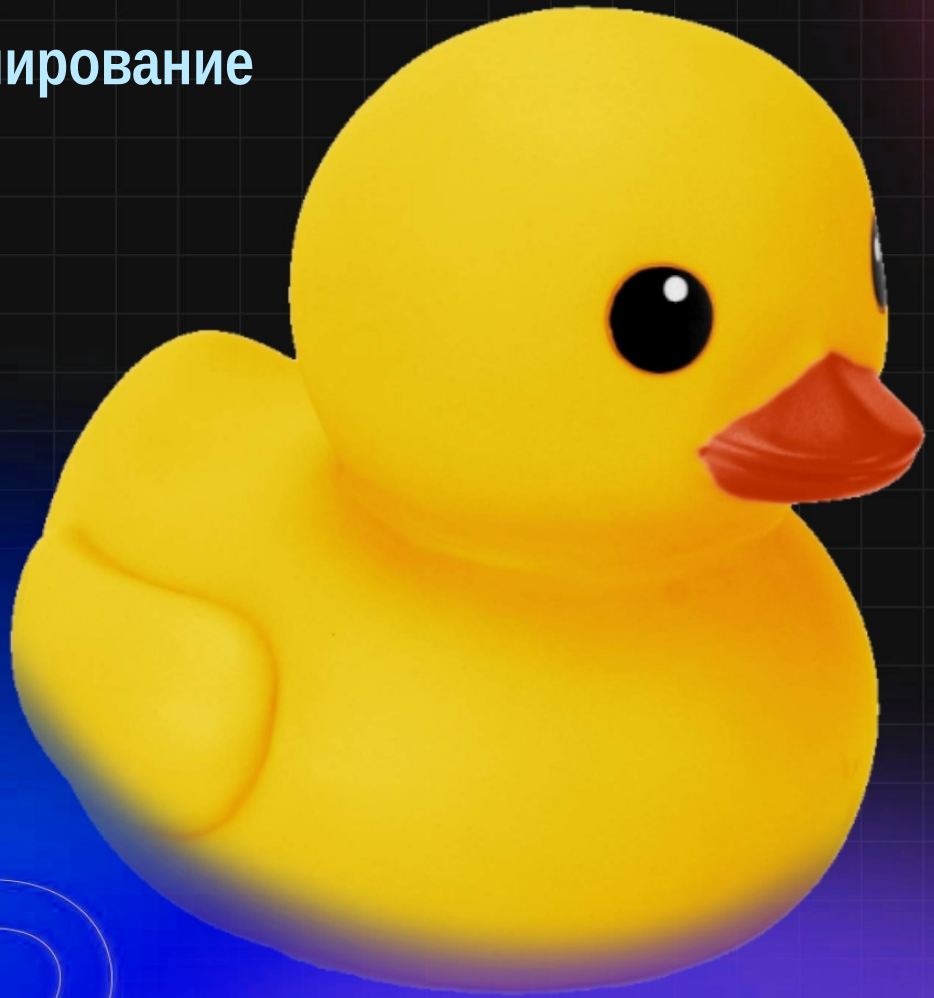
Программирование
2 семестр
2025



ИТМО

Шаблоны проектирования

Основные принципы проектирования

ИТМО

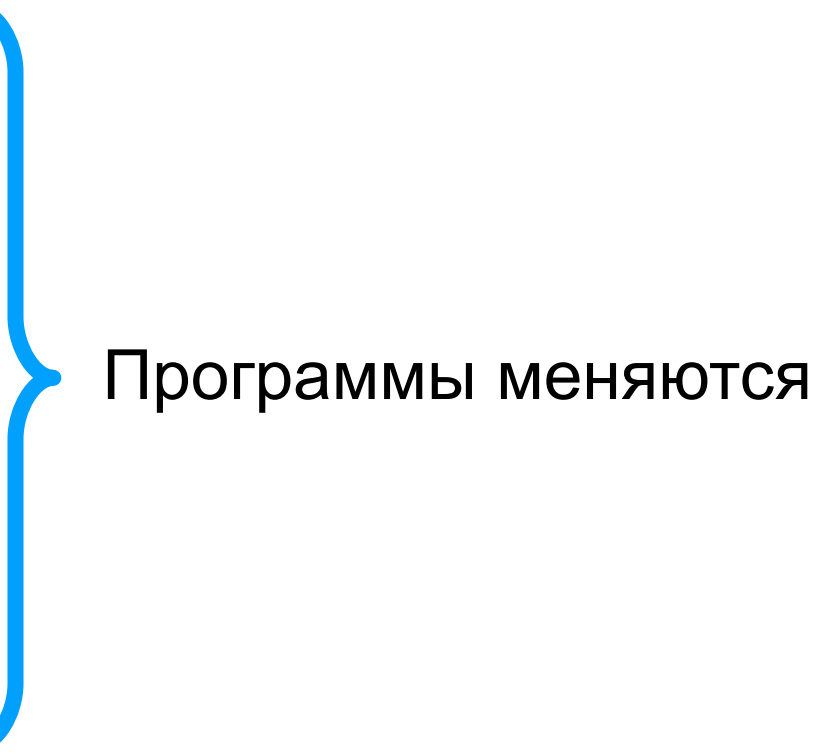
- Повторное использование кода
  - DRY - Don't repeat yourself
  - стандартные библиотеки
  - фреймворки
- Расширяемость
