Design Patterns 2

Week 12

Design patterns are communication tools. If nobody else on your team understands them, then they don't help you

– Martin Fowler

Objectives

- Understand the structural patterns
- Understand the behavioral patterns

Contents

- Structural patterns
  - Adapter
  - Facade
  - Decorator
  - Proxy
  - Composite
- Behavioral patterns
  - Visitor
  - Iterator
  - Observer
  - Strategy

Recap: GoF Classification of Design Patterns

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	**Factory Method**	Adapter (class)	Interpreter Template Method
	Object	**Abstract Factory** **Builder** **Prototype** **Singleton**	**Adapter (object)** Bridge **Composite** **Decorator** **Facade** Flyweight **Proxy**	Chain of Responsibility Command **Iterator** Mediator Memento **Observer** State **Strategy** **Visitor**

Structural Patterns & Behavioral Patterns

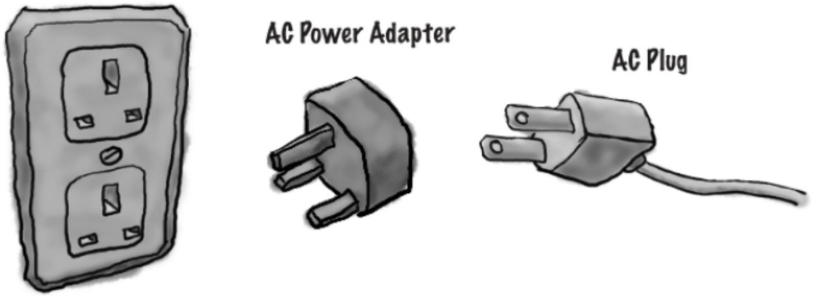
- Structural patterns are concerned with how classes and objects are **composed to form larger structures**
- Behavioral patterns are concerted with algorithms and **the assignment of responsibilities** between objects
- Behavioral patterns describe not just the patterns of objects or classes but also **the patterns of communication between them**

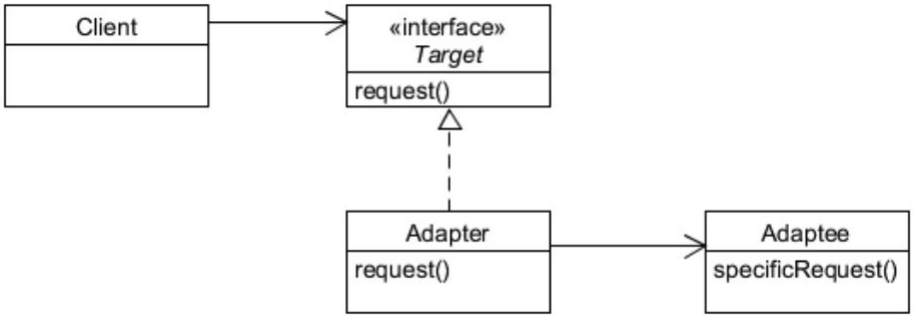
Structural Patterns

- Structural patterns let you compose classes or objects into larger structures
- Examples of structural patterns
  - Adapter
  - Facade
  - Decorator
  - Proxy
  - Composite

Adapter

- Adapter Pattern converts the interface of a class into another interface the clients expect
- Use when
  - a class to be used doesn't meet interface requirements

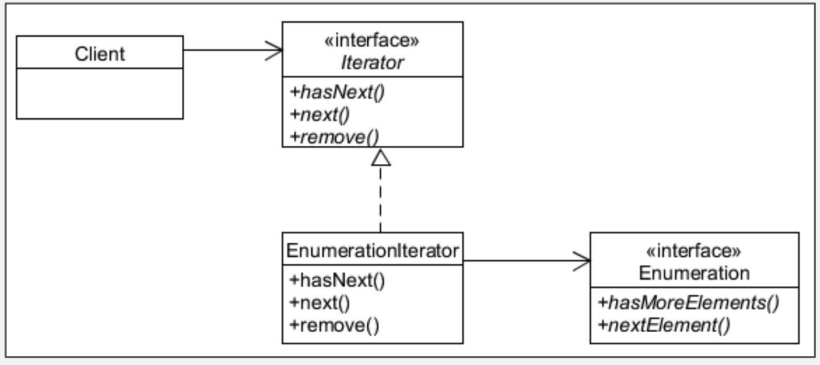




Code with Adapter

```
public class EnumerationIterator implements Iterator {
    Enumeration enum;

    public EnumerationIterator(Enumeration enum) {
        this.enum = enum;
    }
    public boolean hasNext() {
        return enum.hasMoreElements();
    }
    public Object next() {
        return enum.nextElement();
    }
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
```

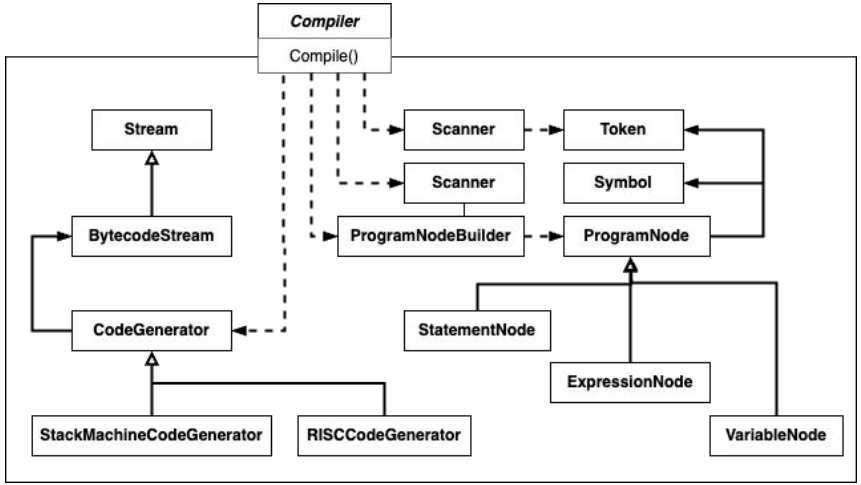
We cannot modify the *Enumeration* class



Facade

- Facade Pattern provides a **unified interface** to a set of interfaces in a subsystem
- It defines a **higher-level interface** that makes the subsystem easier to use
- Use when
  - a simple interface is needed to provide access to a complex system
  - the adaptor pattern not only converts an existing API but also simplifies it

Example of Facade



We just call *Compile()* method to use *Compiler*

Language: C++

```
class Scanner {//….};
class Parser {//….};
...
class CodeGenerator {//….};

class Compiler { //facade class
    public:
    Compiler();
    virtual void Compile(istream&, BytecodeStream&);
};

void Compiler::Compile(istream& input, BytecodeStream& output) {
    Scanner scanner(input);
    ProgramNodeBuilder builder;
    Parser parser;
    parser.Parse(scanner, builder);
    RISCCodeGenerator generator(output);
    ProgramNode* parseTree = builder.GetRootNode();
    parseTree->Traverse(generator);
}
```

