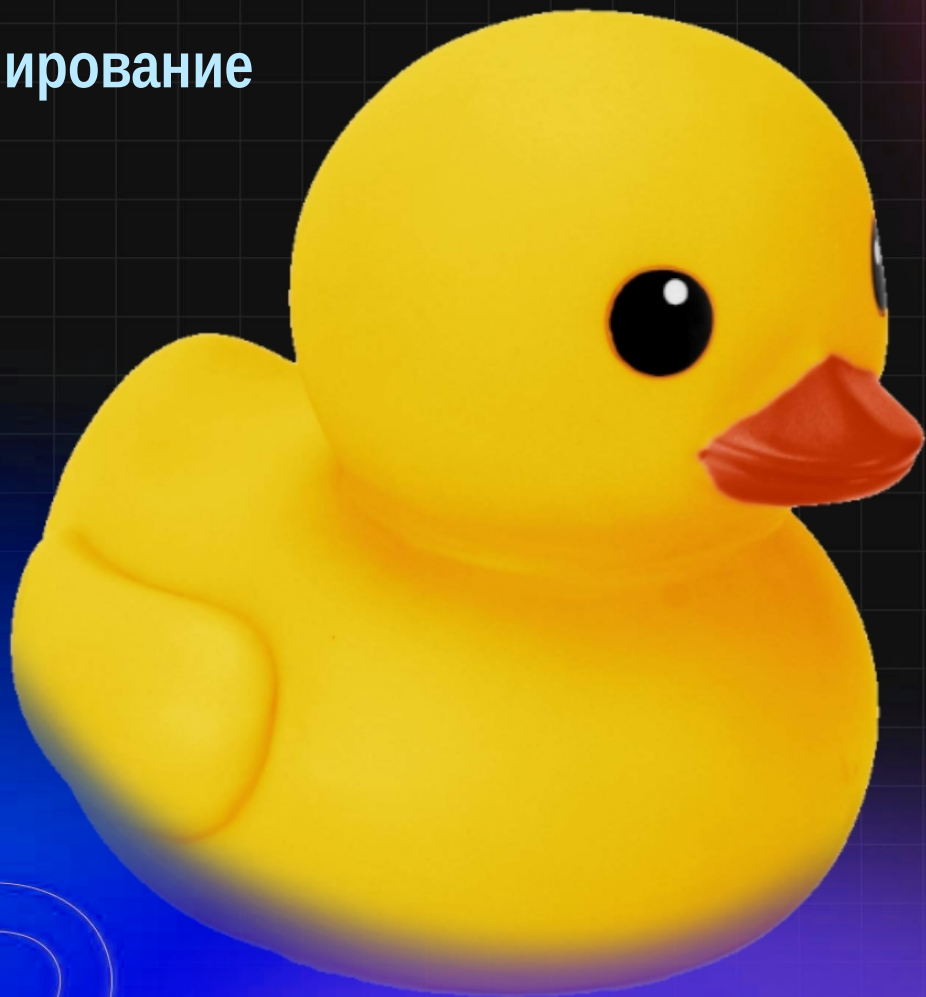


Программирование
2 семестр
2024



ІТМО

Шаблоны
проектирования

- Повторное использование кода
 - ❖ DRY - Don't repeat yourself
 - ❖ стандартные библиотеки
 - ❖ фреймворки
- Расширяемость
 - ❖ Учет возможных будущих изменений

- Все меняется
 - ❖ требования заказчика
 - ❖ пожелания заказчика
 - ❖ версии библиотек
 - ❖ операционные системы
 - ❖ новые устройства
 - ❖ взгляды разработчика
 - ❖ внешние условия
 - ❖ находятся баги

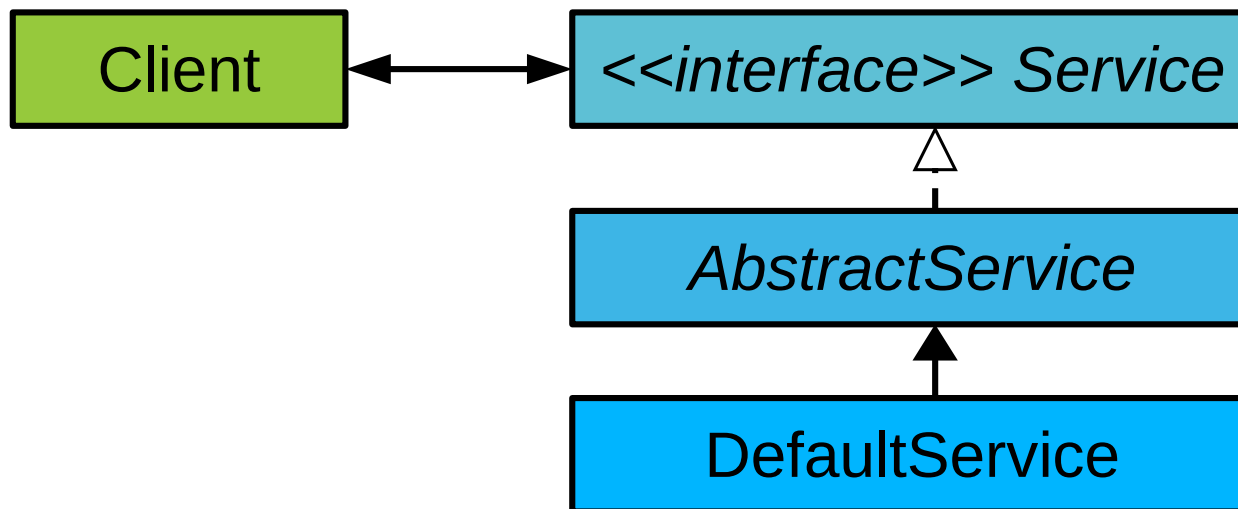
Программы меняются

- Кто-то когда-то уже делал что-то похожее
- Пришлось вносить изменения - возникли проблемы
- Нужно сразу было делать по-другому!

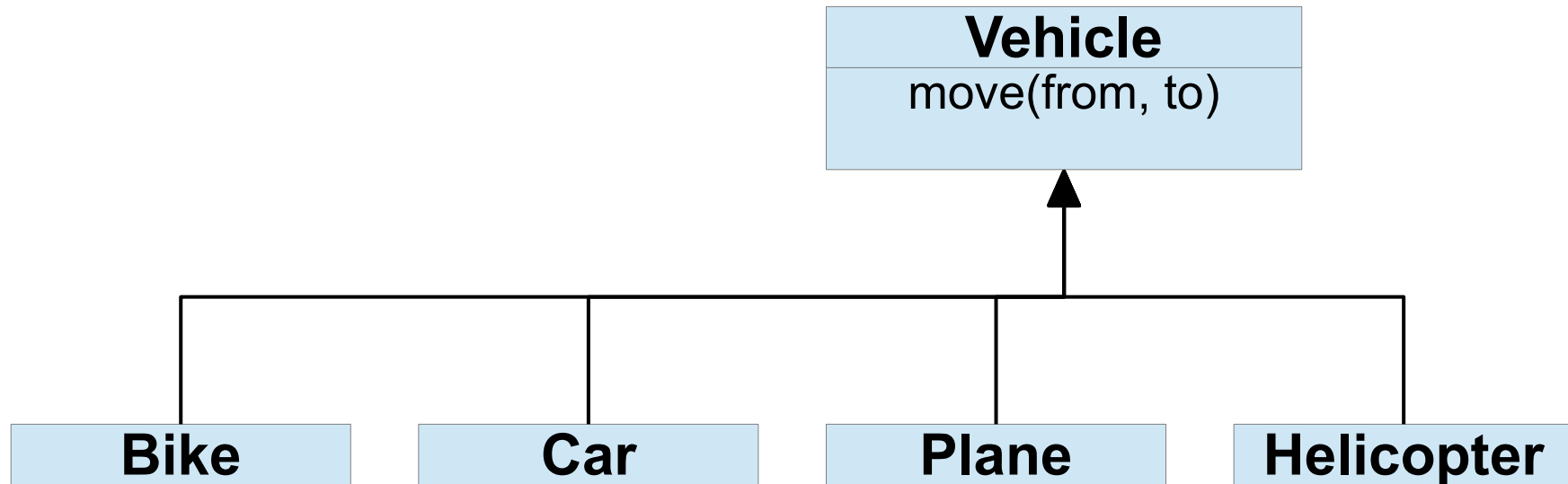


GoF book

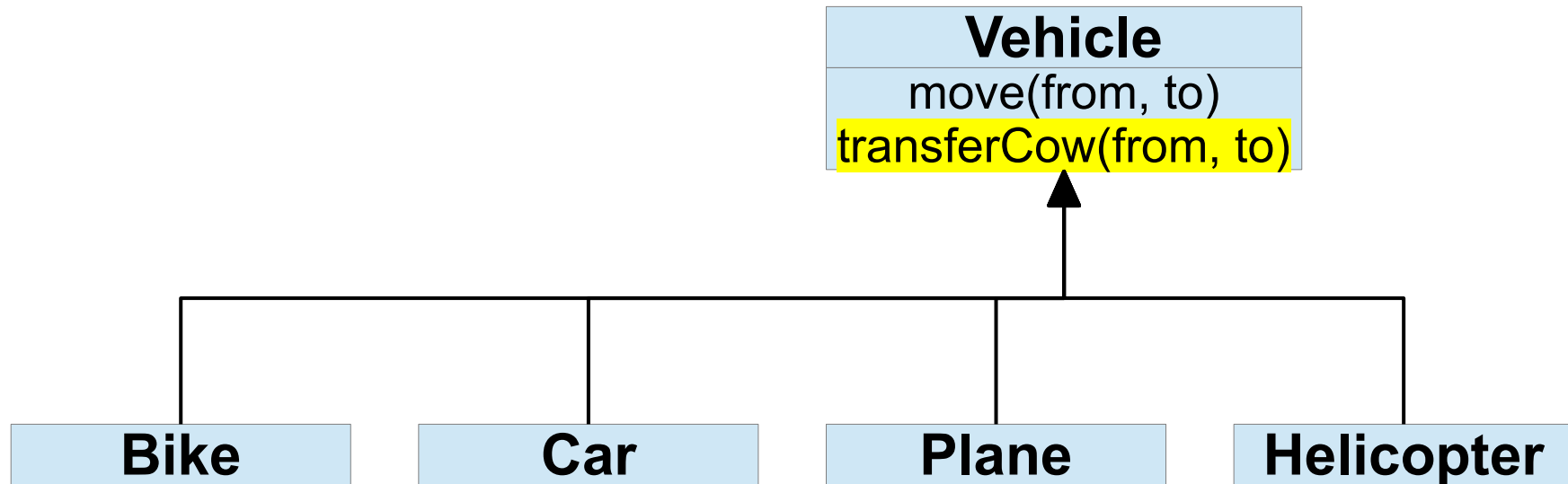
- Взаимодействие с абстракцией - ИНТЕРФЕЙС



- Умеет предок - умеют потомки
- Полиморфизм



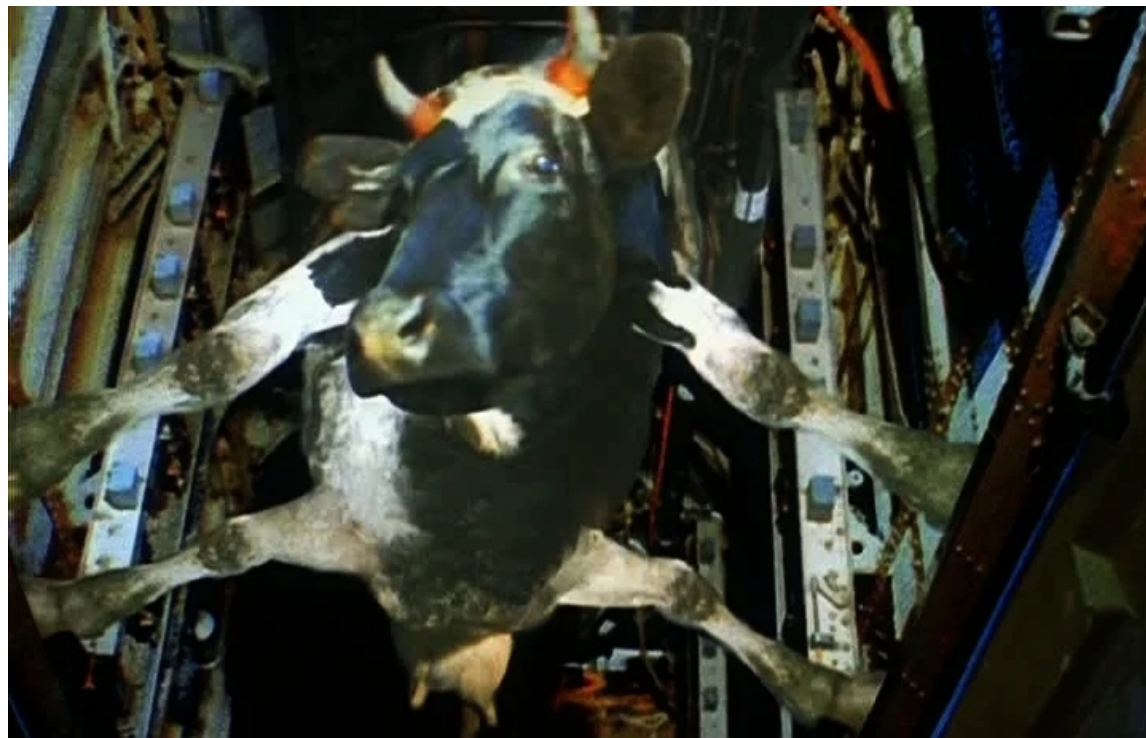
- Перевозка коровы - наследование??





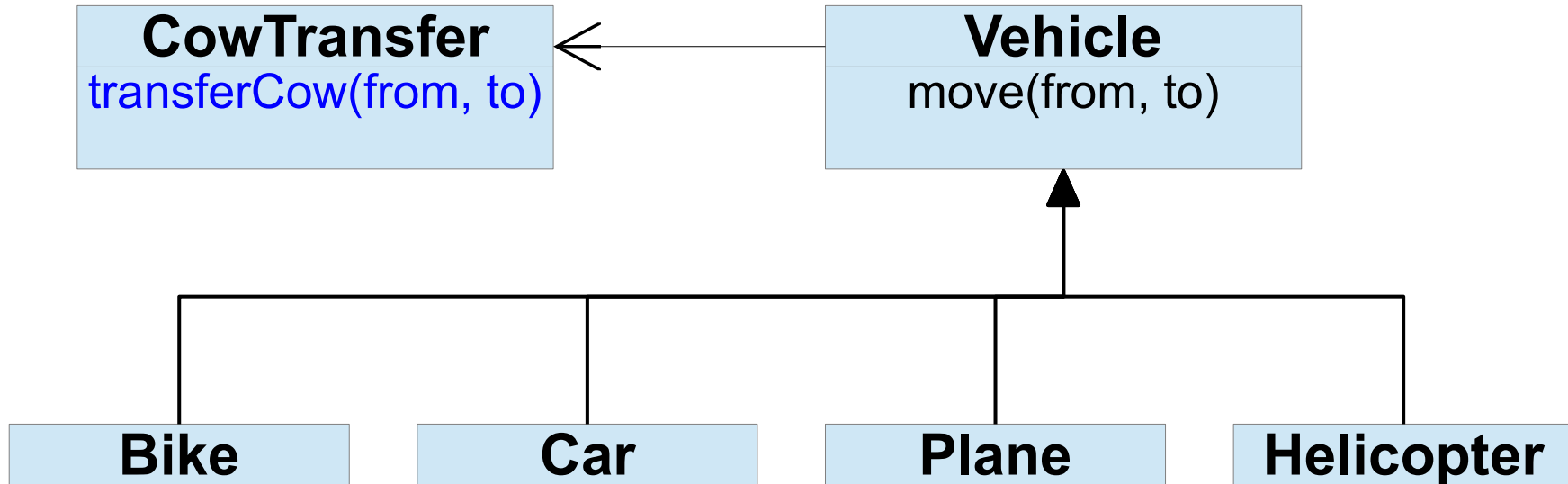
(С) х/ф «Мимино»

Перевозка коровы





- Перевозка коровы - делегирование



- Наследование

- ❖ статическое отношение
- ❖ классификация

```
class Animal {  
    sleep() { ... }  
}  
  
class Cat extends Animal { }  
  
Animal animal = new Cat();  
animal.sleep();
```

- Делегирование

- ❖ динамическое отношение
- ❖ роль

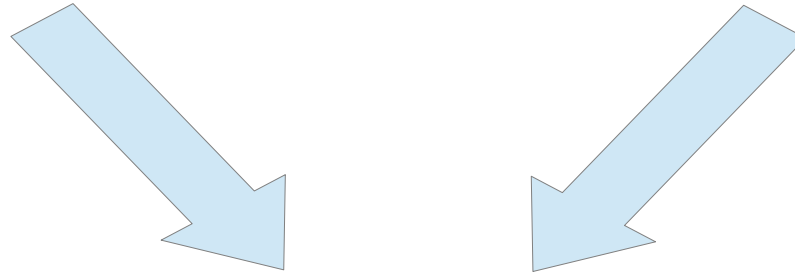
```
class Vehicle {  
    move(from, to) { ... }  
}  
  
class CowTransfer {  
    Vehicle vehicle; Cow cow;  
    transfer(from, to) {  
        attach(cow, vehicle);  
        vehicle.move(from, to);  
        detach(cow, vehicle);  
    }  
}
```

Инкапсуляция изменений

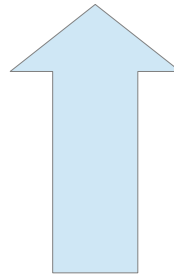
- Отделить изменяющееся от постоянного
- Инкапсулировать изменяющееся

