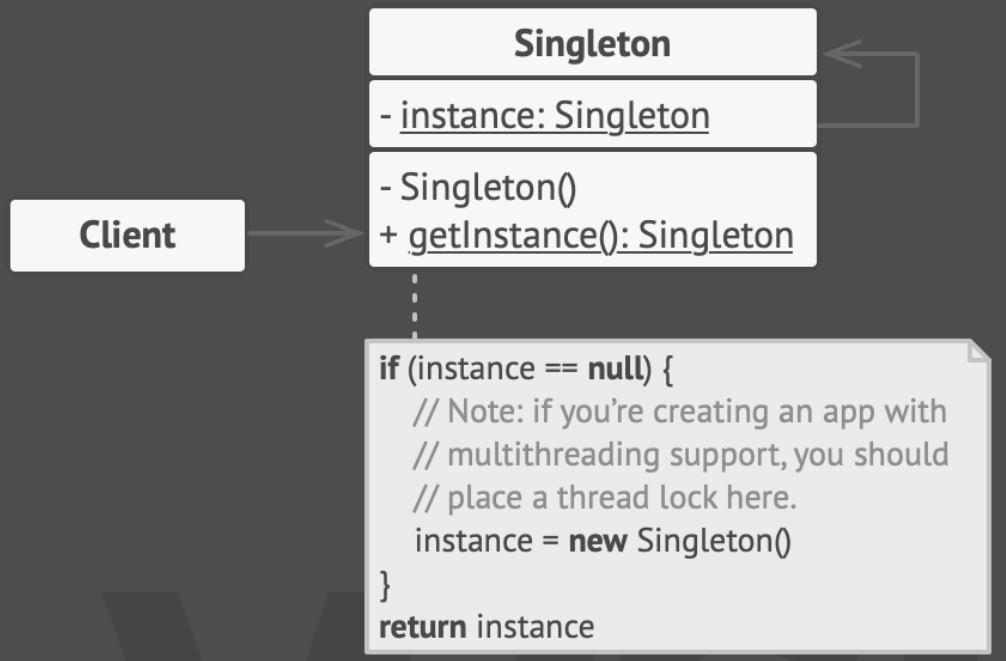
Introduction

- Defines an "Instance" or "GetInstance" operation that lets clients access the unique instance of the class. Instance is a class operation (that is, a class method in Smalltalk and a static member function in C#).
- May be (and usually is) responsible for creating its own unique instance.



Use the Singleton pattern when

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code
- the constructor is not anymore visible for clients



Real World Example

- The government is an excellent example of the Singleton pattern. A country can have only one official government. Regardless of the personal identities of the individuals who form governments, the title, “The Government of X”, is a global point of access that identifies the group of people in charge.
- Database is also a good example as well.

Demo Code

```
1   public class SingleObject {
2
3      //will not be able to create an object of this class
4      private SingleObject(){}
5
6      private static SingleObject _instance;
7      public static SingleObject GetInstance()
8      {
9         if (_instance == null)
10        {
11           _instance = new SingleObject();
12        }
13        return _instance;
14     }
15
16     public void showMessage(){
17        //print("Hello World!");
18     }
19  }
```

```
21  public class SingletonPatternDemo {
22     public static void main(String[] args) {
23        // it doesn't work
24        //SingleObject object = new SingleObject();
25        SingleObject object = SingleObject.GetInstance();
26        object.showMessage();
27     }
28  }
```

WSU

Pros and Cons

- You can be sure that a class has only a single instance.
- You gain a global access point to that instance.
- The singleton object is initialized only when it's requested for the first time.