  - Учет возможных будущих изменений

Java

Изменения

ИТМО

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



Java

Шаблоны проектирования

ИТМО

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

GoF book

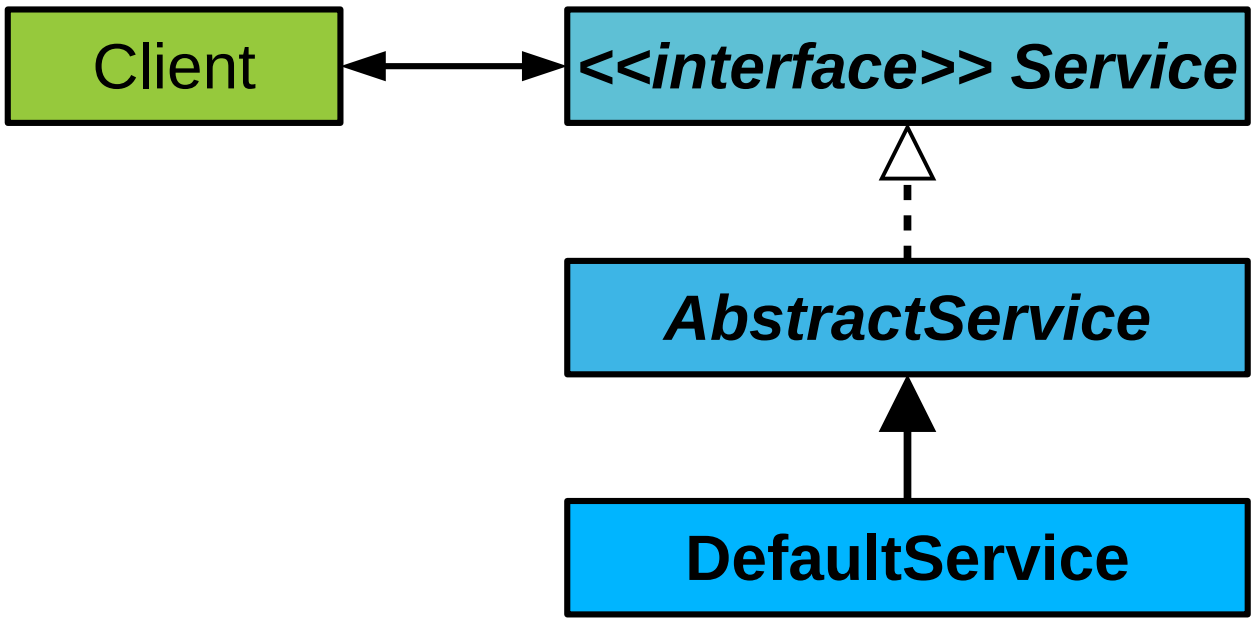
- **Шаблон - это не код!**
  Это принцип написания кода