Интерфейсы

Делегирование

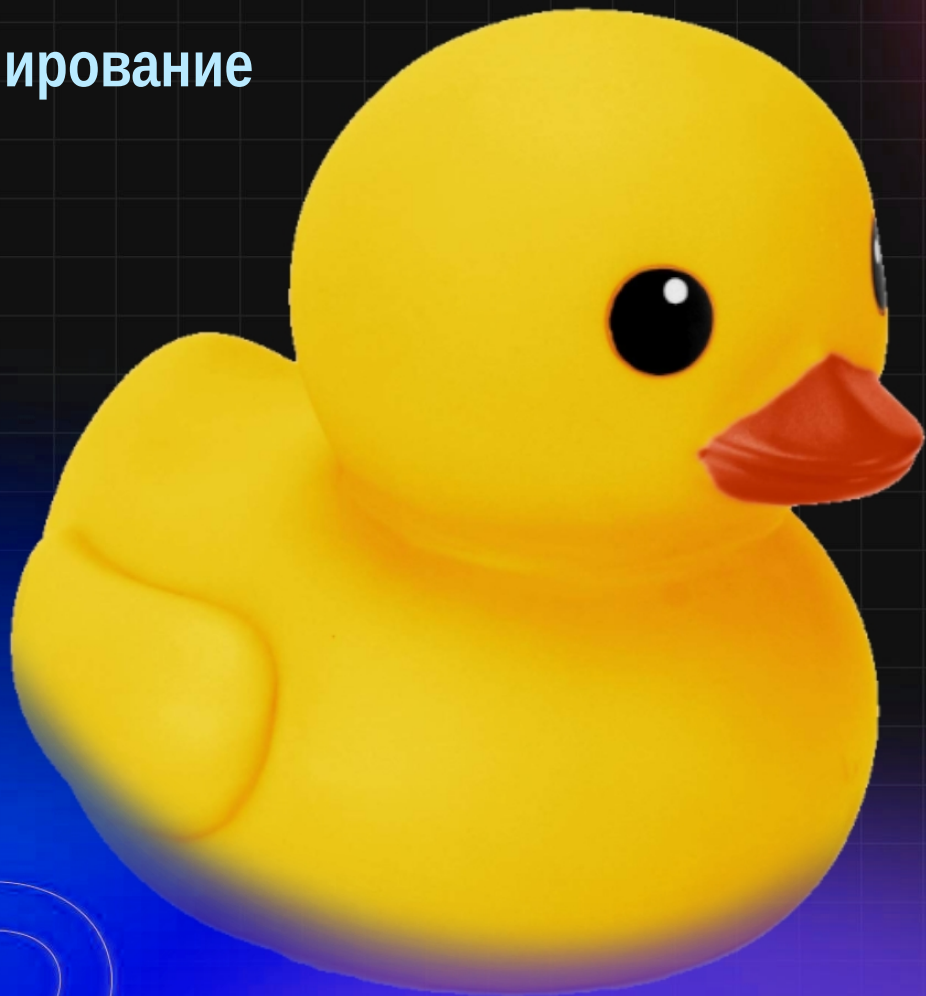


**ШАБЛОНЫ
PATTERNS**



Инкапсуляция изменений

Программирование
2 семестр
2024



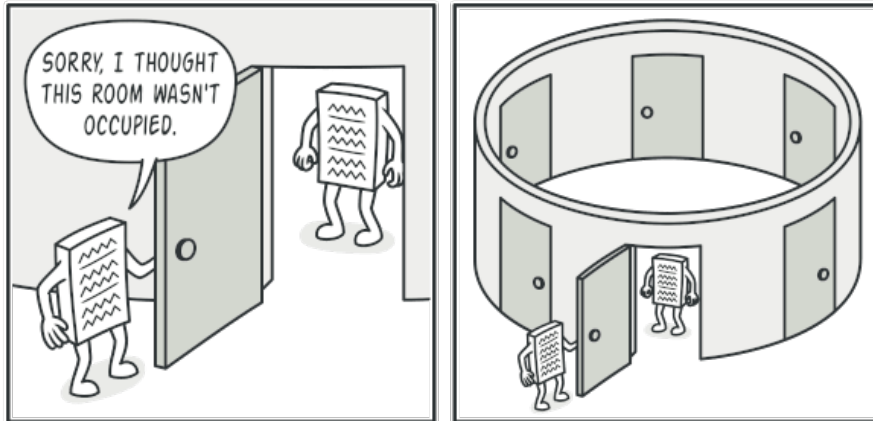
ІТМО

**Порождающие
шаблоны
(Creational)**

- Singleton - Одиночка
- Object Pool — Пул объектов
- Factory Method — Фабричный метод
- Abstract Factory — Абстрактная фабрика
- Prototype — Прототип
- Builder — Строитель

- Одиночка

- ❖ Гарантирует наличие единственного экземпляра класса
- ❖ Предоставляет глобальную точку доступа



Singleton
-instance : Singleton
-Singleton()
+Instance(): Singleton

```
class Singleton {  
    private final static Singleton INSTANCE = new Singleton();  
  
    private Singleton() { }  
  
    public static Singleton getInstance() {  
        return INSTANCE;  
    }  
}
```

Singleton

-instance : Singleton

-Singleton()

+Instance(): Singleton

- Варианты
 - ❖ Ленивая инициализация
 - ❖ Синхронизация при многопоточности
- Использование
 - ❖ логгер
 - ❖ конфигуратор
 - ❖ фабрика
 - ❖ `java.lang.Runtime`

Singleton
-instance : Singleton
-Singleton() +Instance(): Singleton

Singleton с ленивой инициализацией

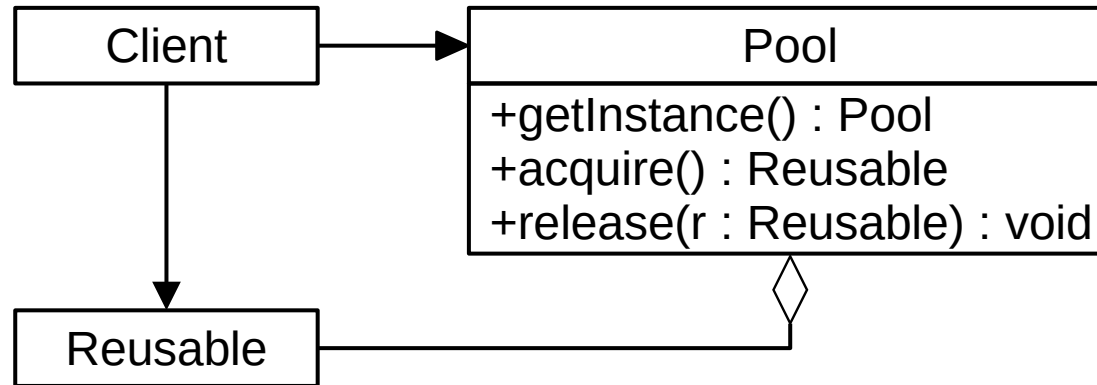
```
class Singleton {
    private static Singleton instance;

    private Singleton() { }

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Singleton
-instance : Singleton
-Singleton()
+Instance(): Singleton

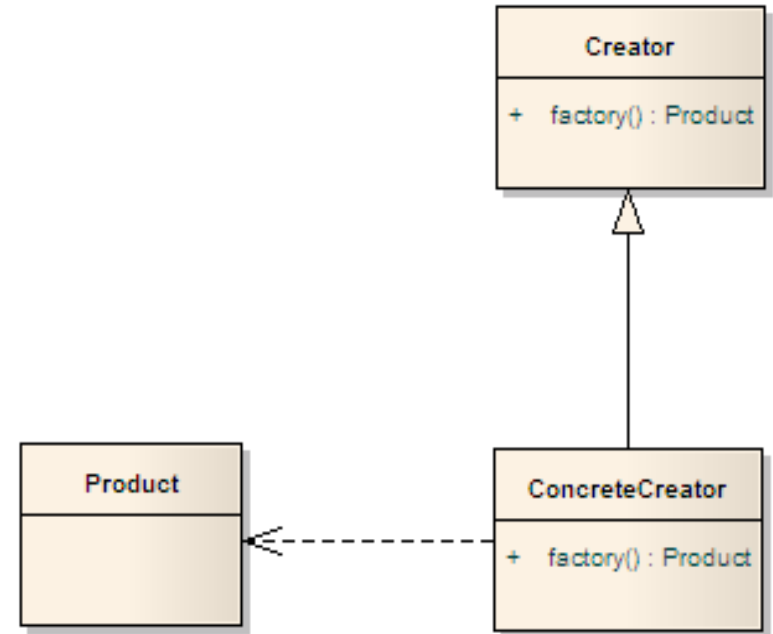
- Почти как Singleton, только объектов больше одного
 - ❖ Применяется когда создание объекта требует ресурсов
 - ❖ После использования объект возвращается в пул



Simple Factory

- Простая фабрика (не совсем паттерн)
 - ❖ Объект, создающий другие объекты

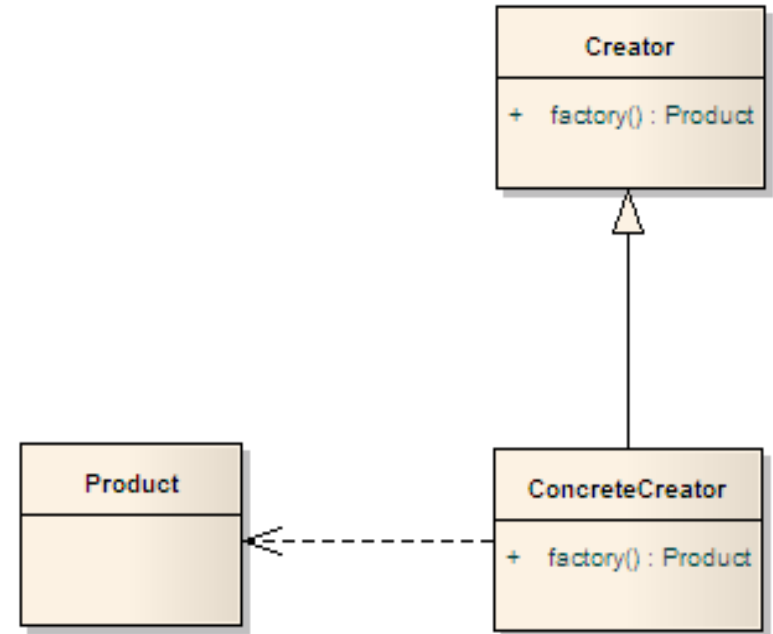
```
Animal[] animals = new Animal[2];
animals[0] = new Dog();
animals[1] = new Cat();
for (Animal a : animals) {
    a.sleep();
    a.makeSound();
}
```



Simple Factory

- Простая фабрика (не совсем паттерн)
 - ❖ Объект, создающий другие объекты

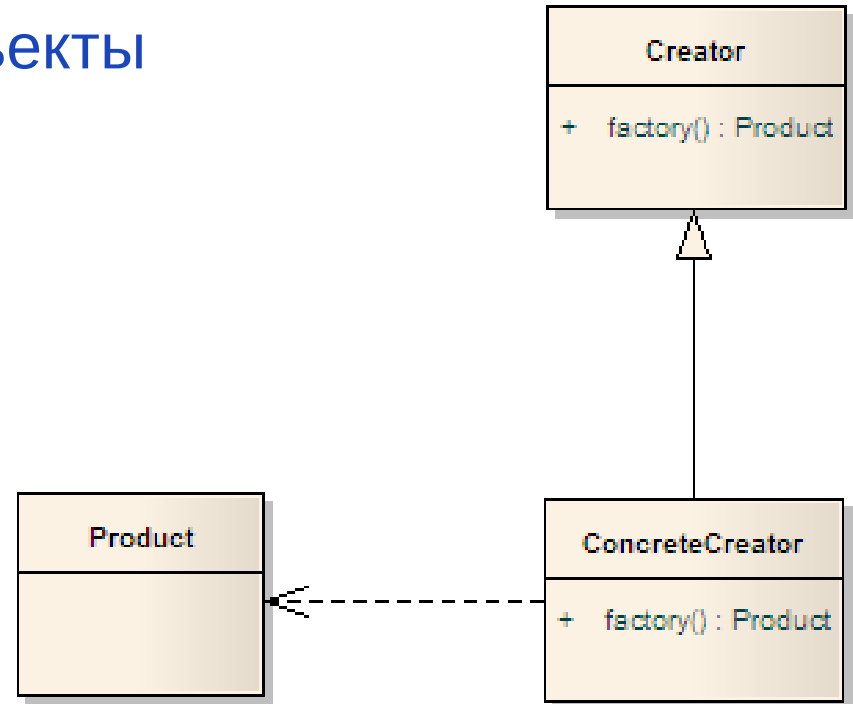
```
class Creator {  
    public static Product get(int type) {  
        return switch(type) {  
            case 1 -> product1;  
            case 2 -> product2;  
        }  
    }  
}
```



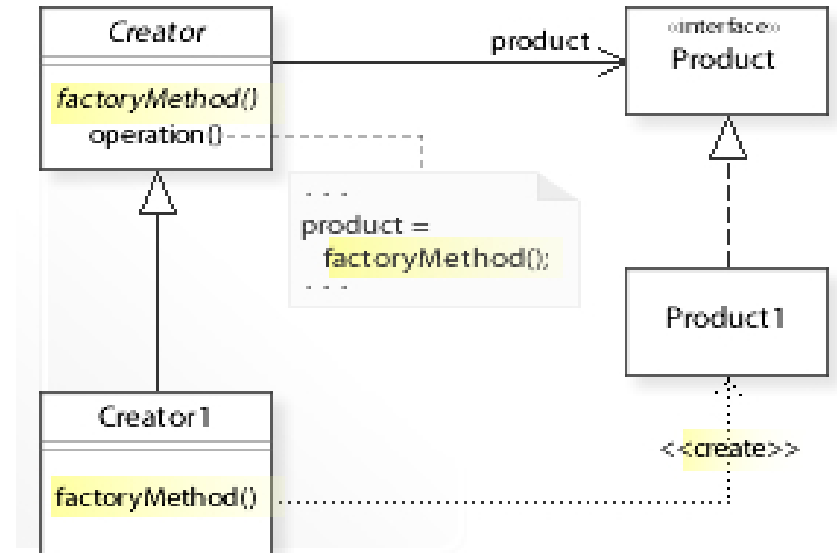
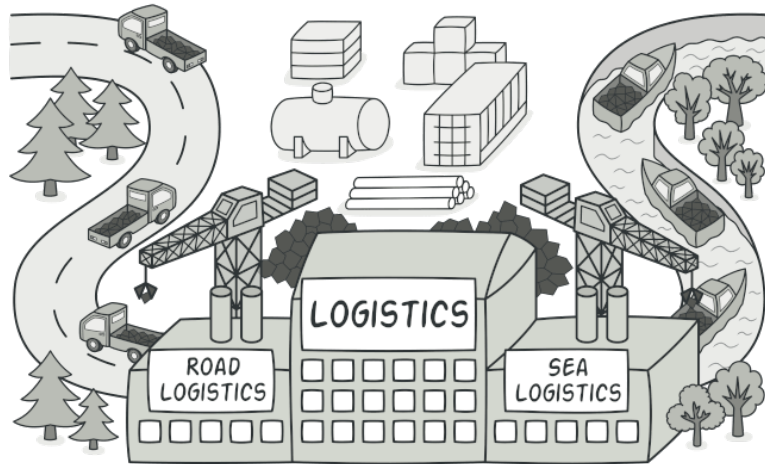
Simple Factory

- Простая фабрика (не совсем паттерн)
 - ❖ Объект, создающий другие объекты

```
Animal[] animals = new Animal[2];
animals[0] = animalFactory.get(1);
animals[1] = animalFactory.get(2);
for (Animal a : animals) {
    a.sleep();
    a.makeSound();
}
```

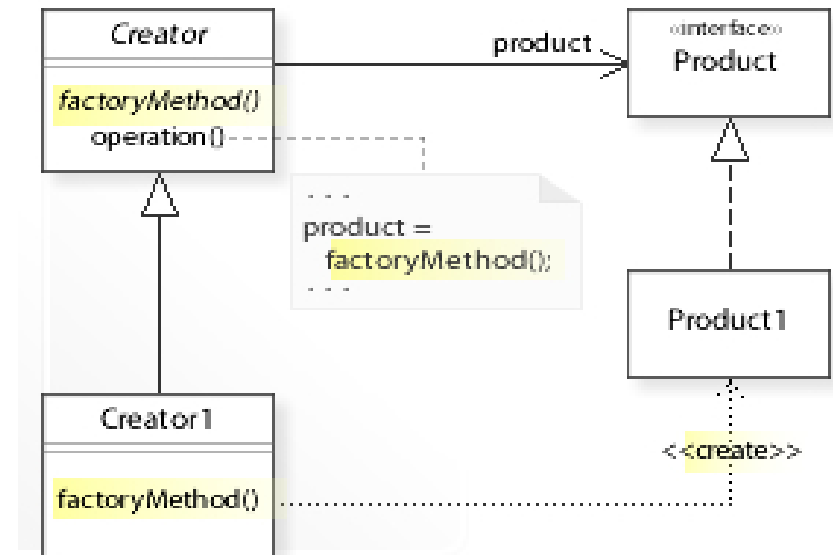


- Фабричный метод
 - ❖ Создание объектов отдаем подклассам



Factory Method

- Фабричный метод
 - ❖ Создание объектов отдаем подклассам
- Product - общий интерфейс
- Product1 - конкретный продукт
- Creator - интерфейс фабрики
 - ❖ Product factoryMethod()
 - ❖ другие методы...
- Creator1 - создает Product1



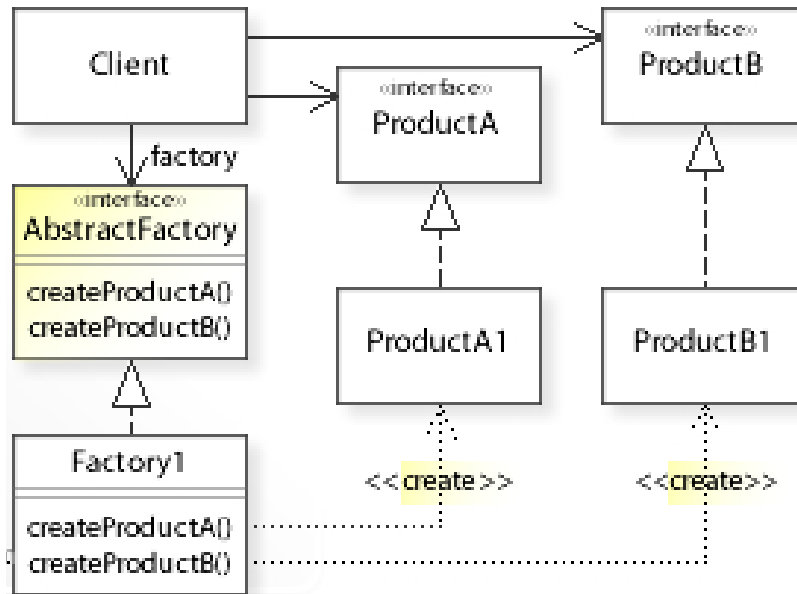
- Фабричный метод
 - ❖ Statement stat = connection.createStatement()

```
class Creator {  
    protected abstract Product getProduct();  
}  
  
class Creator1 {  
    public Product getProduct() {  
        return new Product1();  
    }  
}
```

- Убирает зависимость от конкретных продуктов
- Упрощает добавление новых продуктов
- На каждый продукт нужен свой создатель

Abstract Factory

- Абстрактная фабрика
 - ❖ Фабрика для создания семейства продуктов



- Абстрактные продукты — интерфейсы продуктов разных видов (кола, апельсин, лимон)
- Конкретные продукты — классы различных видов и семейств (миринда, спрайт)
- Абстрактная фабрика объявляет методы для создания продуктов разного вида
- Конкретные фабрики умеют создавать все виды продуктов одного семейства (фабрика пепси)



Abstract Factory

- Абстрактная фабрика
 - ❖ Фабрика для создания семейства продуктов

```
Factory f1 = Provider.getFactory("Coca-Cola");  
Drink d1 = f1.getOrangeDrink(); // Фанта
```