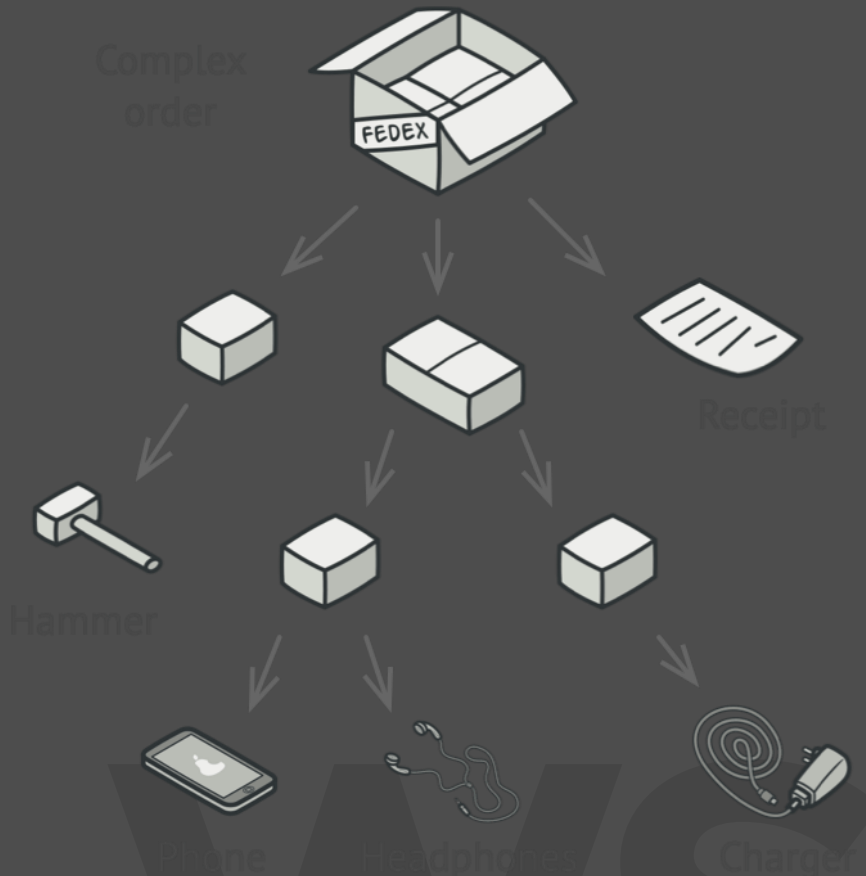
- The pattern requires special treatment in a multithreaded environment so that multiple threads won't create a singleton object several times.
- It may be difficult to unit test the client code of the Singleton because many test frameworks rely on inheritance when producing mock objects. Since the constructor of the singleton class is private and overriding static methods is impossible in most languages, you will need to think of a creative way to mock the singleton. Or just don't write the tests. Or don't use the Singleton pattern.

WSU

Composite Pattern

Introduction

- The composite pattern is a partitioning design pattern. The composite pattern describes a group of objects that are treated the same way as a single instance of the same type of object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.