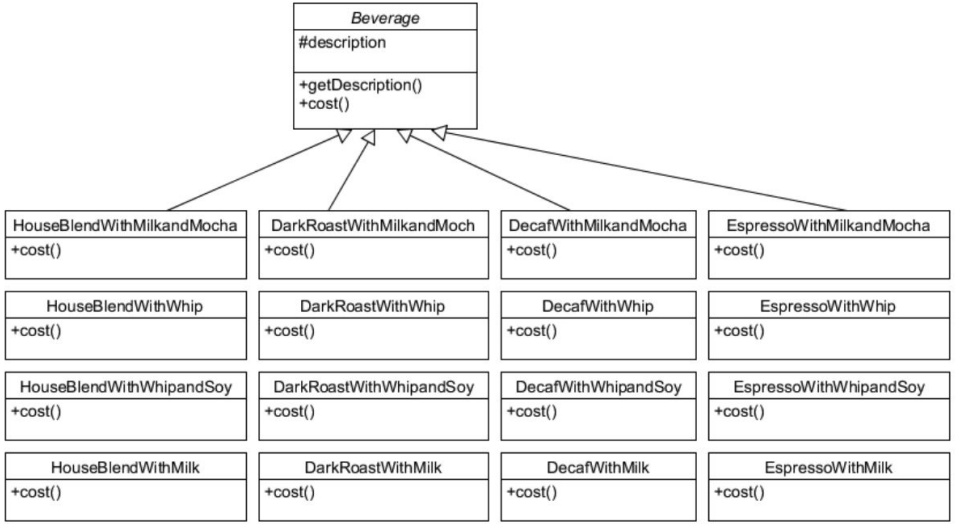
Decorator

- Decorator Pattern **dynamically** attaches additional responsibilities to an object
- It provides a flexible alternative to subclassing for extending functionality
- Use when
  - subclassing to achieve modification is impractical or impossible
  - object responsibilities and behaviors should be dynamically modifiable

Why Decorator?

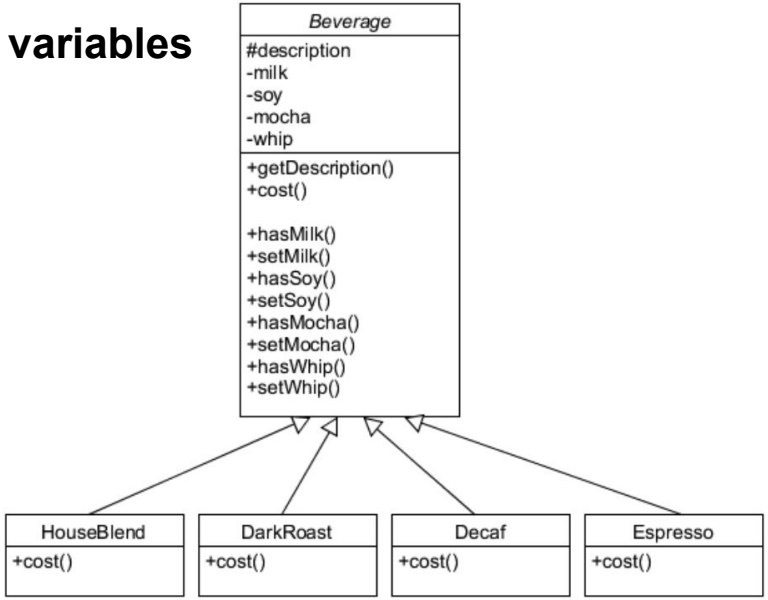
- We want to make beverage with many condiments

Option 1:
Subclass



Too many subclasses

Option 2:
Adding boolean variables



Too many modifications

Code without Decorator (Option 2)

```
public class Beverage{
    boolean milk, soy, mocha, whip;
    public float cost(){
        float condimentCost = 0.0;
        if (hasMilk())
            condimentCost += milkCost;
        if (hasSoy())
            condimentCost += soyCost;
        if (hasMocha())
            condimentCost += mochaCost;
        if (hasWhip())
            condimentCost += whipCost;
        return condimentCost;
    }
}
```

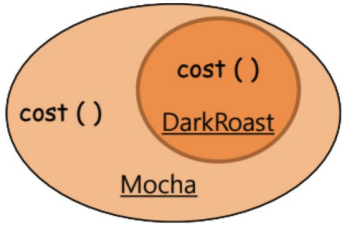
- Price changes for condiments require changes in the existing code
- New condiments require new attributes and changes in the cost method
- For some of these beverages the condiments may not be appropriate
- What if a customer wants double mocha?

Solution with Decorator

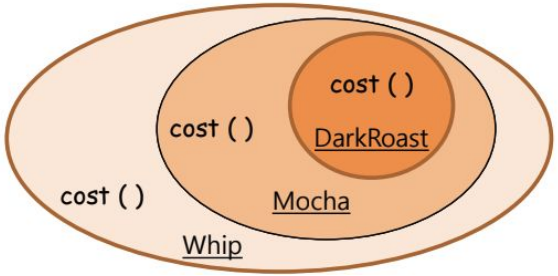
- Concept: Wrap an instance with another instance Iteratively