```
Factory f2 = Provider.getFactory("Pepsi");  
Drink d2 = f2.getLemonDrink(); // 7-up
```



- Абстрактная фабрика

- ❖ `Connection conn = DriverManager.getConnection(args)`

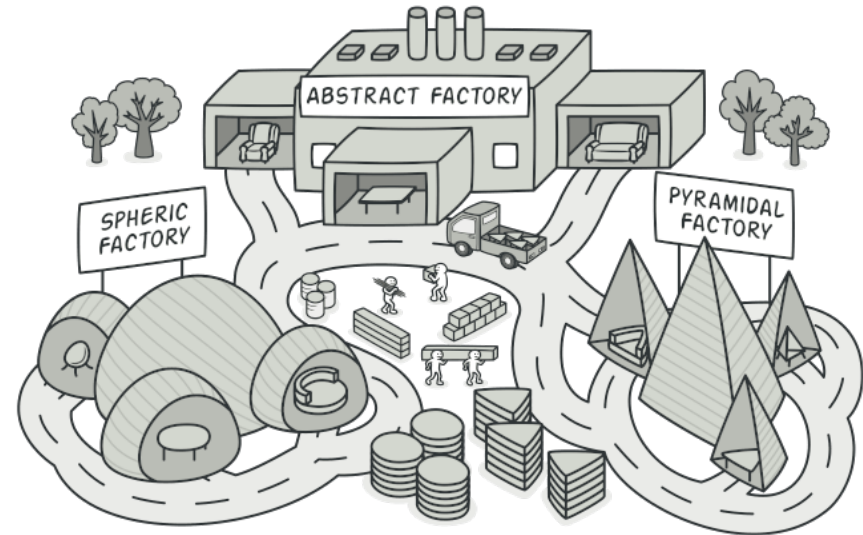
```
Factory f1 = Provider.getFactory("Coca-Cola");  
Drink d1 = f1.getOrangeDrink(); // Фанта
```

```
Factory f2 = Provider.getFactory("Pepsi");  
Drink d2 = f2.getLemonDrink(); // 7-up
```

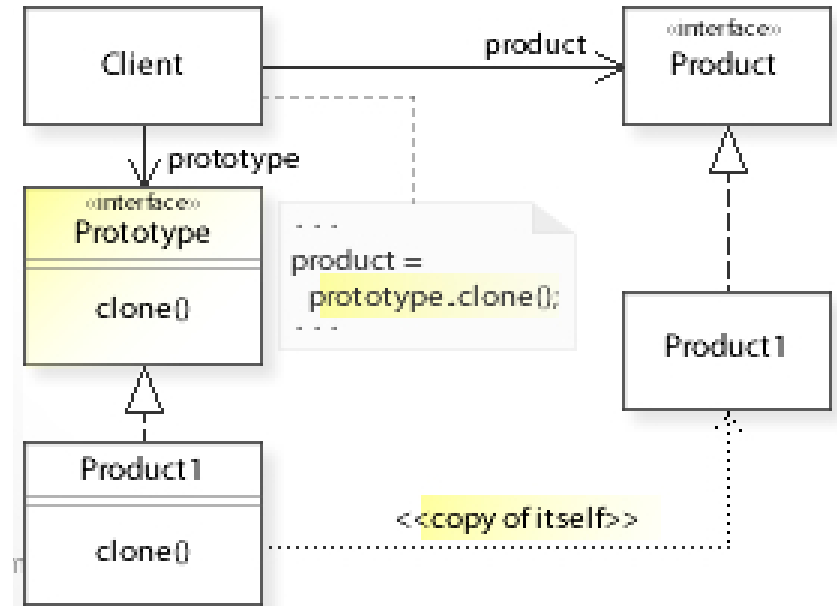
```
Factory f3 = Provider.getFactory("Добрый");  
Drink d3 = f3.getColaDrink(); // Добрый кола
```



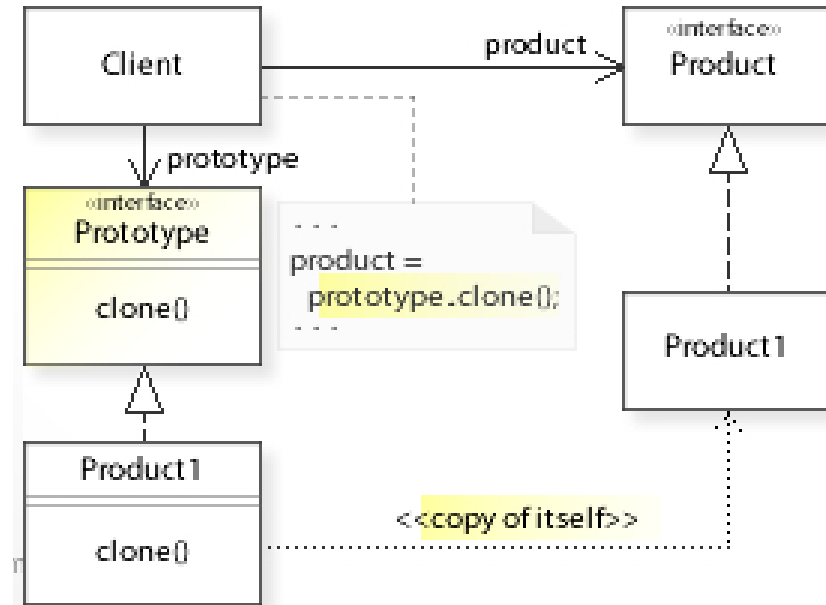
- Устраняет зависимость от конкретных продуктов
- Гарантирует совместимость продуктов в наборе
- Требуется наличие всех типов продуктов в семействе
- Более сложный код



- Прототип
 - ❖ Клонирование вместо создания



- Интерфейс Prototype задает операцию клонирования
- Конкретный прототип реализует операцию клонирования самого себя
- Клиент создаёт копию объекта, вызывая метод клонирования через интерфейс прототипа
- Object.clone(), Cloneable

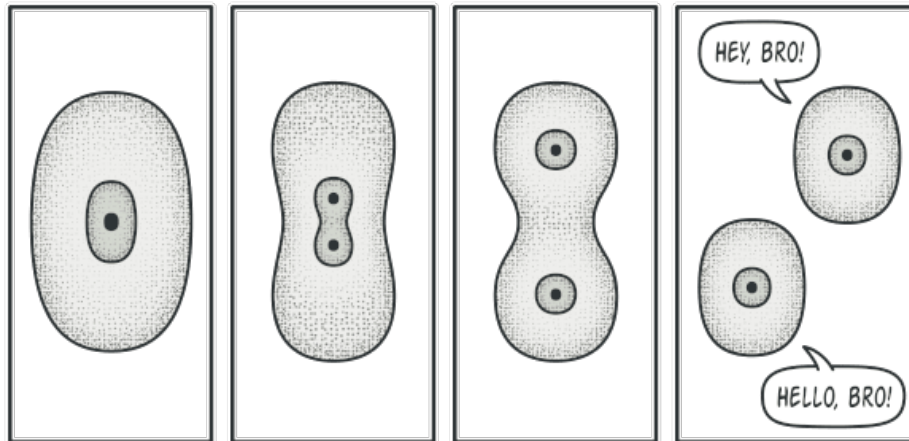


- Глубокая и неглубокая копия (deep / shallow copy)

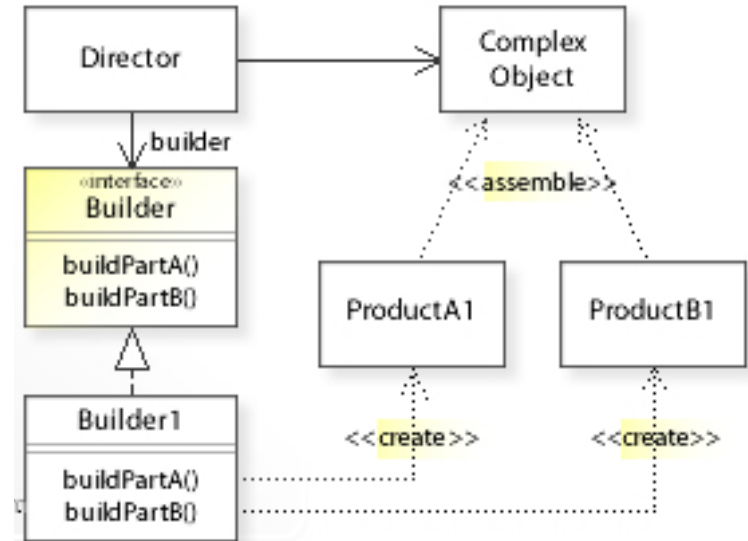
```
Trooper trooper = new Trooper();  
trooper.setHP(999);  
trooper.setSpeed(888);  
trooper.setPower(777);  
trooper.setAttack("Laser beam");
```

```
var army = new ArrayList<Trooper>();  
for (i = 0; i < 99; i++) {  
    var oneMore = trooper.clone();  
    army.add(oneMore);  
}
```

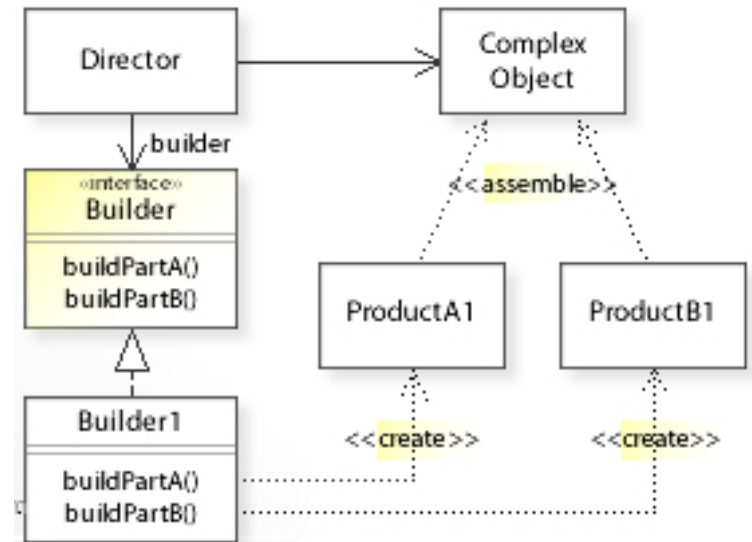
- Клонирование без зависимости от конкретных классов
- Позволяет хранить готовый набор сложных объектов
- Сложно реализовывать глубокое клонирование



- Строитель
 - ❖ Строит сложный объект по шагам



- Интерфейс Builder объявляет этапы создания продуктов
- Конкретные строители реализуют этапы создания
- Продукт — создается строителем при вызове метода create()
- Директор управляет строителем для производства нужной конфигурации продукта



- Класс со множеством полей
- При создании какие-то поля нужно задать
- Возможно не все

```
class Trip {  
    private Location[] locations;  
    private Transport transport;  
    private int numberOfPersons;  
    private Date startDate;  
    private Date finalDate;  
    private Hotel hotel;  
    ...  
}
```


- Телескопические конструкторы - ужас №1

```
class Trip {  
    public Trip(Date date)  
    public Trip(Location location)  
    public Trip(Date date, Location location)  
    public Trip(Date date, Transport transport)  
    public Trip(Date date, Location location, Transport transport)  
    public Trip(Date date, Location location, int guests)  
    public Trip(Date from, Date until, Location location)  
    ...  
}
```

- Аргументы по умолчанию - ужас №2

```
class sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100,),
activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto',
learning_rate='constant', learning_rate_init=0.001, power_t=0.5,
max_iter=200, shuffle=True, random_state=None, tol=0.0001,
verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True,
early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999,
epsilon=1e-08, n_iter_no_change=10, max_fun=15000)
```

- Простой конструктор и набор сеттеров
- Объект не полностью готов в промежуточном состоянии

```
class Trip {  
    public Trip()  
    public setLocation(Location)  
    public setDate(Date)  
    public setTransport(Transport)  
    ...  
}
```

- Строитель создает объект по шагам
- Методы возвращают строителя
- `StringBuilder.append()`

```
Builder b = new Builder()
    .setWhere("Moscow")
    .setTransport("train")
    .setDate(1,7,2023);
    ...

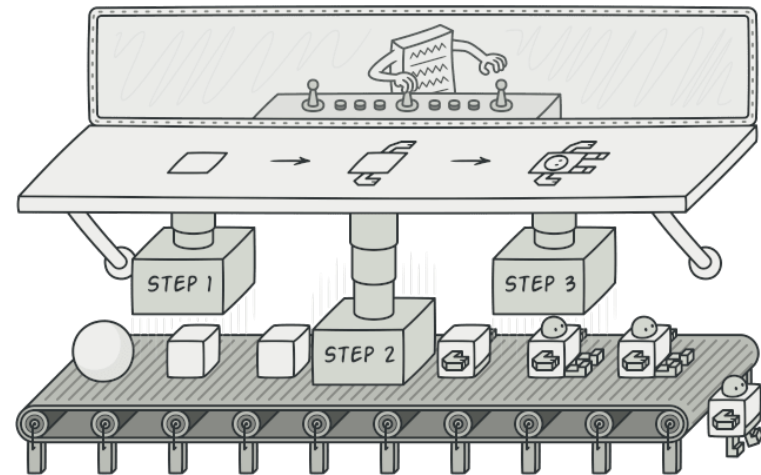
Trip trip = b.createTrip();
```

- Строитель создает объект по шагам
- Методы возвращают строителя
- Директор создает объект с помощью строителя одним методом

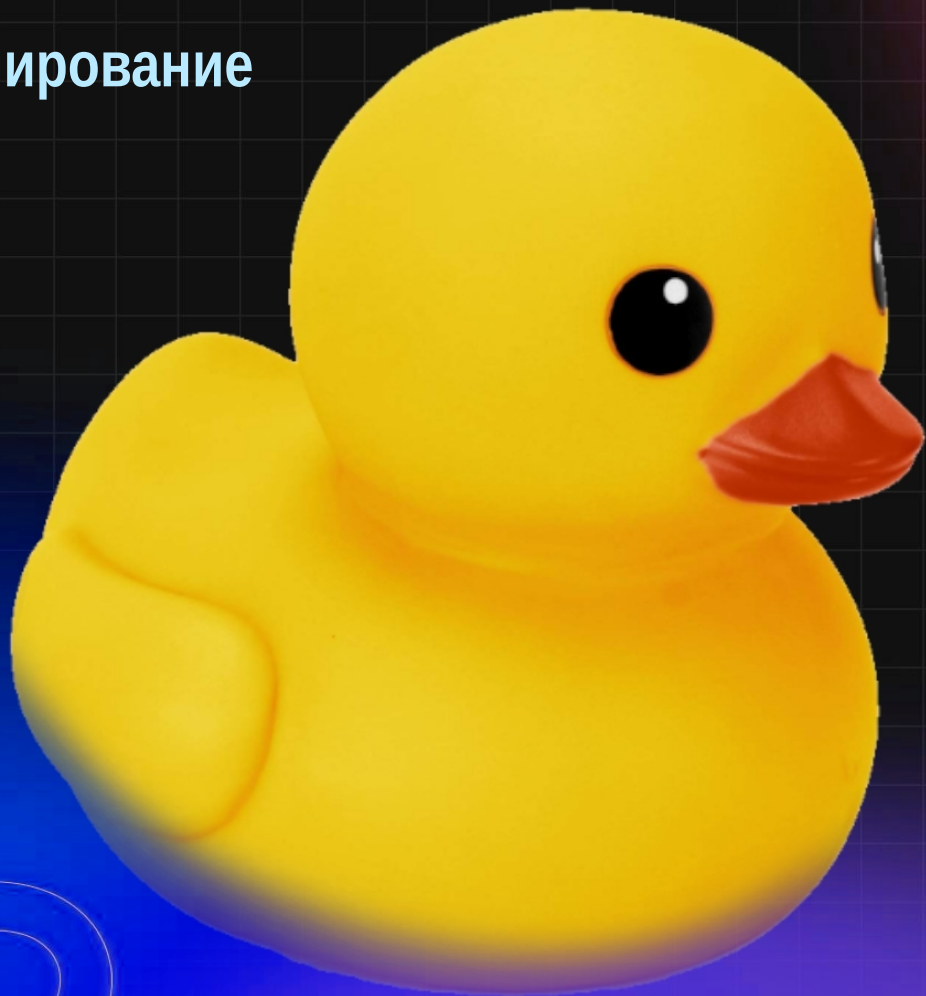
```
Builder b = new Builder()  
    .setWhere("Moscow")  
    .setTransport("train")  
    .setDate(1,7,2023);  
    ...  
Trip trip = b.createTrip();
```

```
Director d = new Director();  
Trip trip = d.makeTrainTrip("Moscow", new Date(1,7,2023));
```

- Изоляция сложной сборки от бизнес-логики
- Разрешает пошаговое создание
- Усложнение структуры программы (+ строитель)



Программирование
2 семестр
2024

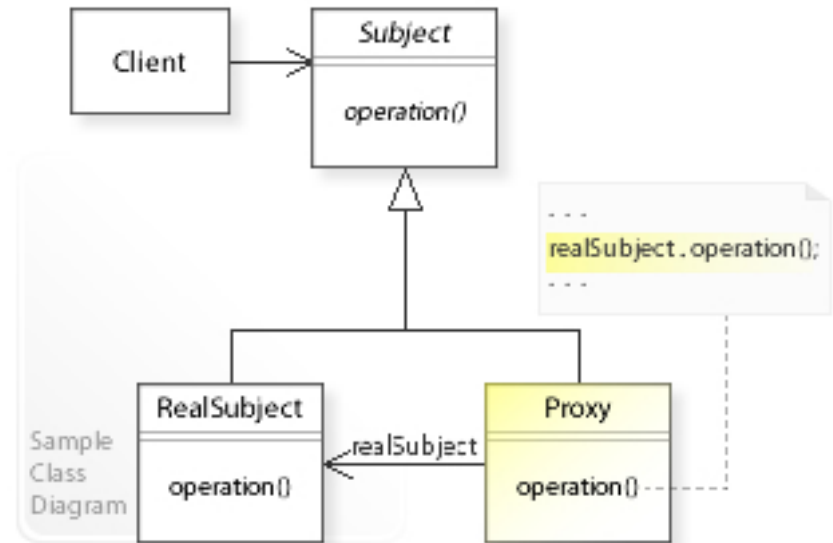


ІТМО

Структурные
шаблоны
(Structural)

- Adapter — Адаптер
- Bridge — Мост
- Composite - Компоновщик
- Decorator - Декоратор
- Facade - Фасад
- Flyweight - Легковес
- Проху - Заместитель

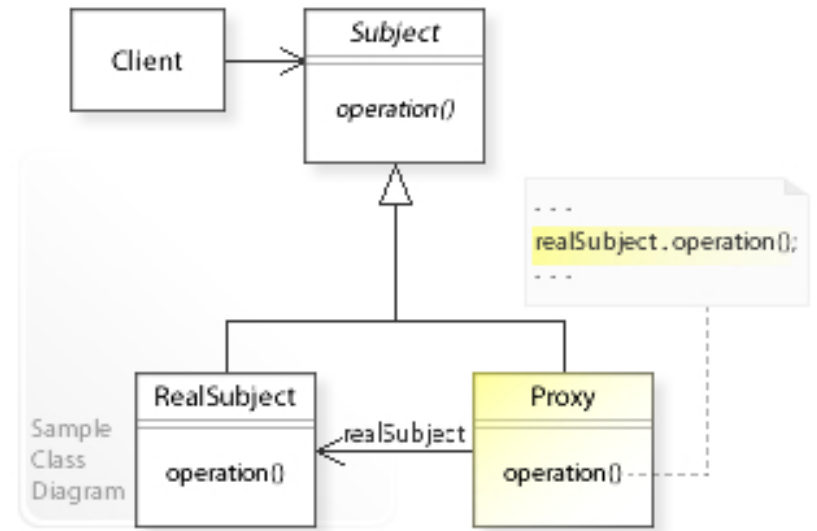
- Заместитель - делегирование **с тем же интерфейсом**
- Подмена реального объекта заместителем
 - ❖ Virtual - ленивое создание
 - ❖ Remote - удаленный объект
 - ❖ Security - контроль доступа
 - ❖ Caching - кэширование
 - ❖ Smart - логирование
- `java.lang.reflect.Proxy`



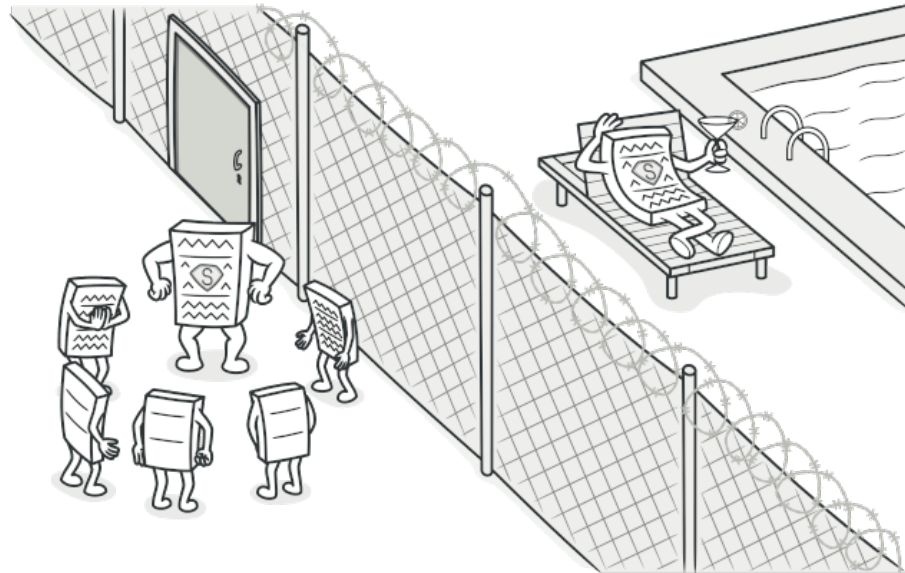
- Заместитель