Use the Composite pattern when

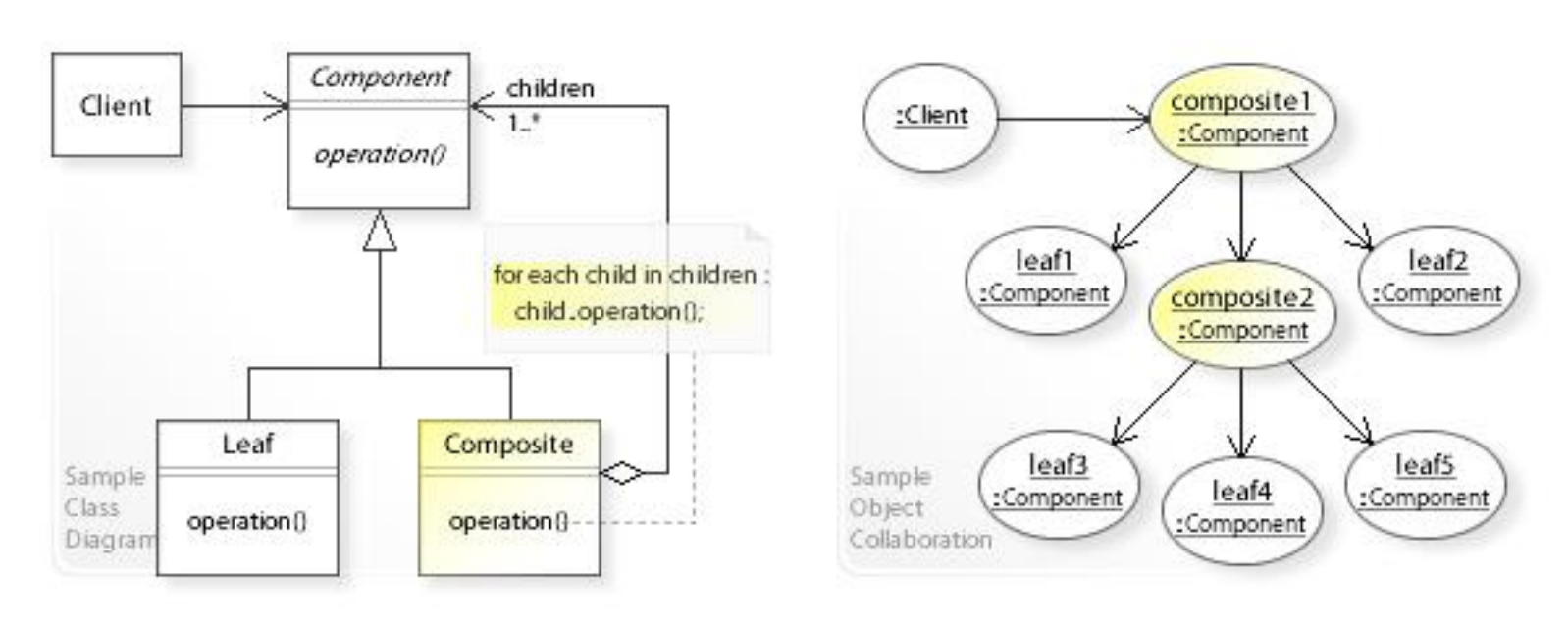
- You want to represent part-whole hierarchies of objects.
- You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