Java

Базовые принципы - интерфейсы

ИТМО

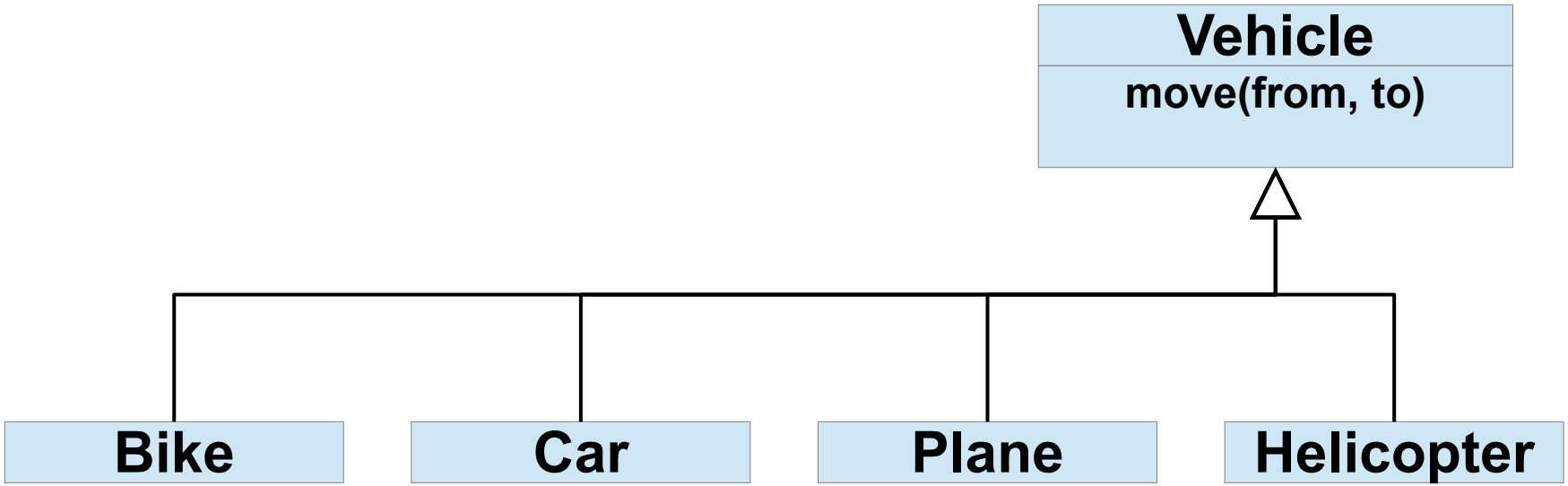
- Взаимодействие с абстракцией - ИНТЕРФЕЙС