```
class Proxy implements Subject {  
    RealSubject real;  
    public request() {  
        return real.request();  
    }  
}
```

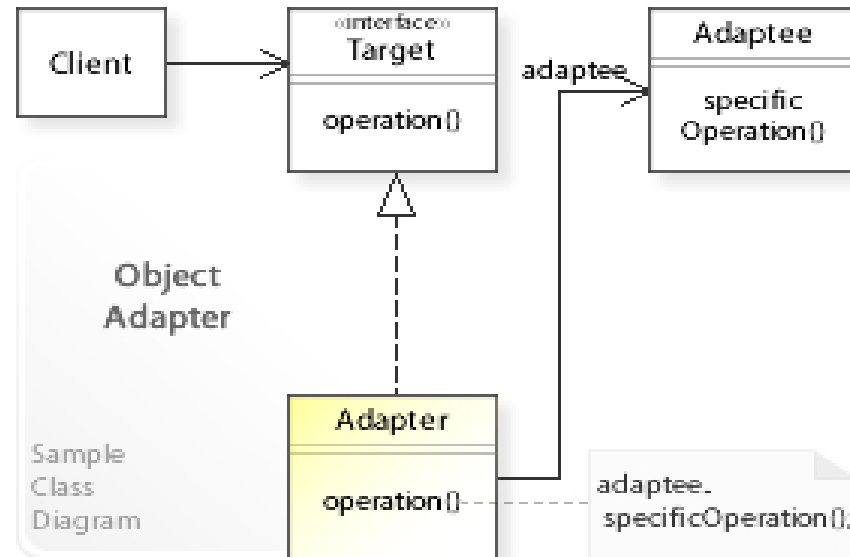
```
Subject subject = new Proxy();  
subject.request();
```



- Контроль работы с реальным объектом
- Клиент не знает, используется прокси или нет
- Замедление и усложнение доступа



- Адаптер - делегирование с заменой интерфейса
- Адаптирует один интерфейс к другому



- Адаптирует один интерфейс к другому

```
class EUPlug {  
    connect(EUSocket eus) {  
        eus.getEUPower();  
    }  
}
```



```
interface EUSocket {  
    getEUPower()  
}
```

- Адаптирует один интерфейс к другому

```
class EUPlug {  
    connect(EUSocket eus) {  
        eus.getEUPower();  
    }  
}
```



```
interface UKSocket {  
    getUKPower()  
}
```

```
interface EUSocket {  
    getEUPower()  
}
```

- Адаптирует один интерфейс к другому

```
class EUPlug {  
    connect(EUSocket eus) {  
        eus.getEUPower();  
    }  
}
```

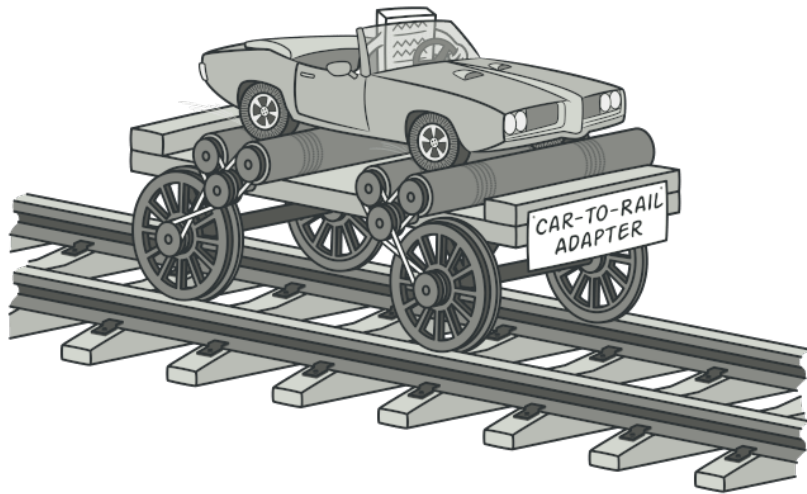


```
class Adapter  
    implements EUSocket {  
    UKSocket uks;  
    getEUPower() {  
        uks.getUKPower();  
    }  
}
```

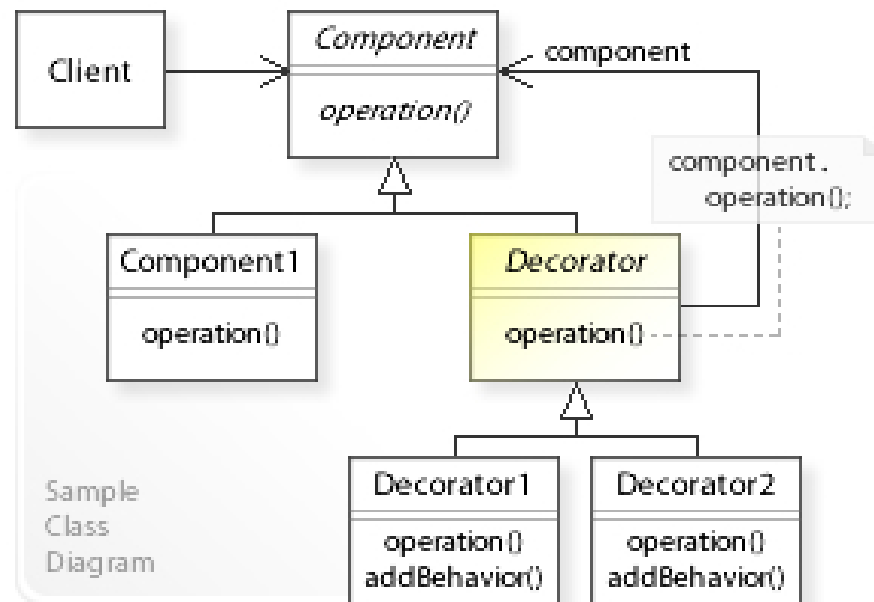
```
interface UKSocket {  
    getUKPower()  
}
```

```
interface EUSocket {  
    getEUPower()  
}
```

- Позволяет устранить несовместимость интерфейсов
- Не нужно переписывать существующий код
- Усложнение структуры (+ адаптер)

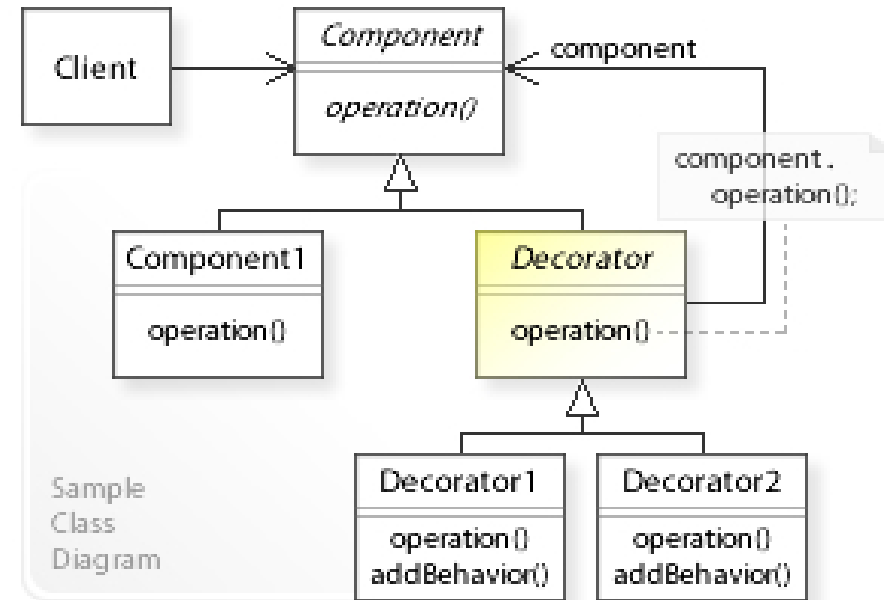


- Декоратор - делегирование с **новой функциональностью**
- Блины с добавками в Теремке
- InputStream
 - ❖ BufferedInputStream,
 - ❖ LineNumberInputStream,
 - ❖ ...

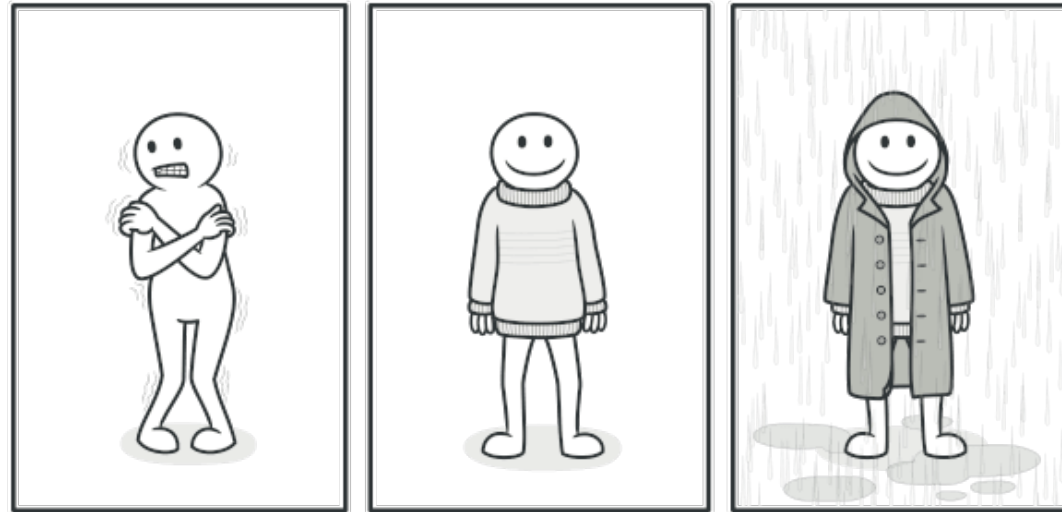


- Делегирование с расширением функциональности

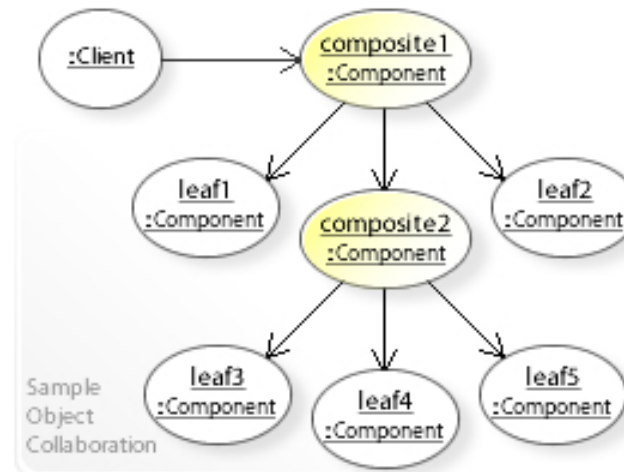
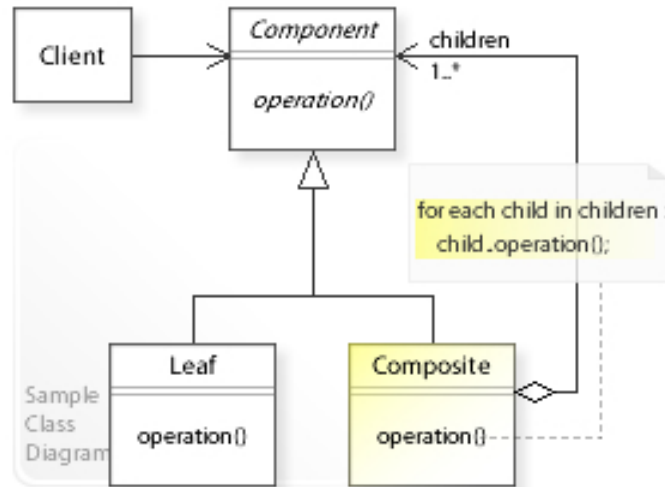
```
class Decorator implements Component {  
    Component1 component;  
    public operation() {  
        component.operation();  
        addCoolBehavior();  
    }  
}
```



- Позволяет добавлять функциональность динамически
- Вместо большой иерархии - несколько декораторов
- Сложность конфигурирования



- Компоновщик - иерархическая древовидная структура
- Абстрактный компонент и его потомки: компонент-лист и компонент-контейнер

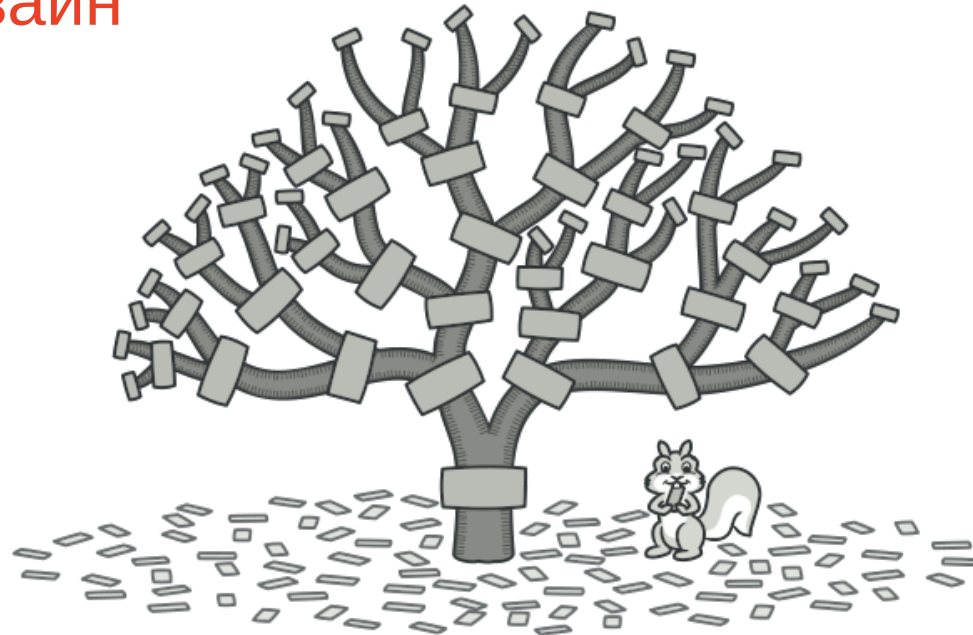


- Компоновщик - иерархическая древовидная структура
- Абстрактный компонент и его потомки: компонент-лист и компонент-контейнер

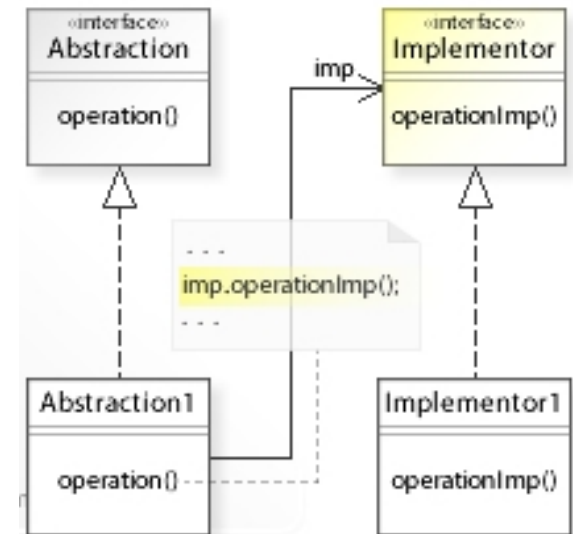
```
class Composite extends Component {  
    List<Component> children;  
    public operation() {  
        for (var c : children) {  
            c.operation();  
        }  
    }  
}
```

- Дополнительные операции
 - ❖ `addChild(Component c)`
 - ❖ `removeChild(Component c)`
 - ❖ `Component[] getChildren()`

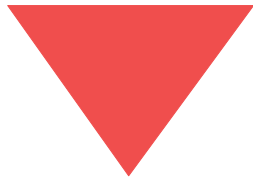
- Простая работа с деревом компонентов
- Простое добавление новых компонентов
- Слишком общий дизайн



- Мост - разделяет иерархии абстракций и реализаций
 - ❖ Абстракции - фигуры (треугольник, прямоугольник)
 - ❖ Реализации - стиль углов (острые, закругленные)
- Поведение делегируется реализации
- Позволяет менять обе иерархии




```
class Triangle  
class Rectangle
```

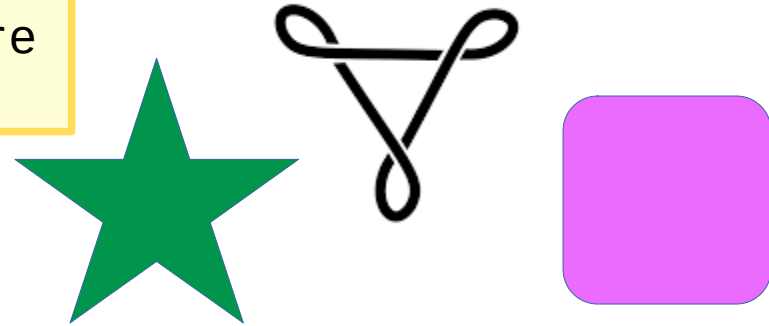