WSU

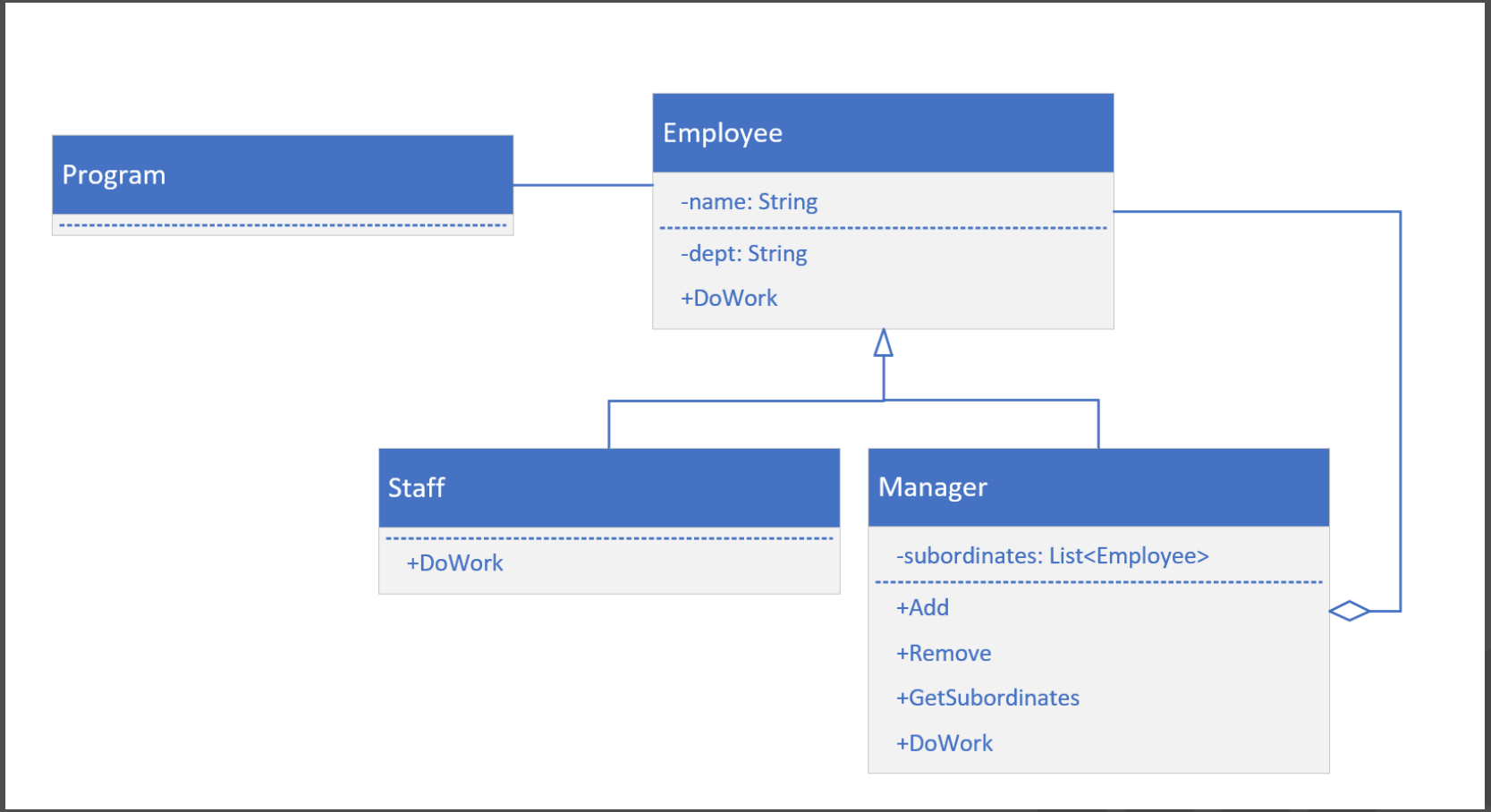
Real World Example

- Armies of most countries are structured as hierarchies. An army consists of several divisions; a division is a set of brigades, and a brigade consists of platoons, which can be broken down into squads. Finally, a squad is a small group of real soldiers. Orders are given at the top of the hierarchy and passed down onto each level until every soldier knows what needs to be done.
- A company structure is also a good example.

UML Diagram


Client
Component
operation()
children
1..*
for each child in children :
child.operation();
Leaf
operation()
Composite
operation()
Sample
Class
Diagram
:Client
composite1
:Component
leaf1
:Component
composite2
:Component
leaf2
:Component
leaf3
:Component
leaf4
:Component
leaf5
:Component
Sample
Object
Collaboration

Example UML


Program
Employee
-name: String
-dept: String
+DoWork
Staff
+DoWork
Manager
-subordinates: List<Employee>
+Add
+Remove
+GetSubordinates
+DoWork

Example

```
public abstract class Employee {
    private String name;
    private String dept;

    public Employee(String name,String dept) {
        this.name = name;
        this.dept = dept;
    }
public class Manager : Employee
{
    private List<Employee> subordinates;
    public Manager(String name,String dept) {
        super(name,dept);
    }
    public void add(Employee e){
        subordinates.add(e);
    }
    public void remove(Employee e) {
        subordinates.remove(e);
    }
    public List<Employee> GetSubordinates(){
        return subordinates;
    }
}
```

```
public class Staff : Employee
{
    public Staff(String name,String dept) {
        super(name,dept);
    }
    public void DoWork(){
        //do work
    }
}

public static void main(String[] args)
{
    Employee CEO = new Employee("John","CEO");
    Employee clerk1 = new Staff("Laura","Marketing");
    Employee salesExecutive1 = new Staff("Richard","Sales");
    CEO.add(salesExecutive1);
    CEO.add(clerk1);
    foreach(var employee in CEO.subordinates){
        foreach(var employee in CEO.subordinates){
            //do work
        }
    }
}
```