1. Single instance

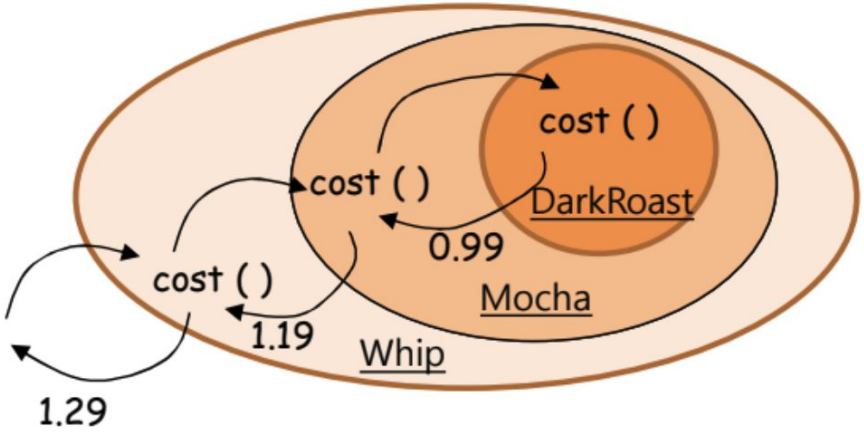
2. Add Mocha



3. Add Whip

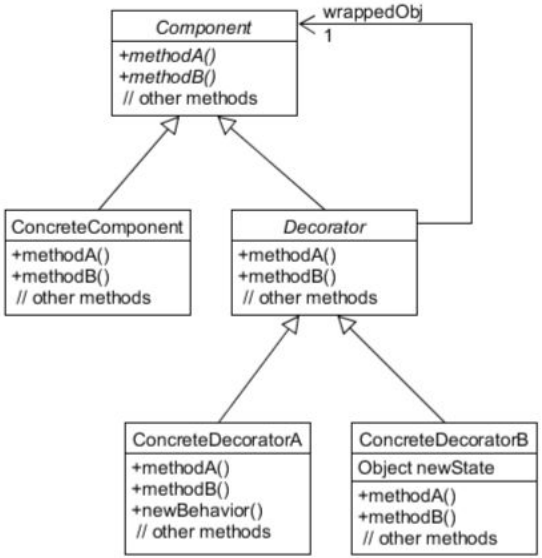


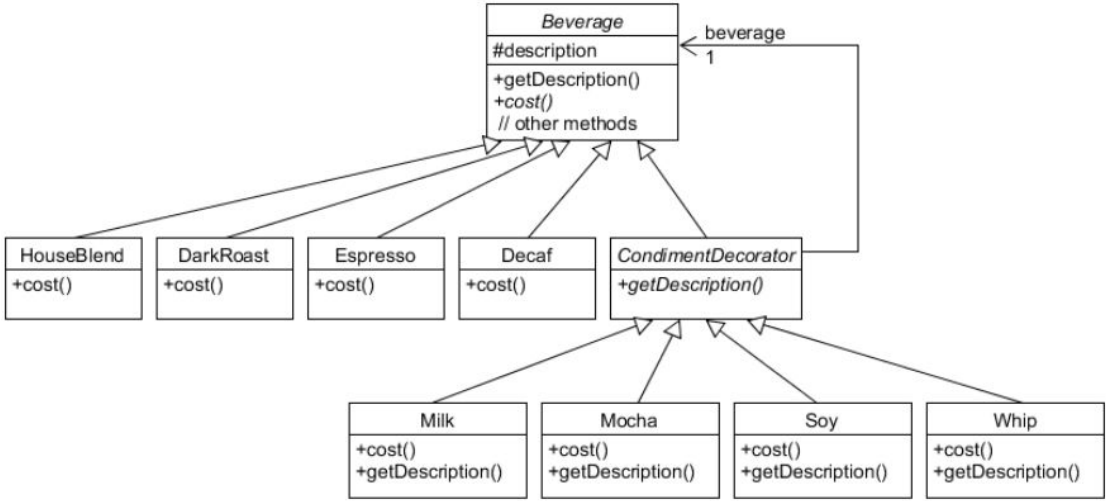
Call *cost()*



Solution with Decorator

- Decorator inherits the Component class (e.g., Beverage)
- Concrete Decorator (e.g., Milk) inherits the Decorator class
- Decorator recursively composites the parent class





Code with Decorator (1/2)

```
public abstract class Beverage {
    ...
    public abstract double cost();
}
```

```
public abstract class CondimentDecorator extends Beverage
{
    protected Beverage beverage;
    public abstract String getDescription();
}
```

```
public class Decaf extends Beverage {
    public double cost() {
        return 1.99;
    }
}
//HouseBland, Darkroast, Expresso
class are similarly defined
```

```
public class Mocha extends CondimentDecorator {
    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }
    public double cost() {
        return .20 + beverage.cost();
    }
}
```

Delegate parent class

Code with Decorator (2/2)

```
public class Whip extends CondimentDecorator {
    public Whip(Beverage beverage) {
        this.beverage = beverage;
    }
    public double cost() {
        return .05 + beverage.cost();
    }
}

//Soy and Milk decorators are similarly defined
```

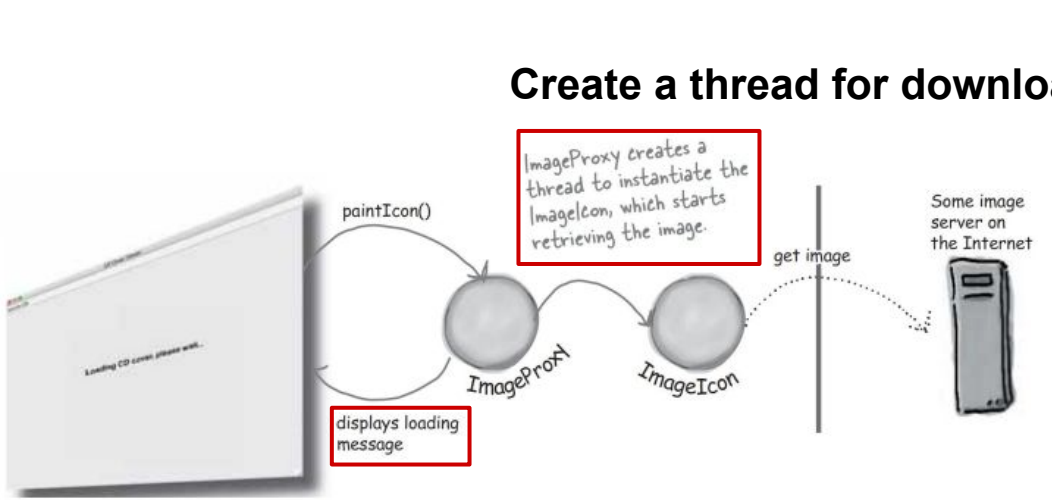
```
public class StarbuzzCoffee {
    public static void main(String args[]) {
        Beverage beverage = new Espresso();

        Beverage beverage2 = new Decaf();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);

        System.out.println(“ $” +
                                beverage2.cost());
    }
}
```