```
class SharpTriangle  
class RoundedTriangle  
class SharpRectangle  
class RoundedRectangle
```

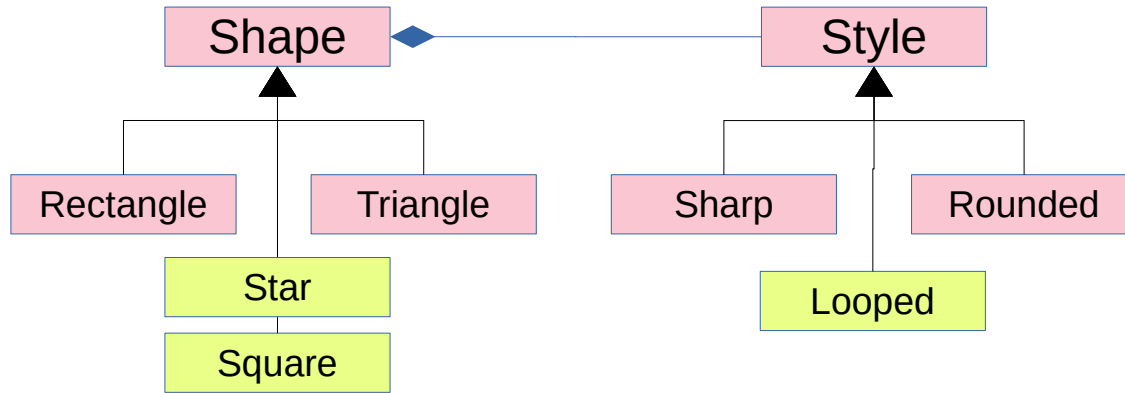


```
class SharpTriangle  
class RoundedTriangle  
class SharpRectangle  
class RoundedRectangle
```

```
+ Looped  
+ Square  
+ Star
```



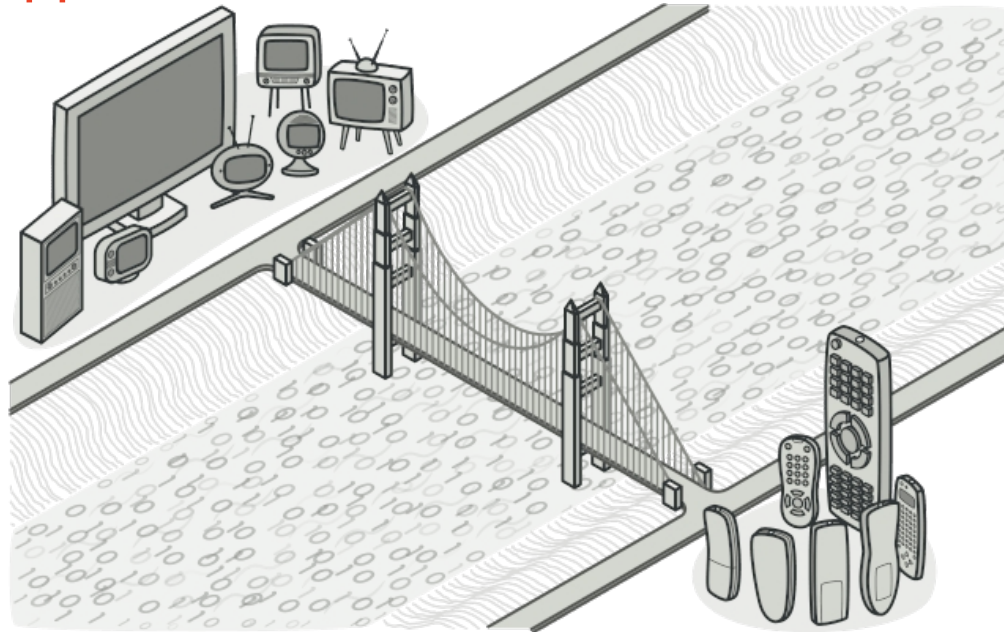
```
+  
class LoopedTriangle  
class RoundedSquare  
class SharpStar  
...
```



```
abstract class Shape {
    Style style;
    setStyle(Style style) {
        this.style = style;
    }
    abstract draw() {
        ...
        style.drawCorner();
        ...
    }
}

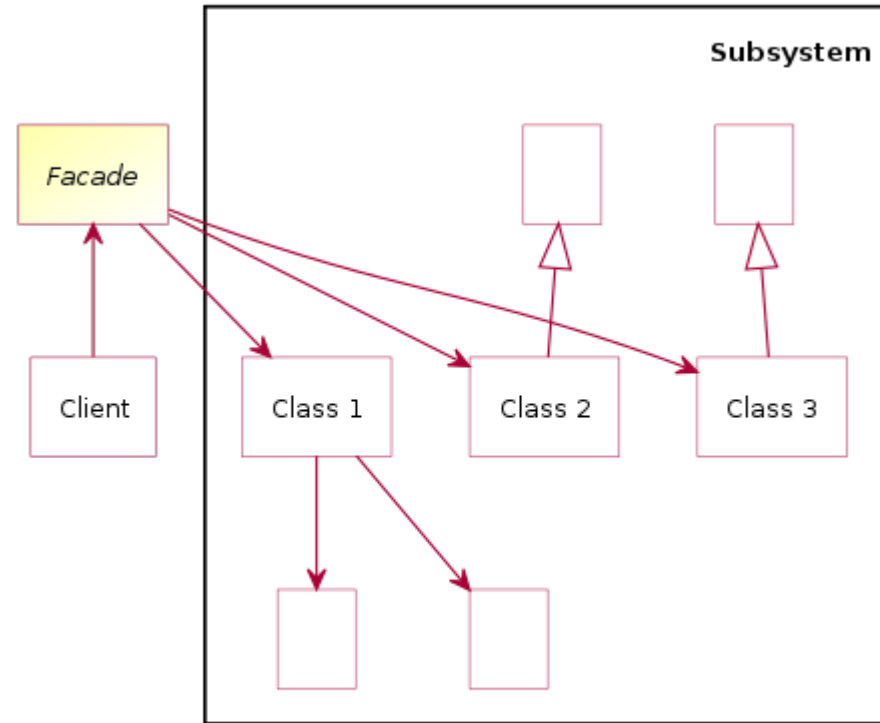
abstract class Style {
    drawCorner();
}
```

- Повышение расширяемости
- Устраняет сложность поддержки нескольких иерархий
- Появление дополнительных классов

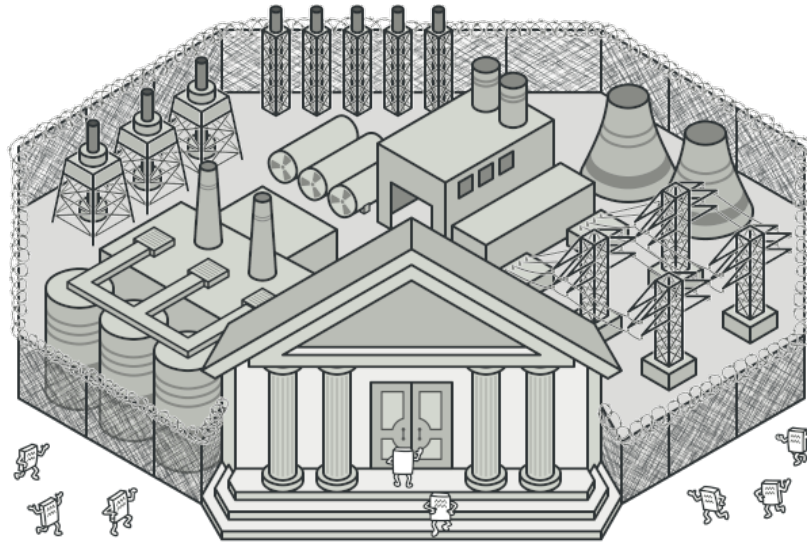


- Фасад - простой интерфейс к сложным системам
- Система может меняться
- Доступ через фасад

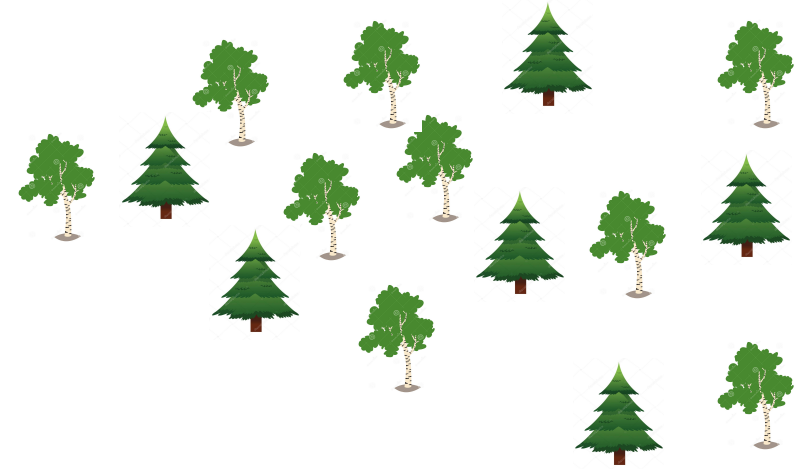
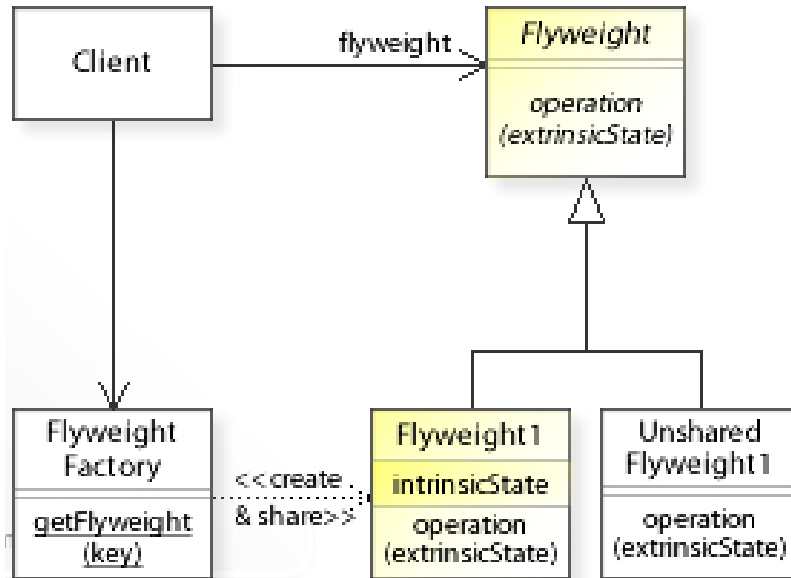
```
class Facade {  
    public go() {  
        ... // очень сложный код  
    }  
}
```



- Снижение зависимости клиента и компонентов системы
- Упрощает внешнее управление сложным объектом
- Есть опасность создать "Божественный объект"



- Легковес - экономия памяти при большом числе объектов



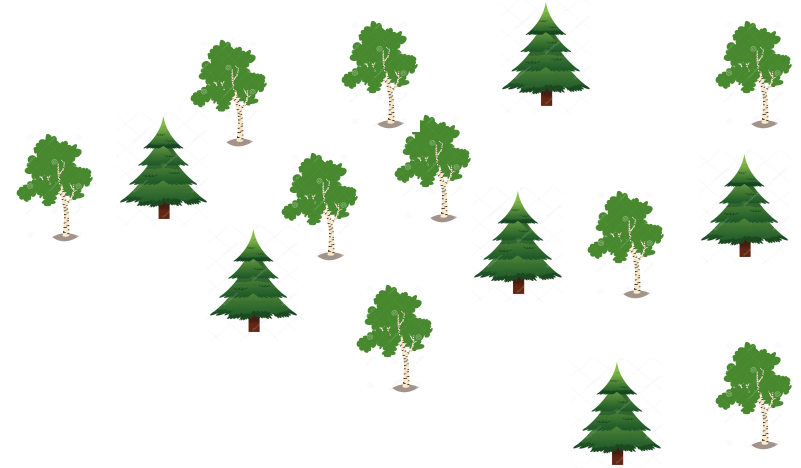
- Каждый объект - свое состояние



```
class Tree {  
    int x, y;  
    Color rgb;  
    Image image;  
}  
new Tree(10, 20, Color.GREEN, 'birch.png');  
new Tree(20, 30, Color.GREEN, 'pine.png');
```

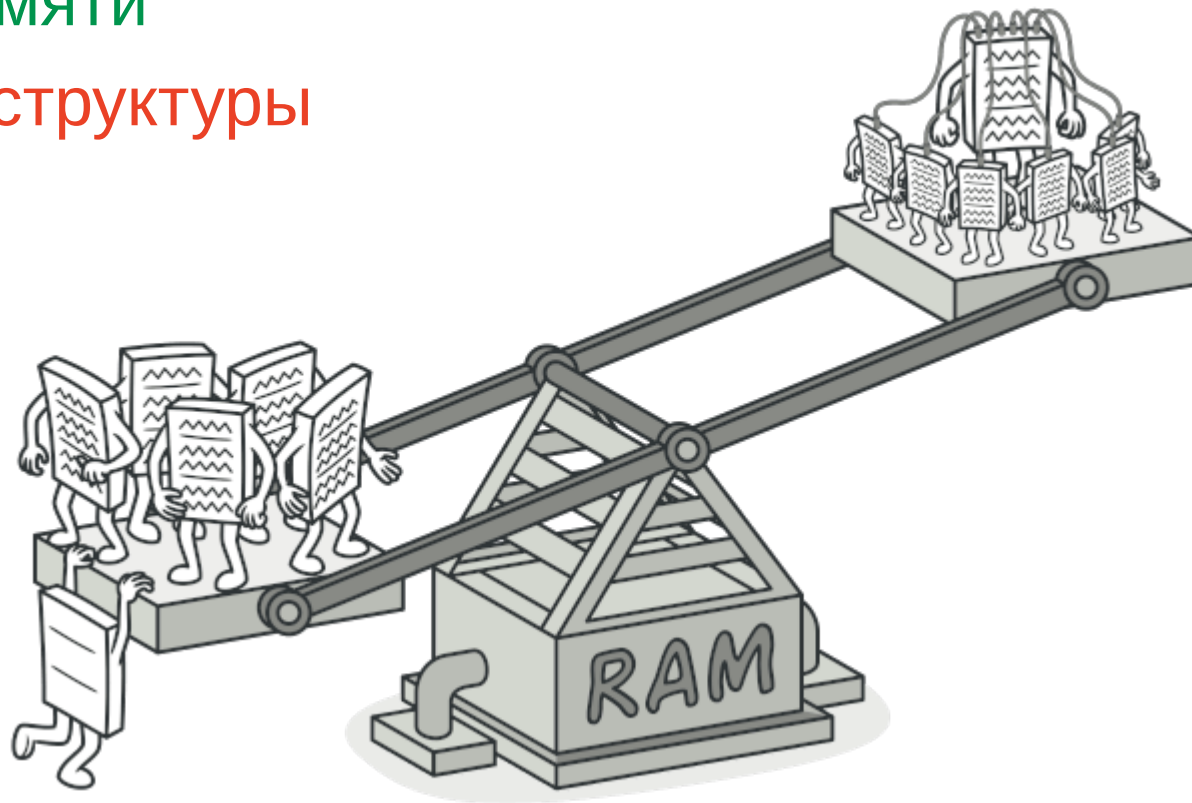
- Общее состояние храним вместе без дублирования
- Уникальное состояние - храним отдельно для каждого

```
class Tree {  
    static List<Coordinate> positions;  
    Color rgb;  
    Image image;  
}
```

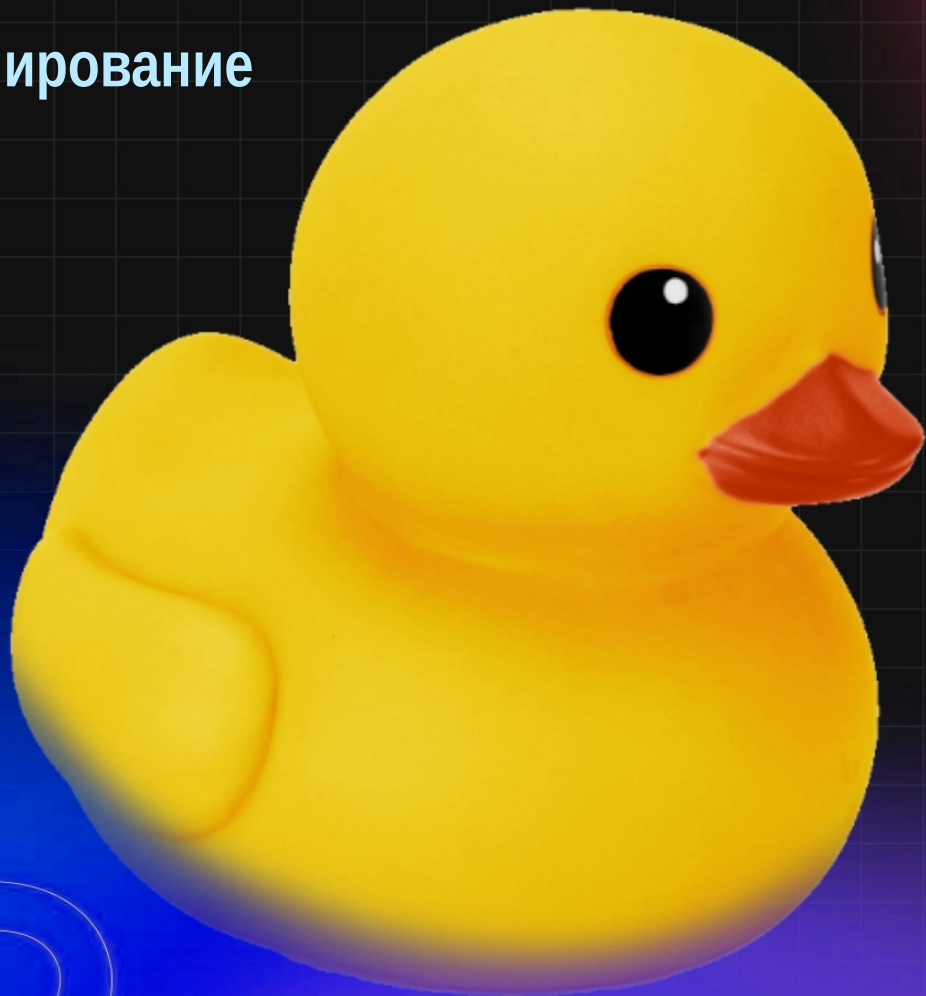