Pros and Cons

- You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
- You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

- It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

WSU

Strategy Pattern

Introductions

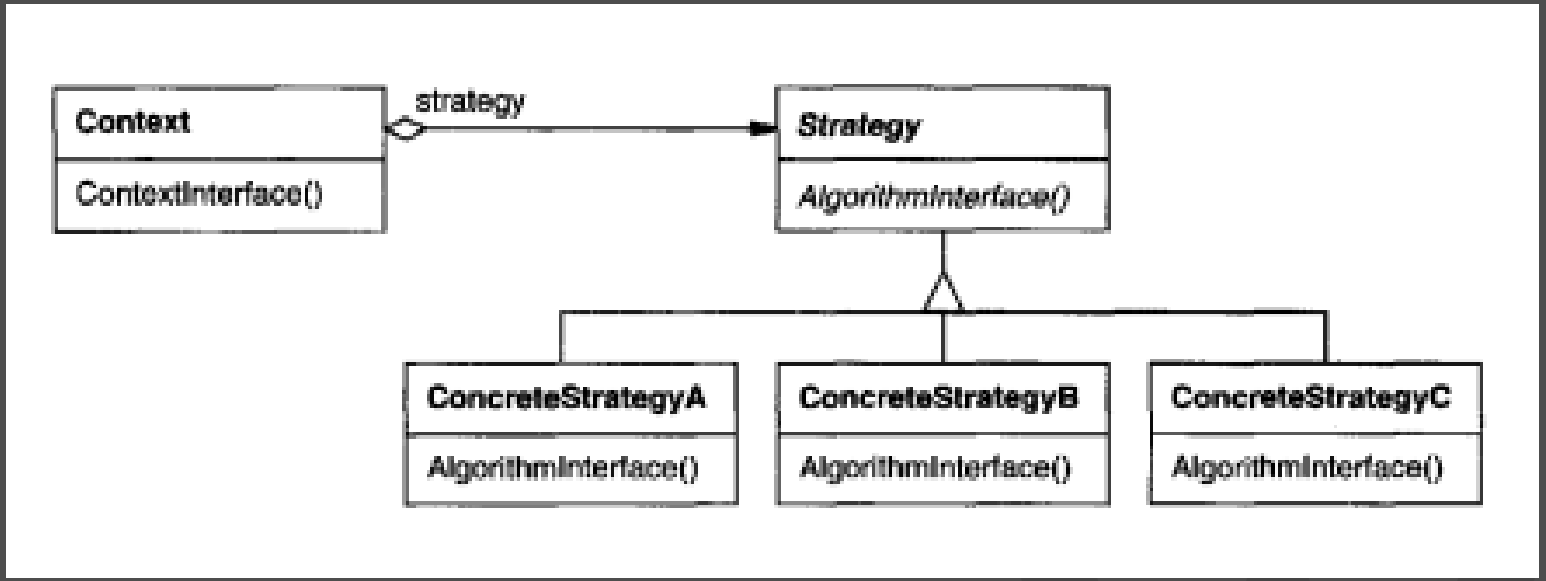
- Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

WSU

Use the Strategy pattern when

- many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- you need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms
- an algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- • a class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Example UML


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

Read world Example

- Imagine that you have to get to the airport. You can catch a bus, order a cab, or get on your bicycle. These are your transportation strategies. You can pick one of the strategies depending on factors such as budget or time constraints.