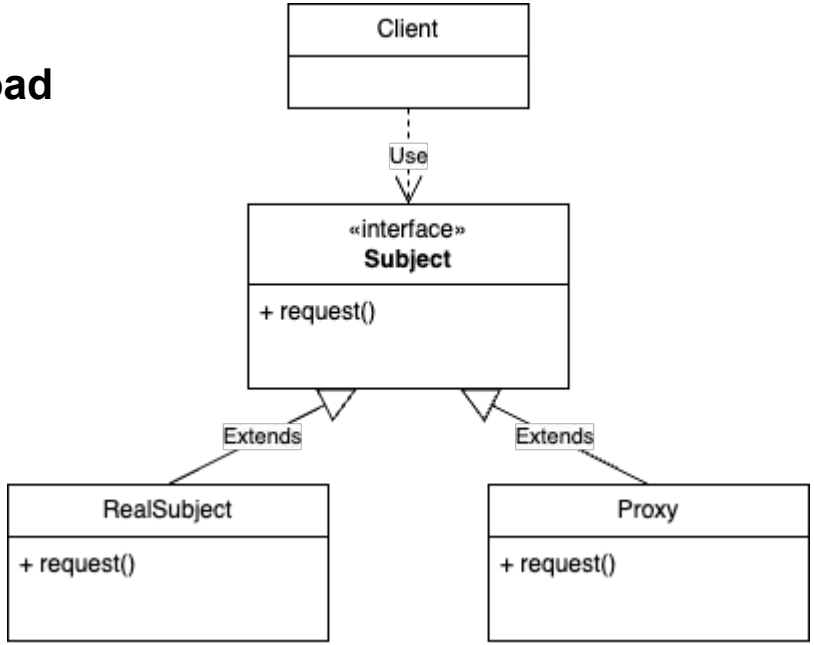
Proxy

- Proxy Pattern provides a surrogate to access-control another object
- Proxy Pattern is often used to access **remote** or **expensive** object to create or in need of securing
- Use when
  - access control for the original object is required
  - access to the original object is costly
  - added functionality is required when an object is accessed

Example of Proxy

- We want to get a very large image from a server





Code with Proxy #1 (1/2)

```
public interface Image { // Subject
   void displayImage();
}
public class RealImage implements Image { // RealSubject
    private String fileName;
    public RealImage(String fileName) {
        this.fileName = fileName;
        // Load the image with fileName (costly op.)
    }
    @Override
    public void displayImage() {
        System.out.println("Displaying " + fileName);
    }
}
```

```
public class ProxyImage implements Image {  // Proxy
    private RealImage realImage;
    private String fileName;
    public ProxyImage(String fileName) {
        this.fileName = fileName;
    }
    @Override
    public void displayImage() { // Lazy loading
        if (realImage == null) {
            // Create a thread to instantiate RealImage
            // Display loading message
        }
        realImage.displayImage();
    }
}
```

Code with Proxy #1 (2/2)

- When the object is requested first, `image.displayImage()` displays loading message and loads image

```
public class Client{
    public static void main(String[] args){
        Image image = new ProxyImage("cat.jpg");
        // The above code can be replaced by Creational Patterns

        image.displayImage();
    }
}
```

Code with Proxy #2 (1/2)

```java
public interface GitUser {
    void push();
}

public class RealGitUser implements GitUser{
    @Override
    public void push() {
        System.out.println("Push the code");
    }
}
```

```java
public class ProxyGitUser implements GitUser{
    private GitUser realGitUser;
    private int permit;
    public ProxyGitUser(int permit){
        this.realGitUser = new RealGitUser();
        this.permit = permit;
    }
    ...
```

Code with Proxy #2 (1/2)

```
  ... // Previous Slide
    @Override
    public void push() { // Protection
        if (this.permit != 1){
            System.out.println("Don't have permission");
        }
        else{
            this.realGitUser.push();
        }
    }
}
```

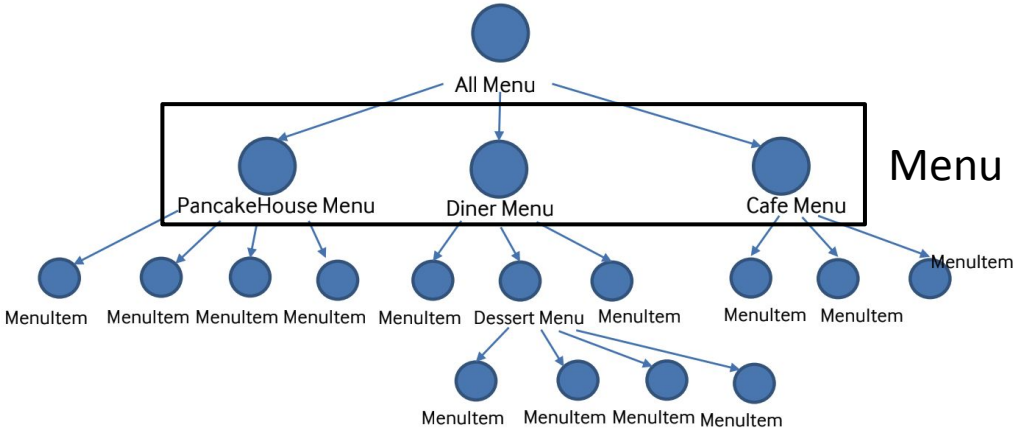
```
public class Client {
    public static void main(String[] args){
        int permit = 0;

        GitUser proxyGitUser =
                new ProxyGitUser(permit);
        proxyGitUser.push();
    }
}
```

Composite

- Composite Pattern allows you to compose objects into **tree structures** to represent **part-whole hierarchies**
- Composite Pattern lets clients treat individual objects and compositions of objects **uniformly**
- Use when
  - facilitates the creation of object hierarchies where each object can be treated independently or as a set of nested objects

Problem

- We want to represent and use the hierarchical structure

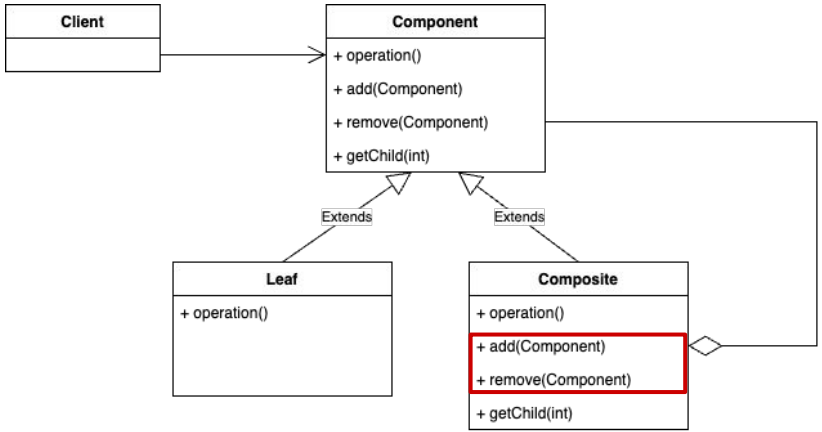


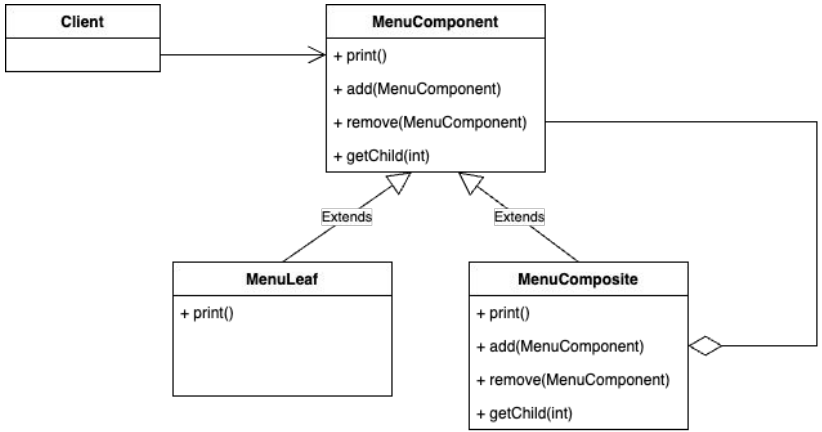

```
if (current instanceOf(MenuItem))
    // handle MenuItem in a way
else if (current instanceOf(Menu))
    // handle Menu in another way
```

non-uniform access

Solution

- Compose one or more objects with the **same interface**
  - Goal: operate on composite as if it was a unit type
  - We can manipulate composite/unit types all in the same way
- Objects must be similar, and exhibit similar functionality