```
Tree birch = new Tree(Color.GREEN, 'birch.png');  
Tree pine = new Tree(Color.GREEN, 'pine.png');  
birch.addPosition(15, 35); birch.addPosition(27, 31);  
pine.addPosition(20, 40); pine.addPosition(32, 14);
```

- Экономия памяти
- Усложнение структуры



Программирование
2 семестр
2024

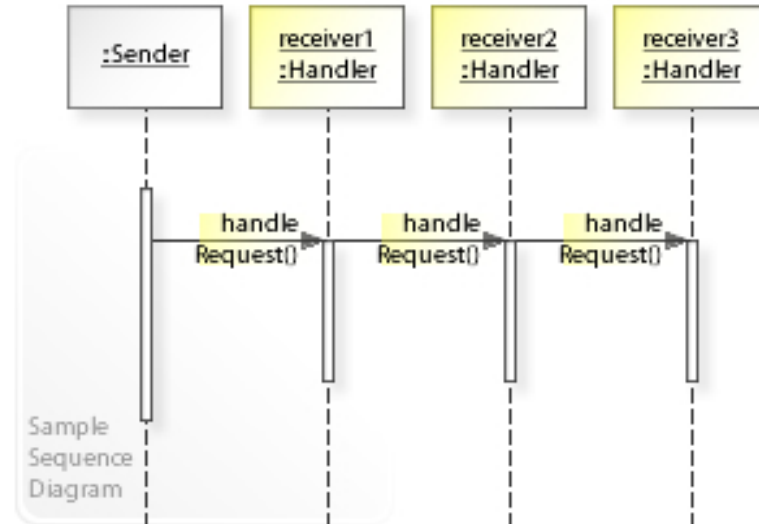
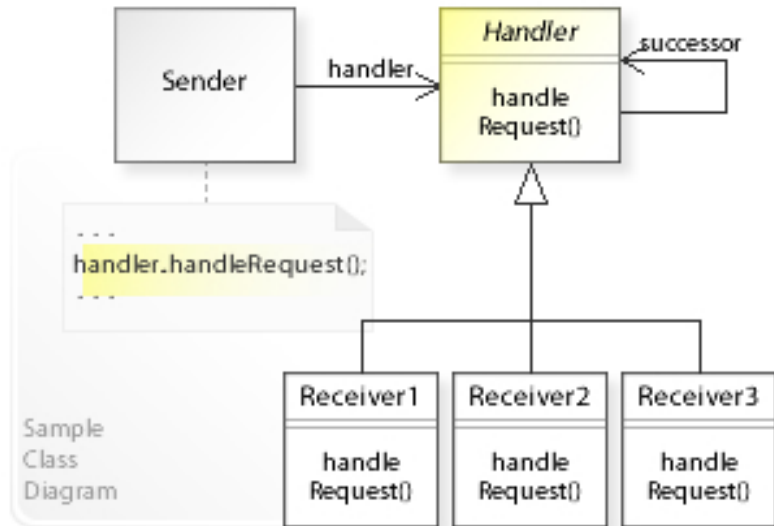


ІТМО

Поведенческие
шаблоны
(Behavioural)

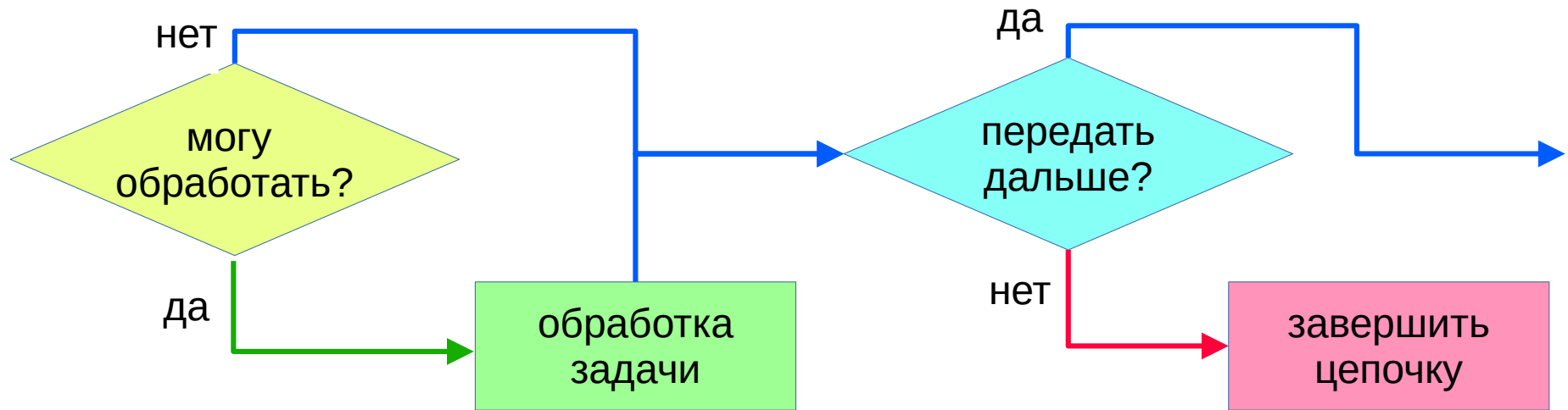
- Chain of Responsibility — Цепочка обязанностей
- Command - Команда
- Interpreter - Интерпретатор
- Iterator - Итератор
- Mediator - Посредник
- Memento - Хранитель
- Observer - Наблюдатель
- State - Состояние
- Strategy - Стратегия
- Template Method — Шаблонный метод
- Visitor - Посетитель

- Передача запроса по цепочке обработчиков

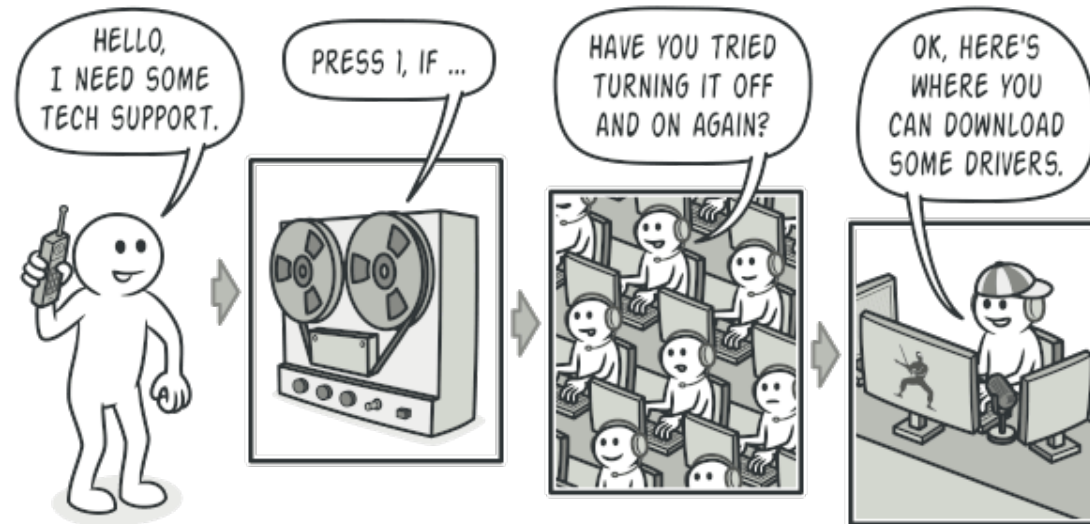


- Передача запроса по цепочке обработчиков
- В банкомате застряла карта
 - ❖ техподдержка 8-800-111-11-11
 - бот: ваш звонок очень важен для нас, хотите кредит — нажмите 1, застряла карта — нажмите 2
 - ❖ 2
 - другой бот: нажмите кнопку "отмена" на банкомате 4 раза
 - ❖ не помогает
 - оператор: стойте рядом, сейчас приедем

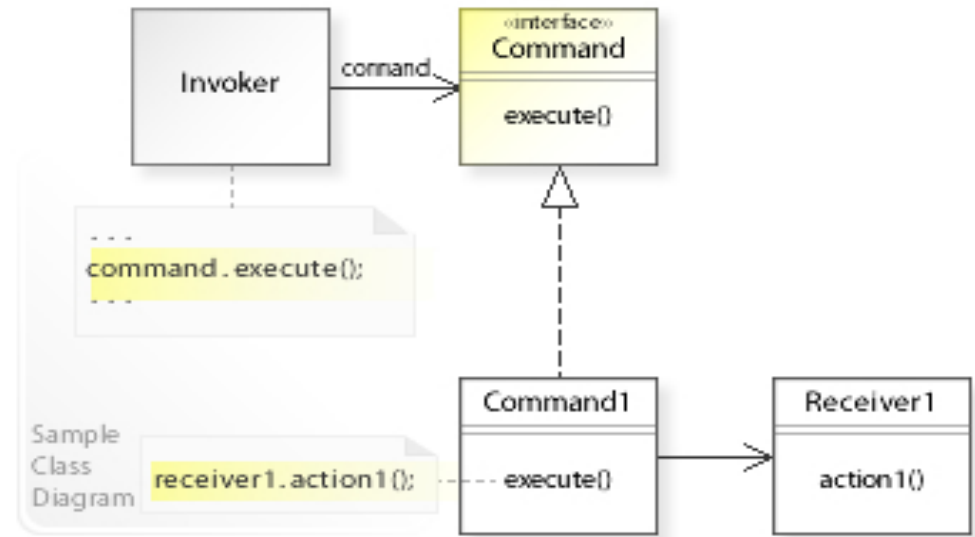
- Передача запроса по цепочке обработчиков
- Варианты действий



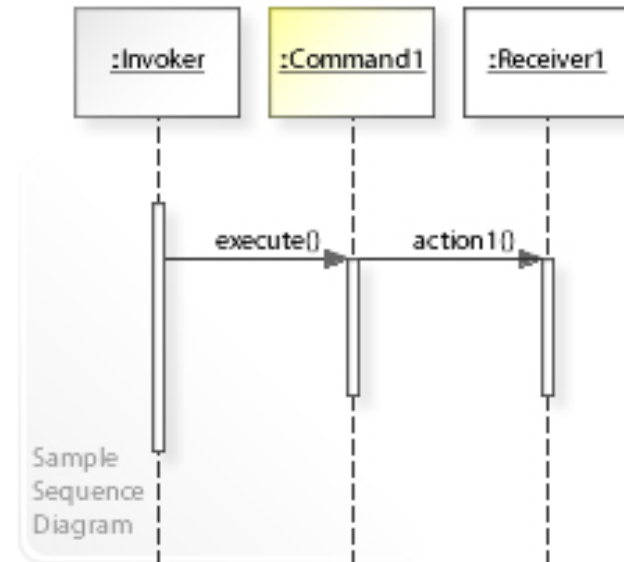
- Снижение зависимости между клиентом и обработчиками
- Запрос пересылается множеству обработчиков
- Запрос может остаться не обработанным



- Команда - разделение вызова и исполнения команд
 - ❖ Invoker - хранит ссылку на команду
 - ❖ Command - команда
 - ❖ Receiver - исполнитель
- Дополнительные возможности
 - ❖ Очередь команд
 - ❖ Макрокоманды
 - ❖ История команд



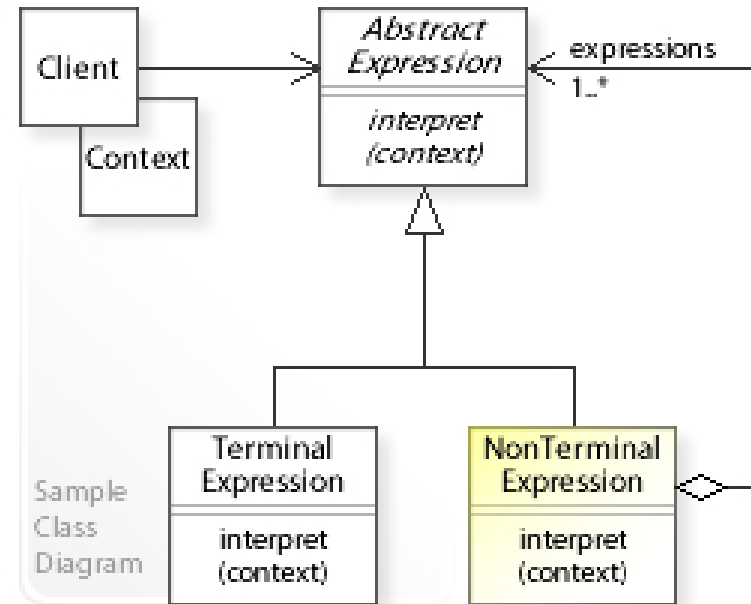
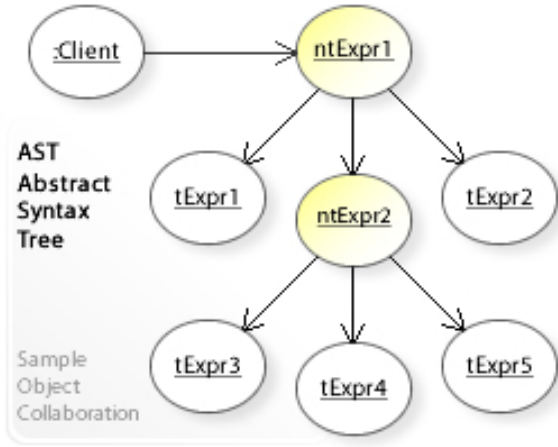
- Команда - разделение вызова и исполнения команд
- Отправитель выполняет команду
- Команда вызывает действие Исполнителя



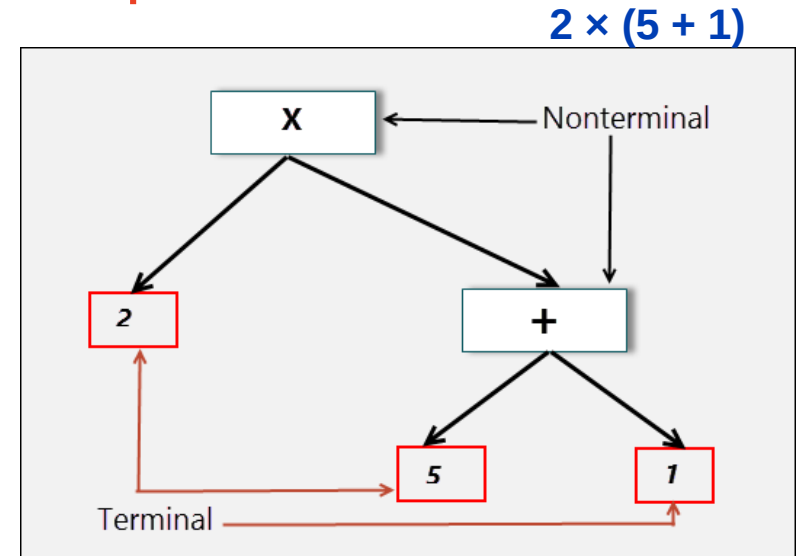
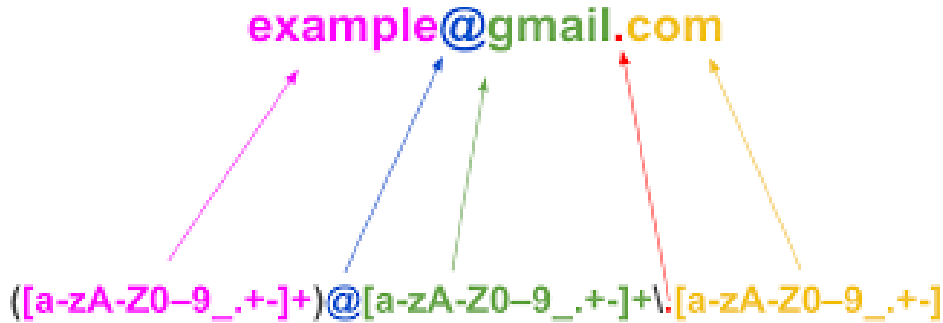
- Отделение кода исполнения команд от вызова
- Дополнительные возможности управления
- Усложнение кода (дополнительные классы)



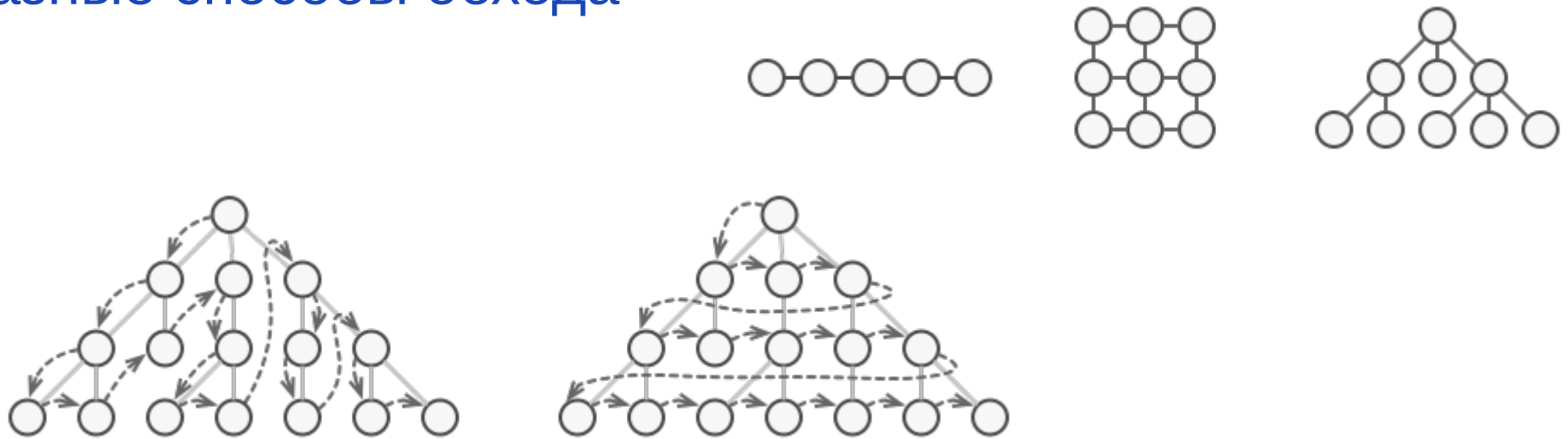
- Интерпретатор языка для управления поведением
 - ❖ Регулярные выражения
 - ❖ Форматирование строк



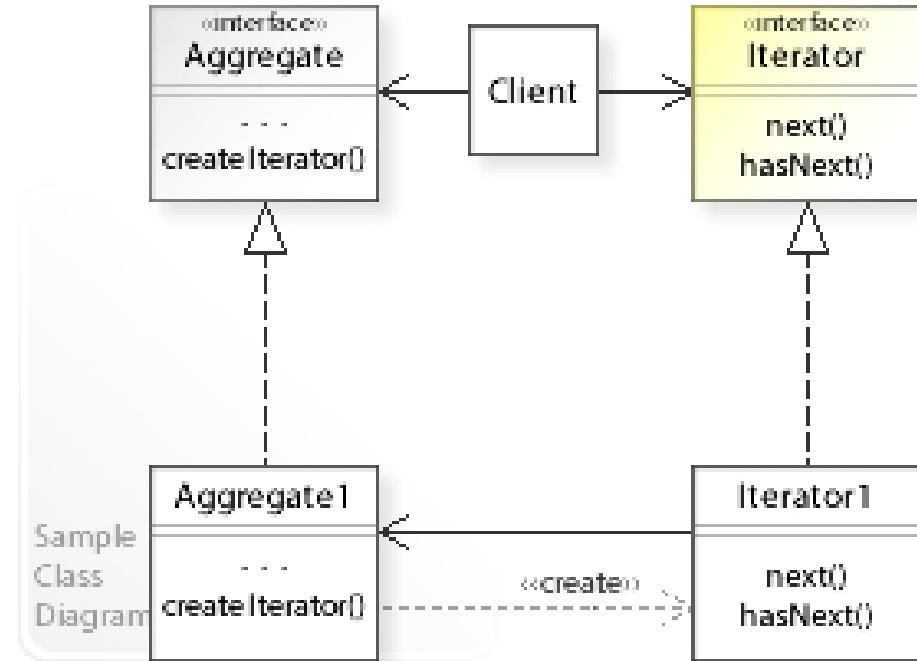
- Простота расширения и изменения языка
- Простота добавления новых способов
- Сложность сопровождения сложных грамматик



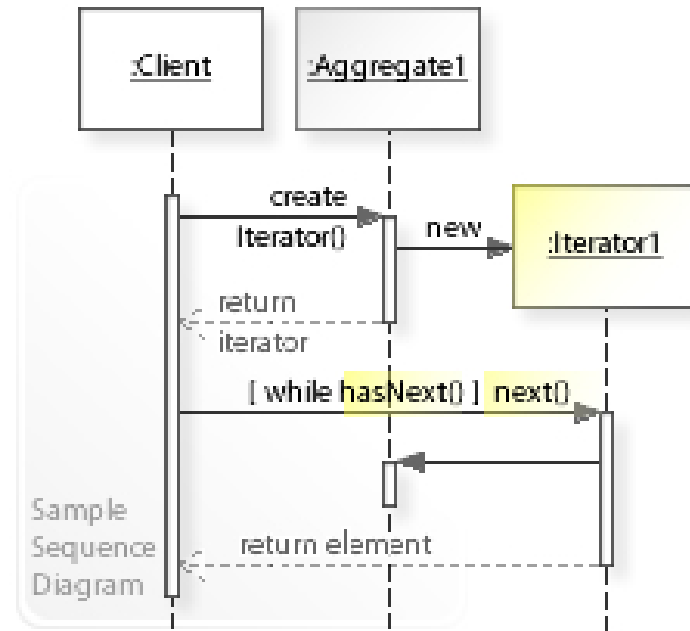
- Последовательный доступ к элементам коллекции
- Итератор знает внутреннюю структуру коллекции
- Разные способы обхода



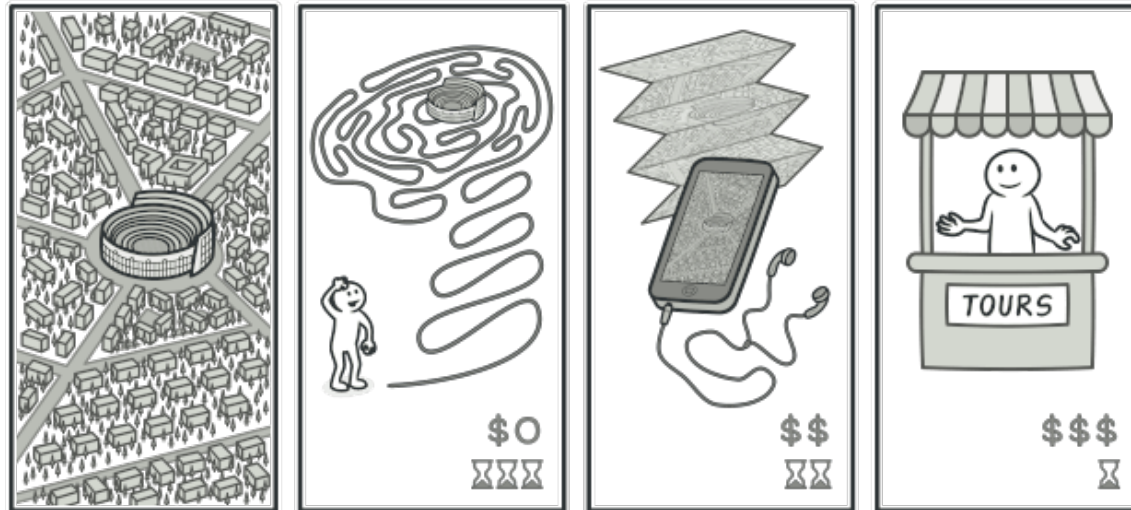
- Последовательный доступ к элементам коллекции
- Экскурсия