Demo Code

```
public interface Strategy {
    public int doOperation(int num1, int num2);
}

public class OperationAdd : Strategy{
    public override int doOperation(int num1, int num2) {
        return num1 + num2;
    }
}

public class Context {
    private Strategy strategy;

    public Context(Strategy strategy){
        this.strategy = strategy;
    }
    public int executeStrategy(int num1, int num2){
        return strategy.doOperation(num1, num2);
    }
}
```

```
22    public static void main(String[] args) {
23        Context context = new Context(new OperationAdd());
24        System.out.println("10 + 5 = " + context.executeStrategy(10, 5));
25    }
```

WSU

Pros and Cons

- You can swap algorithms used inside an object at runtime.
- You can isolate the implementation details of an algorithm from the code that uses it.
- You can replace inheritance with composition.
- Open/Closed Principle. You can introduce new strategies without having to change the context.

- If you only have a couple of algorithms and they rarely change, there's no real reason to overcomplicate the program with new classes and interfaces that come along with the pattern.
- Clients must be aware of the differences between strategies to be able to select a proper one.
- A lot of modern programming languages have functional type support that lets you implement different versions of an algorithm inside a set of anonymous functions. Then you could use these functions exactly as you'd have used the strategy objects, but without bloating your code with extra classes and interfaces.

WSU

Practice Time!

- Walmart wants to develop software to track the warehouse and the work of each employee. Each employee has their own tasks:
- Everyone except the CEO has a reporting manager. Buyers are responsible for replenishing the warehouse, and cashiers are responsible for selling promotional product mixes.
- There are many possible combinations of replenishment and sales

Hits!

- Using Singleton for warehouse.
- Using Composite pattern for Company organization.
- Using Strategy pattern for combinations.

WSU

```
public class Warehouse
{
    private int count = 10;
    private Warehouse()
    {}

    private static Warehouse instance;
    public static Warehouse GetInstance()
    {
        return this.instance;
    }

    public void AddProduct()
    {
        this.count++;
    }
    public void SellProduct(int number)
    {
        this.count -= number;
    }
}
```

```
public abstract class Employee {
    private String name;
    private String dept;
    public Employee(String name,String dept) {
        this.name = name;
        this.dept = dept;
    }
}

public class Manager : Employee
{
    private List<Employee> subordinates;
    public Manager(String name,String dept) {
        super(name,dept);
    }
    public void Add(Employee e){
        this.subordinates.Add(e);
    }

    public List<Employee> GetSubordinates(){
        return this.subordinates;
    }
}

public class Staff : Employee
{
    public Staff(String name,String dept) {
        super(name,dept);
    }
    public void DoWork(Combination cmb){
        cmb.executeStrategy();
    }
}
```

```
51  public interface Combination
52  {
53      public int doOperation();
54  }
55
56  public interface AddOneCombination : Combination
57  {
58      public override int doOperation()
59      {
60          Warehouse wh = Warehouse.GetInstance();
61          warehouse.AddProduct(1);
62      }
63  }
64
65  public interface SellOneCombination : Combination
66  {
67      public override int doOperation()
68      {
69          Warehouse wh = Warehouse.GetInstance();
70          warehouse.SellProduct(1);
71      }
72  }
```

```
74  public class Context {
75      private Combination Combination;
76      public Context(Combination Combination)
77      {
78          this.Combination = Combination;
79      }
80      public int executeStrategy()
81      {
82          return strategy.doOperation();
83      }
84  }
85
```

```
82  public class Program
83  {
84      public static int Main()
85      {
86          Manager CEO = new Manager("John","CEO");
87          Staff buyers = new Staff("Michel","Head Marketing");
88          Staff cashiers = new Staff("Richard","Sales");
89
90          CEO.add(buyers);
91          CEO.add(cashiers);
92
93          Warehouse wh = Warehouse.GetInstance();
94
95          Combination AddCom = new Context(new AddOneCombination());
96          Combination SellCom = new Context(new SellOneCombination());
97
98          buyers.DoWork(wAddCom);
99          cashiers.DoWork(SellCom);
100     }
101 }
```

Any Questions?