Code with Composite

```
public class MenuComponent {
    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public void remove(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }
    public void print() {
        throw new UnsupportedOperationException();
    }
}
```

```
public class MenuLeaf extends MenuComponent {
    ... // Attributes and constructors
    public String getName() { return name; }
    public String getDescription() {
        return description;
    }
    public double getPrice() { return price;}
    public void print() {
        System.out.print(" " + getName());
        System.out.println(", " + getPrice());
        System.out.println(" --" +
getDescription());
    }
}
```

Code with Composite

```
public class MenuComposite extends MenuComponent{
    ArrayList<MenuComponent> menuChildren = new ArrayList<MenuComponent>();
    ... // Other attributes, constructors
    public void add(MenuComponent menuComponent) { menuChildren.add(menuComponent); }
    public void remove(MenuComponent menuComponent) { menuChildren.remove(menuComponent); }
    public MenuComponent getChild(int i) { return (MenuComponent)menuChildren.get(i); }
    public String getName() { return name; }
    public String getDescription () { return description; }
    public void print() {
        Iterator iterator = menuChildren.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent = (MenuComponent)iterator.next();
            menuComponent.print();
        }
    }
}
```

Composition

Methods for composite

We can treat *MenuComosite* and *MenuLeaf* uniformly

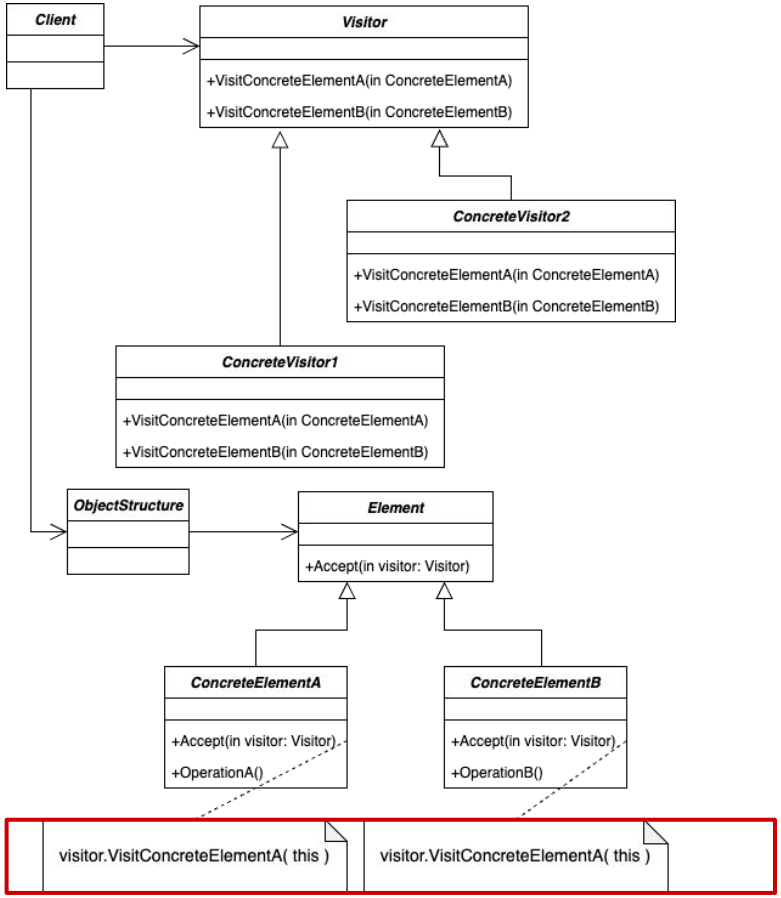
Behavioral Patterns

- Behavioral patterns concern how classes and objects interact and distribute responsibility
- List of behavioral patterns
  - Visitor
  - Iterator
  - Observer
  - Strategy

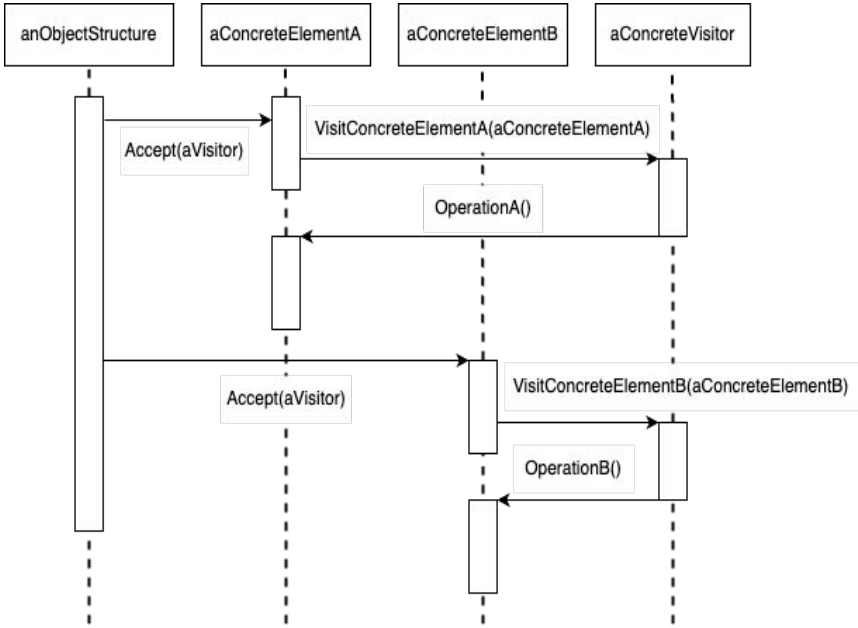
Visitor

- Visitor Pattern allows to perform one or more operations at runtime
- It decouples the operations from the object structure
- It makes it easy to add operations outside of the class's responsibilities
- Use when
  - an object structure must have unrelated operations performed
  - operations must be performed on the concrete classes of an object structure