 - ❖ Реальный гид
 - ❖ Аудиогид
 - ❖ Путеводитель



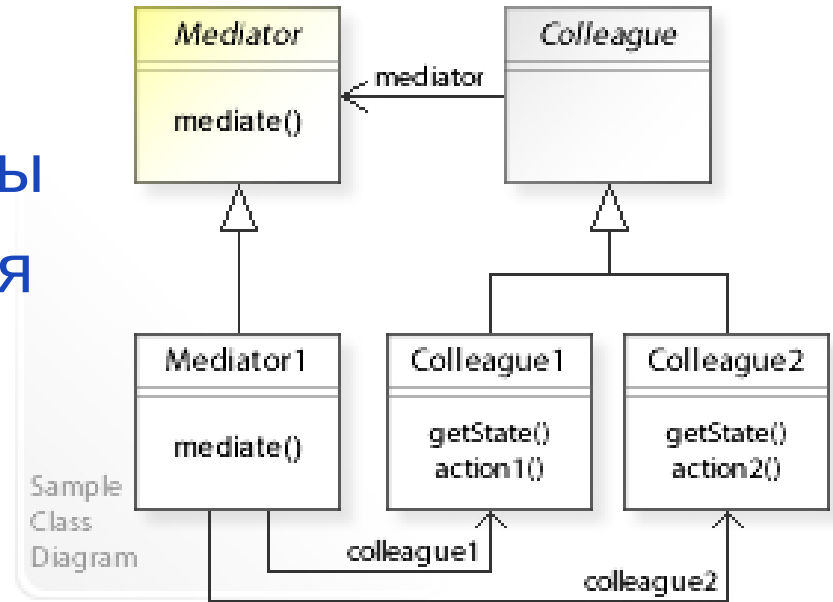
- Последовательный доступ к элементам коллекции
 - ❖ Клиент запрашивает итератор у структуры
 - ❖ Структура возвращает свой итератор клиенту
 - ❖ Клиент запрашивает элементы у итератора по очереди



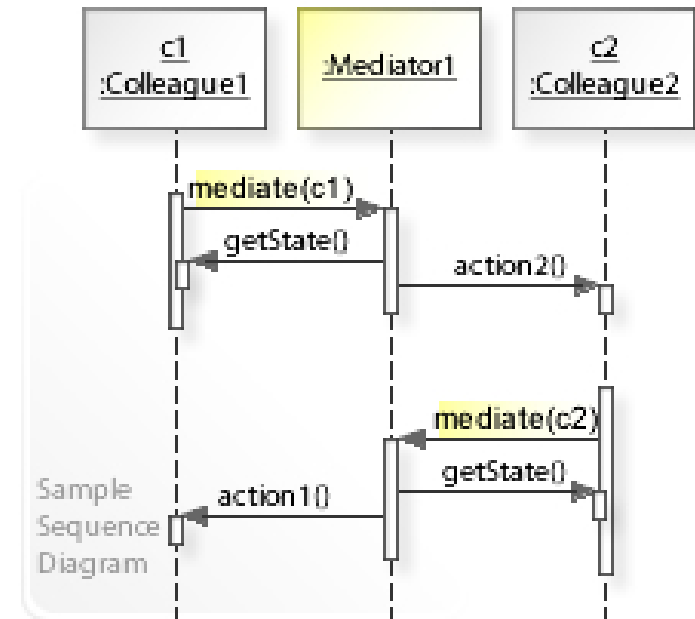
- Универсальный доступ ко всем коллекциям
- Гибкая реализация обхода структур
- Более сложный вариант, чем простой цикл



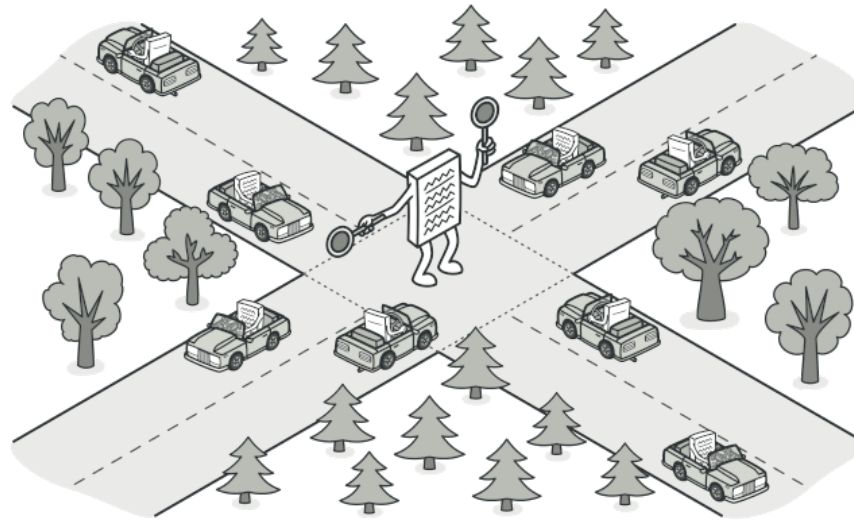
- Посредник
- Решает проблему сложного взаимодействия классов
- Уменьшает связанность системы
- Вместо прямого взаимодействия объектов друг с другом они общаются через посредника



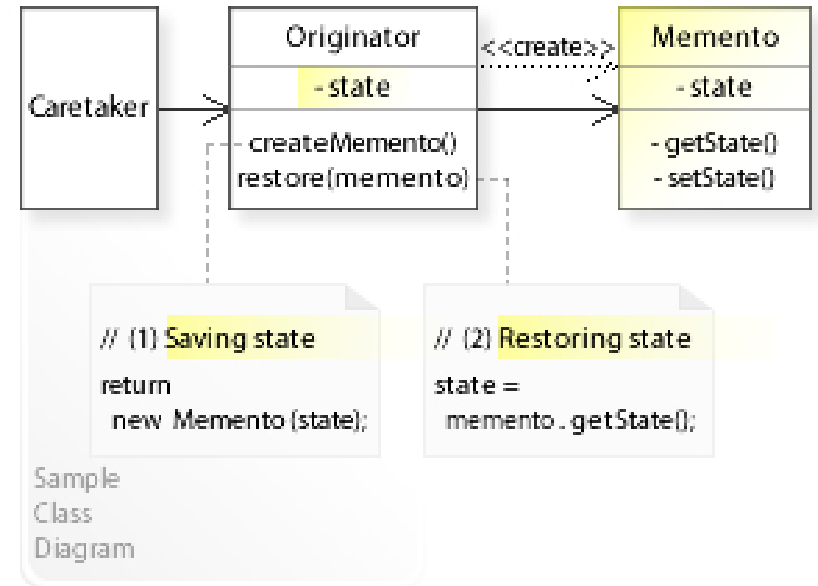
- Если нужно выполнить действие, Коллега обращается к Посреднику
- Посредник знает обо всех Коллегах и может получать их состояние
- Посредник отправляет Коллегам команды



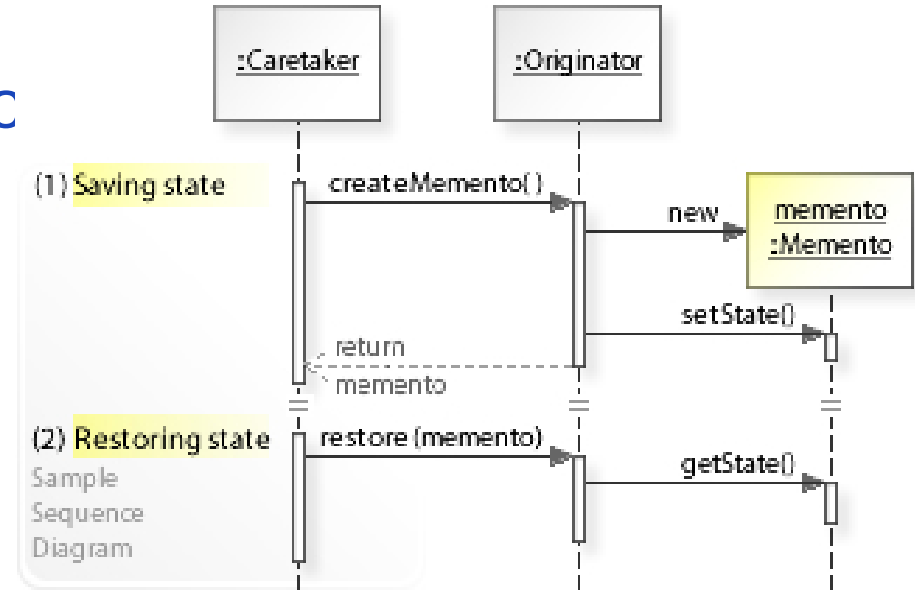
- Снижение зависимости между компонентами
- Простое централизованное управление
- Возможность разрастания посредника



- Хранитель
- Сохранение и восстановление состояния объекта
- Реализация отмены действий, резервных копий и транзакций
- Разделяет операции создания снимков и управления ими
- Позволяет управлять снимками не раскрывая состояние объект



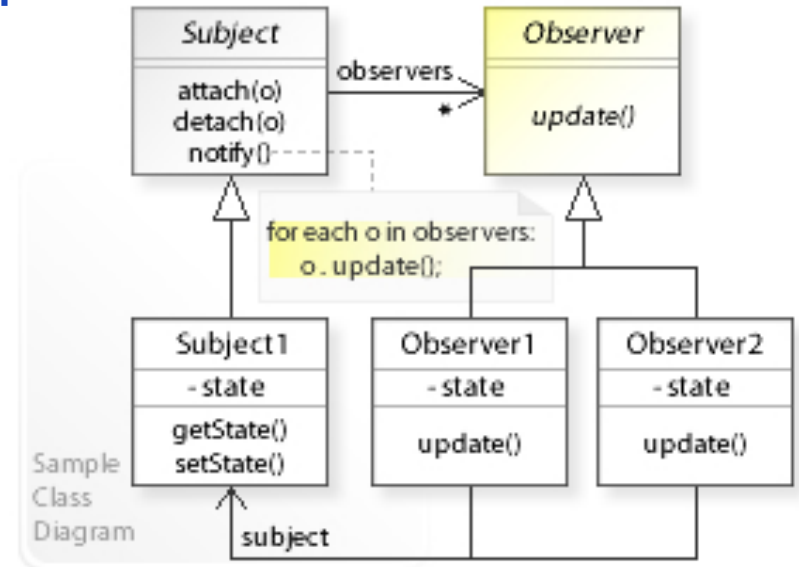
- Создатель (Originator) может сохранять и восстанавливать свое состояние в виде снимка (Memento)
- Опекун (Caretaker) умеет хранить снимки и выдавать их по запросу



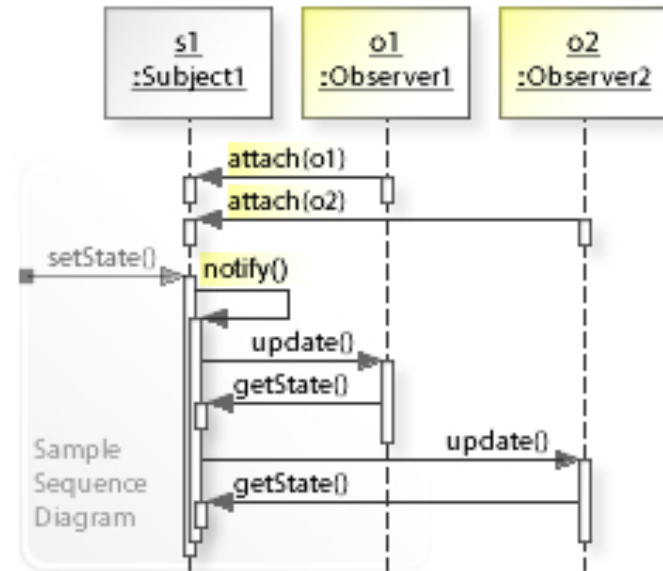
- Сохранение инкапсуляции исходного объекта
- Выделение отдельного хранилища снимков
- Может потребовать большого объема памяти



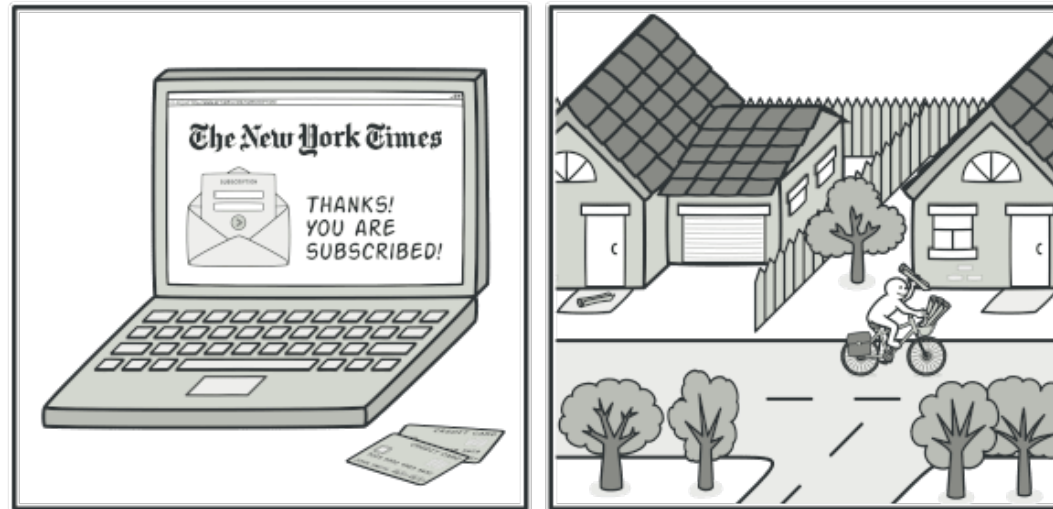
- Оповещение объектов об изменении состояния
- Подписка на извещения о событиях
- Асинхронная реакция на события
 - ❖ Действия пользователя в GUI
 - ❖ Обработка сигналов от датчиков
- Не нужны постоянные проверки
- Извещения получают только заинтересованные объекты



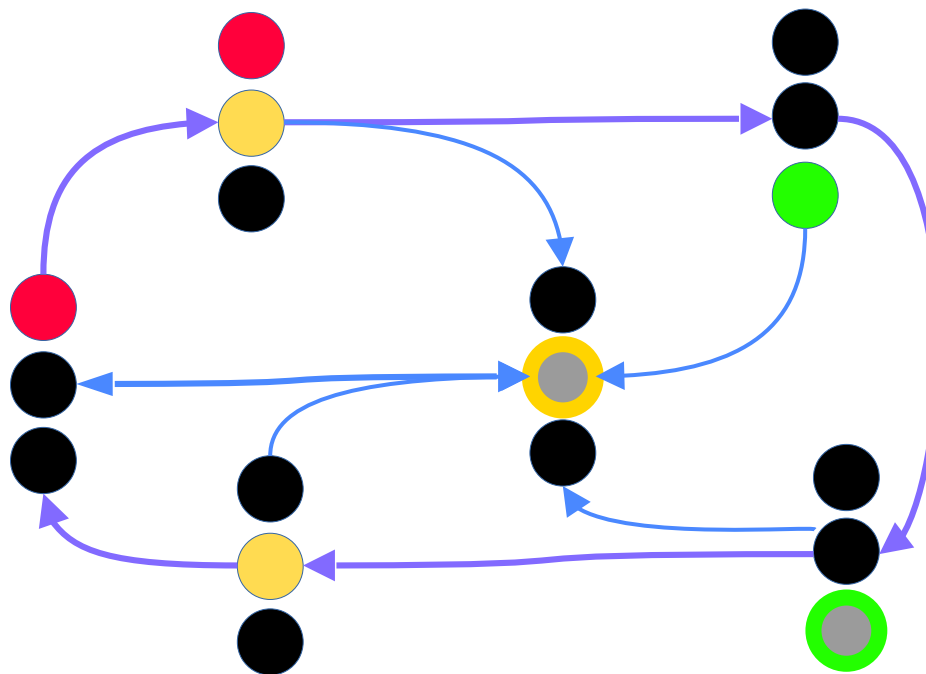
- Наблюдатель хочет реагировать на изменения в Издателе
- Наблюдатели подписываются на Издателя методом `attach(o)`
- При изменении состояния Издателя вызывает у каждого подписанного Наблюдателя метод `update()`
- Наблюдатель может получить состояние Издателя методом `getState()`



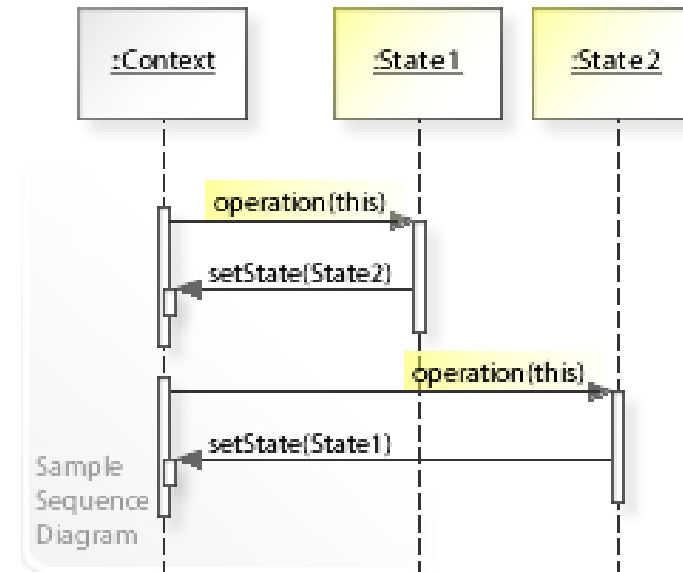
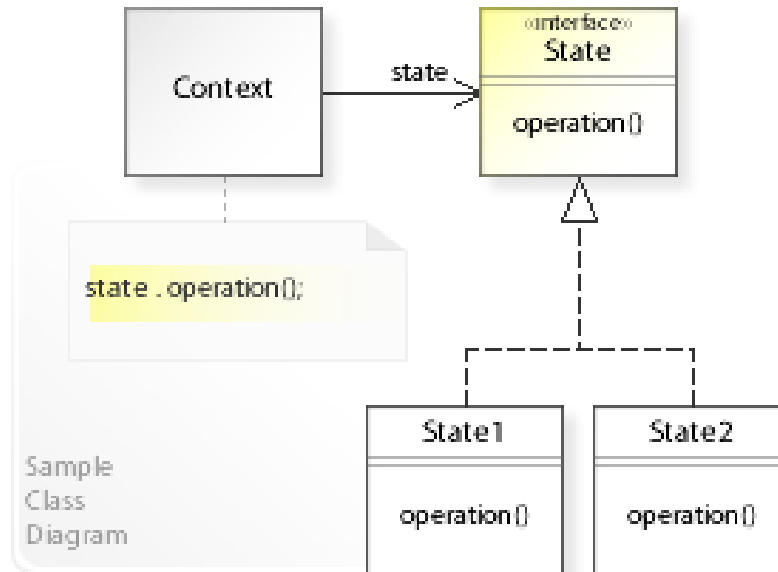
- Динамическая подписка и ее отмена
- Наблюдаемый объект не зависит от наблюдателей
- Сложность отладки



- Изменяет поведение объекта в зависимости от состояния — реализация конечного автомата



- Изменяет поведение объекта в зависимости от состояния — реализация конечного автомата



- Код без шаблона