Visitor



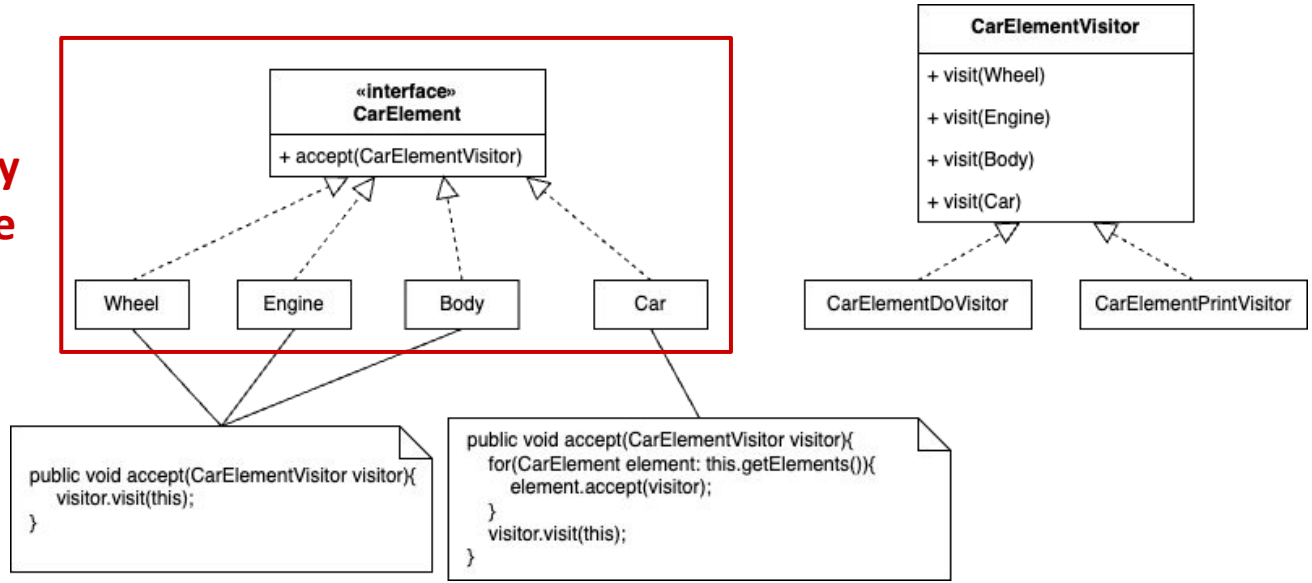
Flow of visitor



Example of Visitor

- We want to add methods for printing description or performing something

We cannot modify these source code



Code without Visitor

```
public interface CarElement {
    void run();
}

class Wheel implements CarElement {
    void run() {
        ...
    }
}
```

```
class Engine implements CarElement{
    void run() {
        ...
    }
}

class Body implements CarElement{
    void run() {
        ...
    }
}
```

Code without Visitor

```
class Car implements CarElement {
    CarElement[] elements;
    public Car() {
        this.elements = new CarElement[] {
            new Wheel("front left"), new Wheel("front right"),
            new Wheel("back left") , new Wheel("back right"),
            new Body(),
            new Engine() };
    }
    public void run(){
        for(CarElement elem: elements)
            elem.run()
    }
}
```

What if we want to add a new function besides *run()* to `CarElement` classes (e.g., Wheel, Engine)?

```
public class Client {
    public static void main(String[] args) {
        CarElement car = new Car();
        car.run();
    }
}
```

Code with Visitor

```
public interface CarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
}
public interface CarElement {
    void run();
    void accept(CarElementVisitor visitor);
}
class Wheel implements CarElement {
    void run(){ ... }
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

```
class Engine implements CarElement {
    void run(){ ... }
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}
class Body implements CarElement {
    void run(){ ... }
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

Code with Visitor

```
class Car implements CarElement {
    CarElement[] elements;
    public Car() {
        this.elements = new CarElement[] {
            new Wheel("front left"), new Wheel("front right"),
            new Wheel("back left") , new Wheel("back right"),
            new Body(),
            new Engine() };
    }
    public void run() { ... }
    public void accept(CarElementVisitor visitor) {
        for(CarElement elem : elements)
            elem.accept(visitor);
        visitor.visit(this);
    }
}
```

- You can perform a new operation with accept() function in the visitor interface
- You can simply create a class that implements *CarElementVisitor*

Code with Visitor

```
class CarElementPrintVisitor implements CarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Visiting " +
                           wheel.getName() + " wheel");
    }
    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }
    public void visit(Body body) {
        System.out.println("Visiting body");
    }
    public void visit(Car car) {
        System.out.println("Visiting car");
    }
}
```

```
class CarElementDoVisitor implements CarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Kicking my " + wheel.getName()
                                    + " wheel"); }
    public void visit(Engine engine) {
        System.out.println("Starting my engine");
    }
    public void visit(Body body) {
        System.out.println("Moving my body");
    }
    public void visit(Car car) {
        System.out.println("Starting my car");
    }
}
```

Flow of Visitor

```
public class Client {
    public static void main(String[] args) {
        CarElement car = new Car();
        car.accept(new CarElementPrintVisitor());
        car.accept(new CarElementDoVisitor());
    }
}
```

```
[OUTPUT]
Visiting front left wheel
Visiting front right wheel
Visiting back left wheel
Visiting back right wheel
Visiting body
Visiting engine
Visiting car
Kicking my front left wheel
Kicking my front right wheel
Kicking my back left wheel
Kicking my back right wheel
Moving my body
Starting my engine
Starting my car
```

Flow of the program

```
car.accept(new CarElementPrintVisitor());
```

Car::accept()

```
for(CarElement elem : elements)
    elem.accept(visitor);
```

Engine::accept()

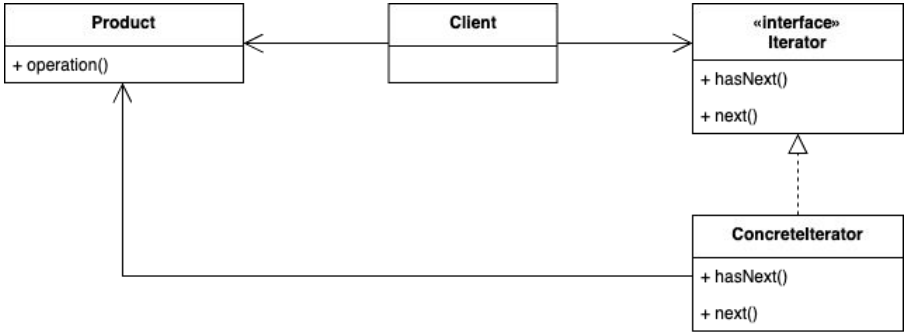
```
public void accept(CarElementVisitor visitor) {
    visitor.visit(this);
}
```

CarElementVisitor::visit()

```
class CarElementPrintVisitor implements
CarElementVisitor {
    ...
    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }
```

Iterator

- Iterator Pattern provides a way to access the elements of an aggregate object **sequentially** without exposing its underlying details
- Use when
  - a uniform interface for traversal is needed
  - access to elements is needed without access to the entire details



Code without Iterator (1/2)

```
public class Menu {
    String name;
    int price;
    public Menu(String name, int price){
        this.name = name;
        this.price = price;
    }
    public String getName(){
        return name;
    }
    public int getPrice(){
        return price;
    }
}
```

```
public class Client {
    public static void main(String[] args){
        ArrayList<Menu> menuArrayList = new ArrayList<Menu>();
        menuArrayList.add(new Menu("Pizza", 14));
        menuArrayList.add(new Menu("Burger", 10));
        menuArrayList.add(new Menu("Salad", 7));

        Menu[] menuArray = new Menu[3];
        menuArray[0] = new Menu("Chips", 7);
        menuArray[1] = new Menu("Sandwich", 9);
        menuArray[2] = new Menu("Ribs", 17);
        ...
```

Code without Iterator (2/2)

- Clients need to know the underlying details of the data structure to iterate through it

```
        ...
        for(int i = 0; i < menuArrayList.size(); i++){
            Menu menu = menuArrayList.get(i);
            System.out.println(menu.getName() + " " + menu.getPrice());
        }

        for(int i = 0; i < menuArray.length; i++){
            Menu menu = menuArray[i];
            System.out.println(menu.getName() + " " + menu.getPrice());
        }
    }
}
```

Code with Iterator (1/3)

```
public interface Iterator {
    boolean hasNext();
    Menu next();
}
```

```
public class ArrayIterator implements Iterator{
    Menu[] menuArray;
    int position = 0;
    public ArrayIterator (Menu[] menuArray) {
        this.menuArray = menuArray;
    }
    ...
```

```
    ...
    @Override
    public boolean hasNext() {
        if (position >= menuArray.length)
            return false;
        else
            return true;
    }
    @Override
    public Menu next() {
        Menu menu = menuArray[position];
        position ++;
        return menu;
    }
}
```

Code with Iterator (2/3)

```
public class ArrayListIterator implements Iterator{
    ArrayList<Menu> menuArrayList;
    int position = 0;
    public ArrayListIterator(ArrayList<Menu>
menuArrayList){
        this.menuArrayList = menuArrayList;
    }
    ...
```

```
    ...
    @Override
    public boolean hasNext() {
        if (position >= menuArrayList.size())
            return false;
        else
            return true;
    }
    @Override
    public Menu next() {
        Menu menu = menuArrayList.get(position);
        position ++;
        return menu;
    }
}
```

Code with Iterator (3/3)

```
public class Client {
    public static void main(String[] args){
        ... // Data initialization
        ArrayListIterator arrayListIterator = new ArrayListIterator(menuArrayList);
        ArrayIterator arrayIterator = new ArrayIterator(menuArray);
        while(arrayListIterator.hasNext()){
            Menu menu = arrayListIterator.next();
            System.out.println(menu.getName() + " " + menu.getPrice());
        }
        while(arrayIterator.hasNext()){
            Menu menu = arrayIterator.next();
            System.out.println(menu.getName() + " " + menu.getPrice());
        }
    }
}
```

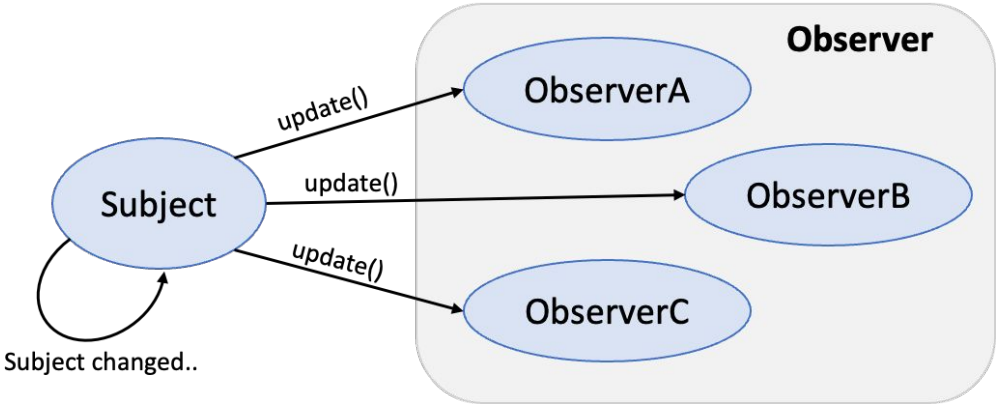
Client can iterate through the data structure with a common interface

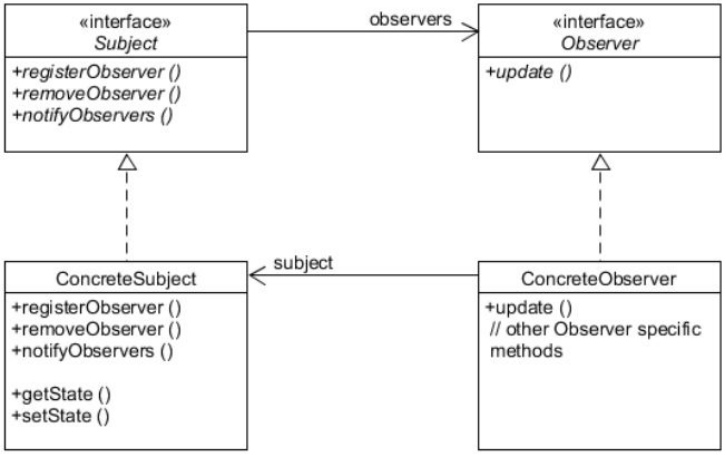
Observer

- Observer Pattern defines a **one-to-many dependency** between objects so that when an object changes state, all of its dependents are notified and updated automatically
- Use when
  - state changes in one or more objects should trigger behavior in other objects
  - loose coupling is needed for communications

Example of Observer

- We want to send notifications to our subscribers





Code with Observer

```
public interface Subject {
    public void registerObserver(Observer o);
    public void notifyObservers();
}
public interface Observer {
    public void update(float temp);
}
```

```
public class Thermometer implements Subject {
    private ArrayList observers;
    private float temperature;
    public Thermometer() {
        observers = new ArrayList();
} ...
```

```
    ...
    public void registerObserver(Observer o) {
        observers.add(o);
    }
    public void notifyObservers() {
        for(int i = 0; i < observers.size(); i++) {
            Observer observer = (Observer)observers.get(i);
            observer.update(temperature);
        }
    }
    public void setMeasurements(float temperature) {
        this.temperature = temperature;
        notifyObservers();
    }
}
```