```
if (state == YELLOW) {  
    yellowOn(10);  
    state = RED;  
} else if (state == RED) {  
    redOn(30);  
    state = RED_YELLOW;  
} else ...
```

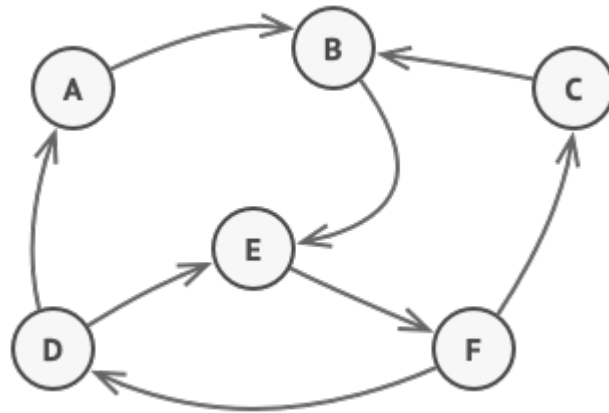
- Код с шаблоном

```
class YellowState
    implements State {
    State operation() {
        yellowOn(10);
        return new RedState();
    }
}
```

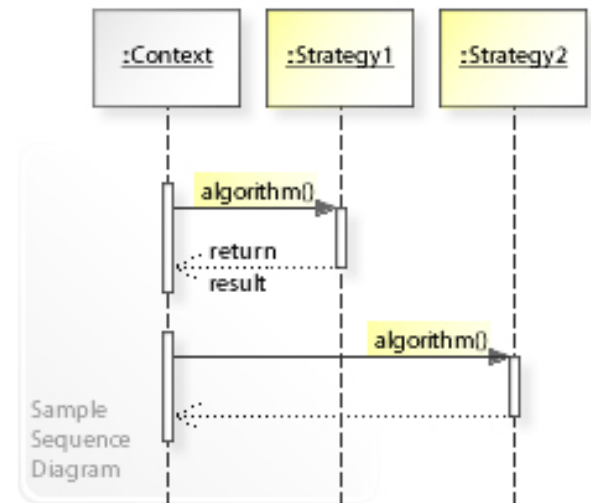
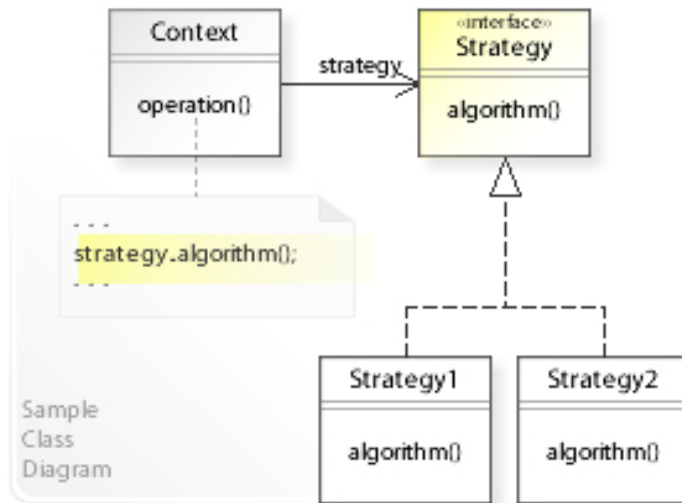
```
class RedState
    implements State {
    State operation() {
        redOn(30);
        return new RedYellowState();
    }
}
```

```
State state = new YellowState();
while (true) {
    state = state.operation();
}
```

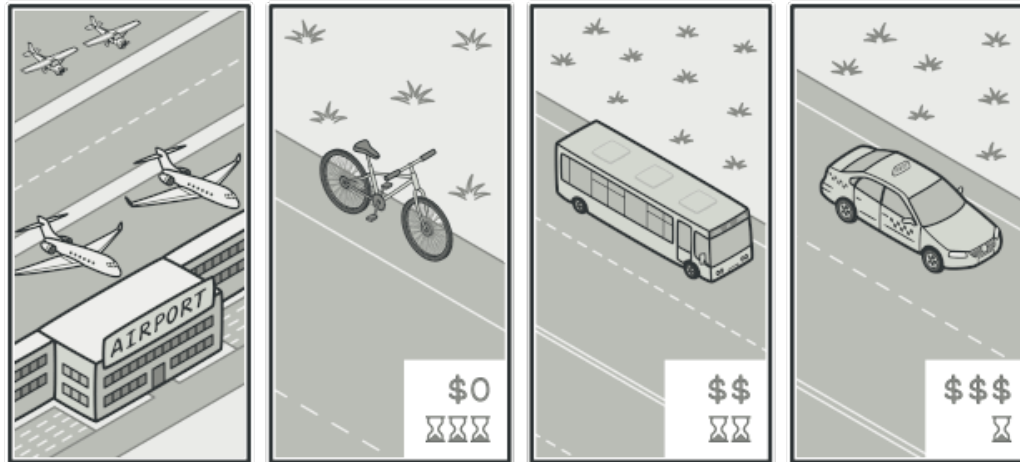
- Избавляет от множества операторов if
- Выделяет код для каждого состояния в одном месте
- В некоторых случаях слишком сложный код



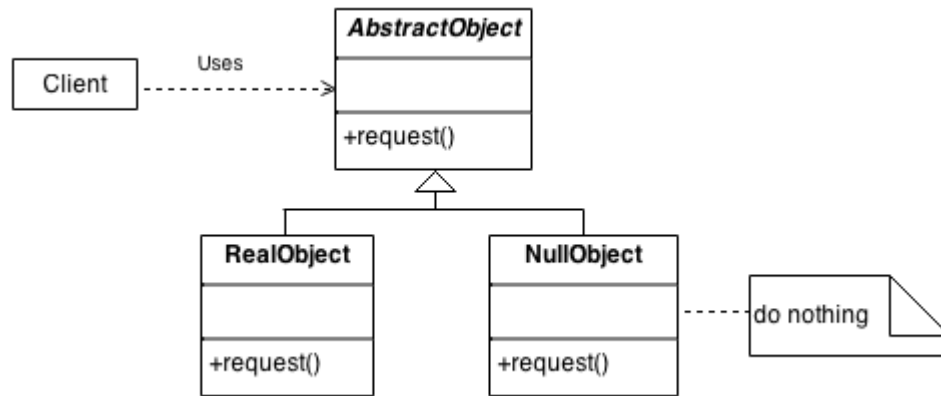
- Выбор одного из алгоритмов, реализованных в классе
 - ❖ Как добраться до аэропорта
 - Автобус
 - Такси
 - Пешком



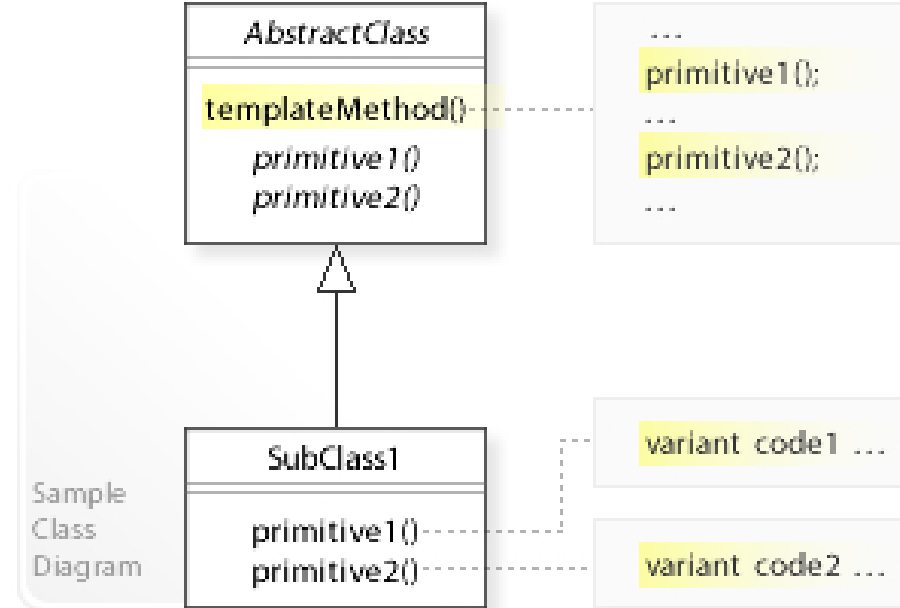
- Изолирует алгоритм в своем классе
- Динамический выбор стратегии
- Усложнение программы (дополнительные классы)



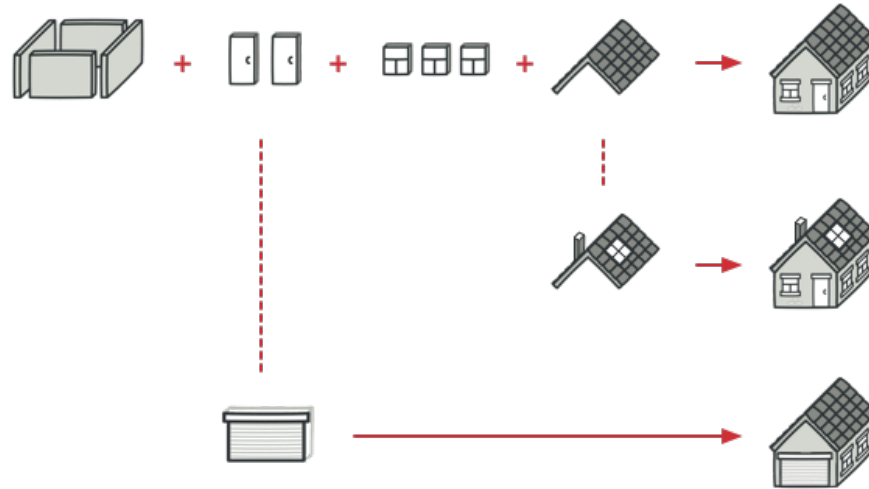
- Предоставляет объект с нулевым состоянием или поведением
- NullObject может замещать реальный объект



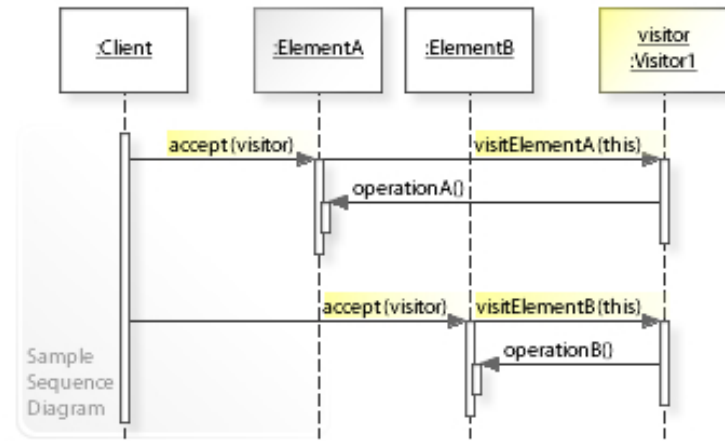
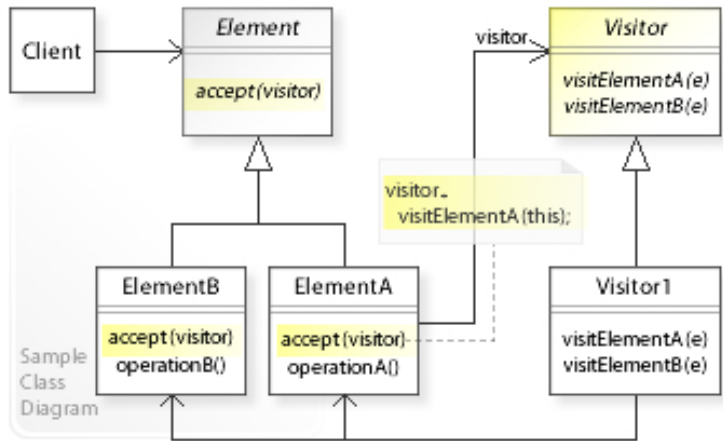
- Позволяет реализовать часть поведения в базовом классе, остальное реализуется в подклассах
- Шаги
 - ❖ абстрактные
 - ❖ дефолтные
 - ❖ хуки (изначально пустые)
- Покемоны
 - ❖ `Move.applySelfDamage()`



- Упрощает повторное использование кода
- Жестко задает последовательность действий
- При большом количестве действий сложно поддерживать

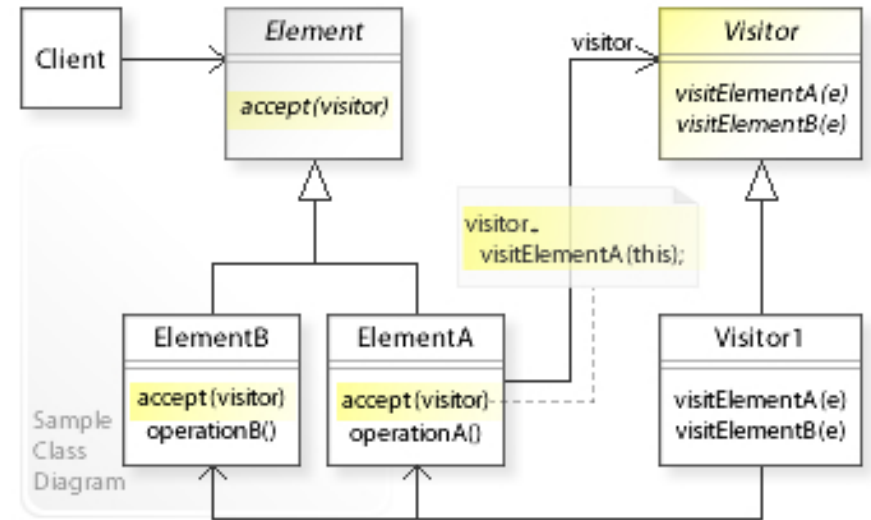


- Позволяет сгруппировать операции, выполняемые над структурой объектов
- Применяется если структура элементов более стабильна, чем набор операций



- Посетитель (Visitor) - общий интерфейс с методами `visit(ElementA e)`, `visit(ElementB e)`
- Конкретный посетитель (Visitor1) реализует поведение для каждого типа элемента
- Элемент (Element) - интерфейс элемента с методом `accept`
- В конкретных элементах:

```
accept(Visitor v) { v.visit(this); }
```



```
class Animal {
    void makeSound();
}

class Cat extends Animal {
    void makeSound() {
        say("мяу");
    }
}

Animal animal = new Cat();
animal.makeSound();
```

```
class Feeder {
    void feed(Animal a) {
        print("Feeding an animal");
    }
    void feed(Cat c) {
        print("Feeding a cat");
    }
}

void main() {
    Feeder feeder = new Feeder();
    Animal animal = new Cat();
    feeder.feed(animal);
}
```

Feeding an animal


```
class Animal {
    void accept(Feeder f) {
        f.feed(this);
    }
}

class Cat extends Animal {
    void accept(Feeder f) {
        f.feed(this);
    }
}
```

```
class Feeder {
    void feed(Animal a) {
        print("Feeding an animal");
    }
    void feed(Cat c) {
        print("Feeding a cat");
    }
}

void main() {
    Feeder feeder = new Feeder();
    Animal animal = new Cat();
    animal.accept(feeder);
}
```

Feeding a cat

- Упрощает добавление новых операций
- Отделяет код операций от структуры элементов
- Усложняет изменение структуры