Code with Observer

```
public class Display implements Observer {
    private float temperature;
    private Subject thermometer;
    public Display(Subject thermometer) {
        this.thermometer = thermometer;
        thermometer.registerObserver(this);
    }
    public void update(float temperature) {
        this.temperature = temperature;
        display();
    }
    public void display() {
        System.out.println("Current temperature: " +
                           temperature + "F degrees");
    }
}
```

```
public class TemperatureStation {
    public static void main(String[] args) {
        Thermometer thermometer = new Thermometer();
        Display display = new Display(thermometer);
        thermometer.setMeasurements(80);
    }
}
```

Output

```
Current temperature: 80F degrees
```

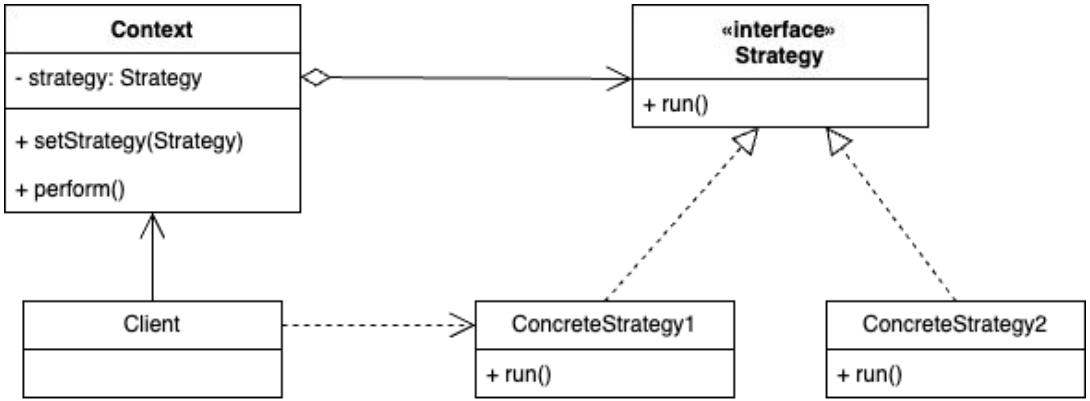
Strategy

- Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable
- We can change the algorithm **independently** from clients
- This pattern is often used to create software libraries
- Use when
  - the only difference between many related classes is their behavior
  - multiple versions or variations of an algorithm are required

Example of Strategy

- Client can easily change and extend the algorithm to use

Basic diagram



Code without Strategy (1/2)

```
public class NavigationContext {
    private void shortestRouteStrategy(String from, String to) {
        System.out.println("Shortest path");
        System.out.println(from + " to " + to);
    }

    private void leastTrafficRouteStrategy(String from, String to) {
        System.out.println("Least traffic route");
        System.out.println(from + " to " + to);
    }
    ...
```

Code without Strategy (2/2)

```
    ...
    public void perform(String from, String to, String routeType) {
        if ("Shortest".equals(routeType)) {
            shortestRouteStrategy(from, to);
        } else if ("LeastTraffic".equals(routeType)) {
            leastTrafficRouteStrategy(from, to);
        }
    }
}
```

We should modify the *perform()* method when a new strategy is added

```
public class Client {
    public static void main(String[] args) {
        NavigationContext navigation = new NavigationContext();
        navigation.perform("Home", "Office", "Shortest");
        navigation.perform("Home", "Gym", "LeastTraffic");
    }
}
```

Code with Strategy (1/3)

```
public interface RouteStrategy {
    void run(String from, String to);
}
```

```
public class ShortestPathStrategy implements RouteStrategy {
    @Override
    public void run(String from, String to) {
        System.out.println("Shortest path");
        System.out.println(from + " to " + to);
    }
}
public class LeastTrafficStrategy implements RouteStrategy {
    @Override
    public void run(String from, String to) {
        System.out.println("Least traffic route");
        System.out.println(from + " to " + to);
    }
}
```

Code with Strategy (2/3)

```
public class NavigationContext {
    private RouteStrategy routeStrategy;

    public void setRouteStrategy(RouteStrategy routeStrategy) {
        this.routeStrategy = routeStrategy;
    }

    public void perform(String from, String to) {
        routeStrategy.run(from, to);
    }
}
```

Code with Strategy (3/3)

- We can create a new class (e.g., `ScenicRouteStrategy`) without any change in the existing code

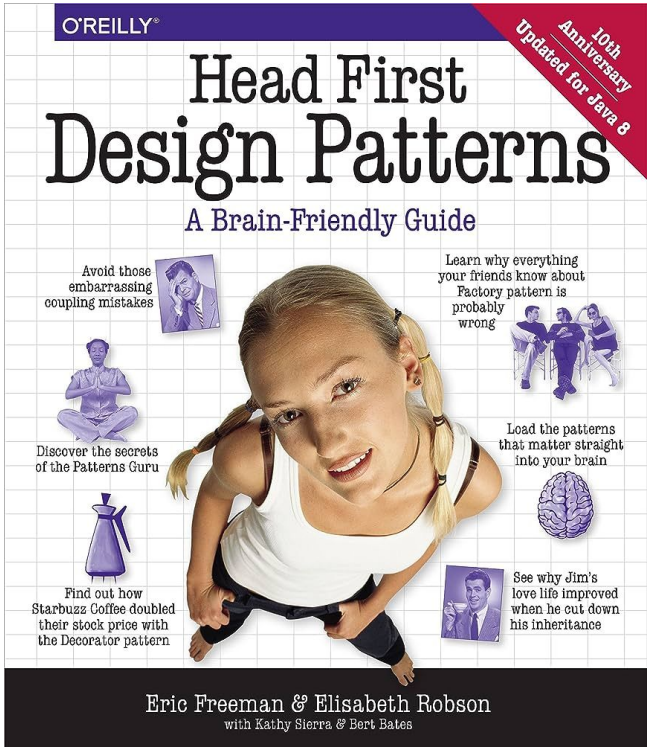
```
public class Client {
    public static void main(String[] args) {
        NavigationContext navigation = new NavigationContext();
        navigation.setRouteStrategy(new ShortestPathStrategy());
        navigation.perform("Home", "Office");

        navigation.setRouteStrategy(new LeastTrafficStrategy());
        navigation.perform("Home", "Gym");
    }
}
```

Summary

- Structural patterns let you compose classes or objects into larger structures
- Behavioral Patterns concern how classes and objects interact and distribute responsibility

References

- Freeman Eric et al. *Head First Design Patterns*. 1st ed. O'Reilly 2004.



Thank You.
Any Questions?