Java

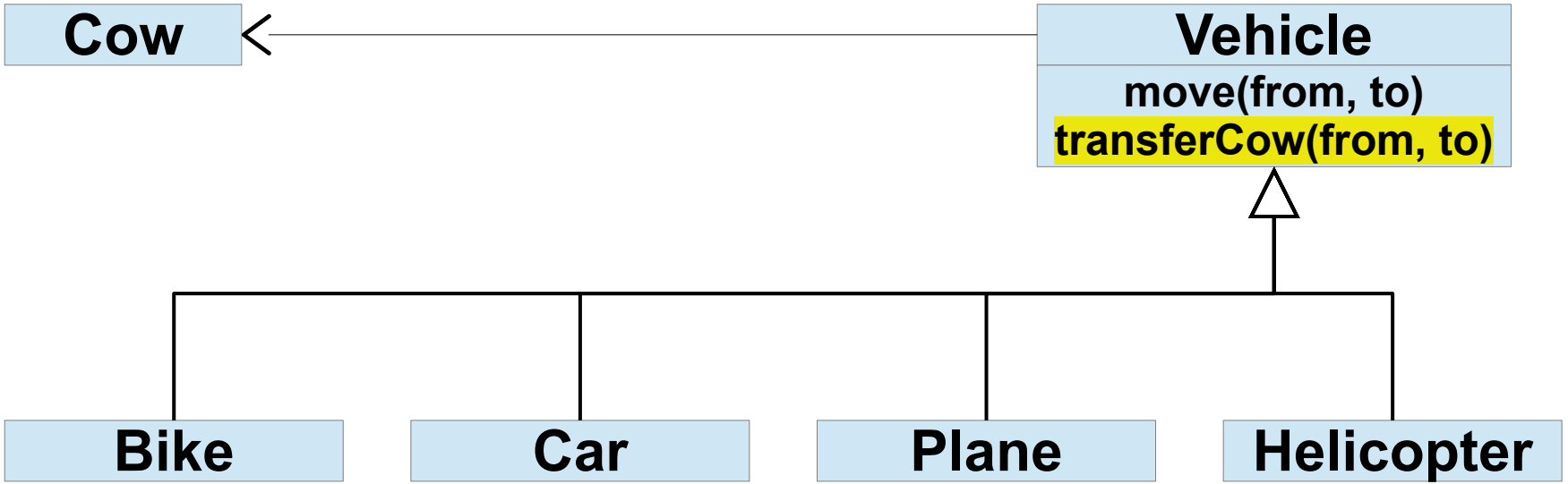
Наследование

- Умеет предок - умеют потомки
- Полиморфизм



Наследование

- Перевозка коровы - наследование??



Перевозка коровы

ИТМО

(С) х/ф «Мимино»

Java

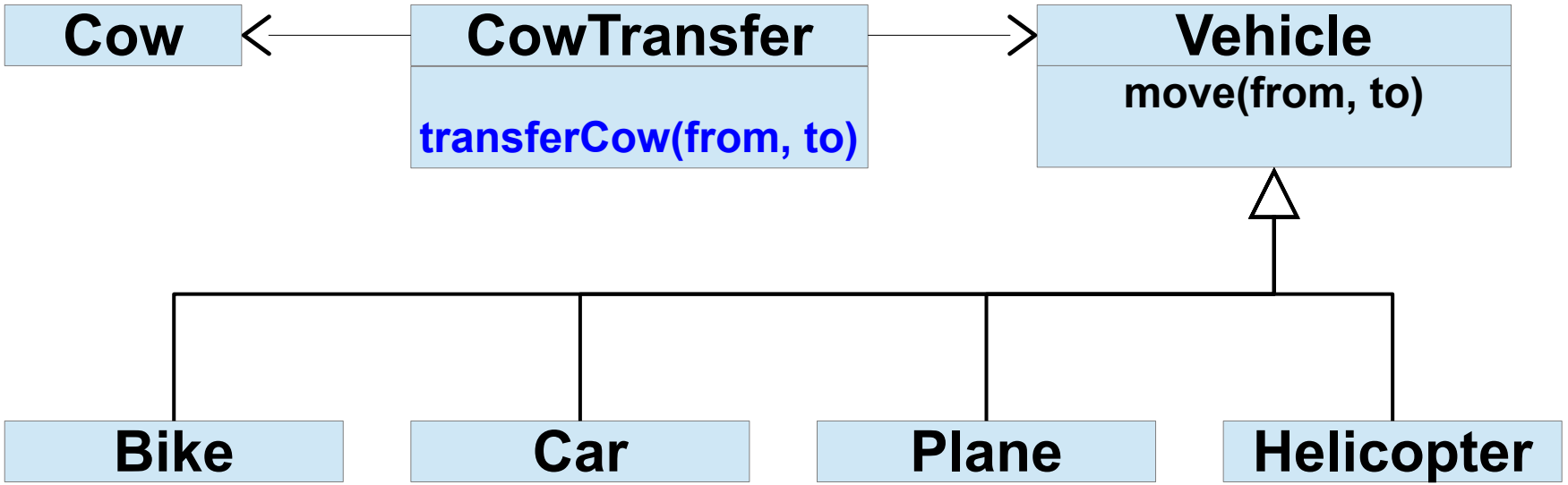
Перевозка коровы

(С) х/ф «Особенности национальной охоты»

Перевозка коровы

Наследование / делегирование

- Перевозка коровы - делегирование



Базовые принципы - делегирование

ИТМО

- Наследование
  - статическое отношение
  - классификация

```
class Animal {
  sleep() { ... }
}

class Cat extends Animal { }

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

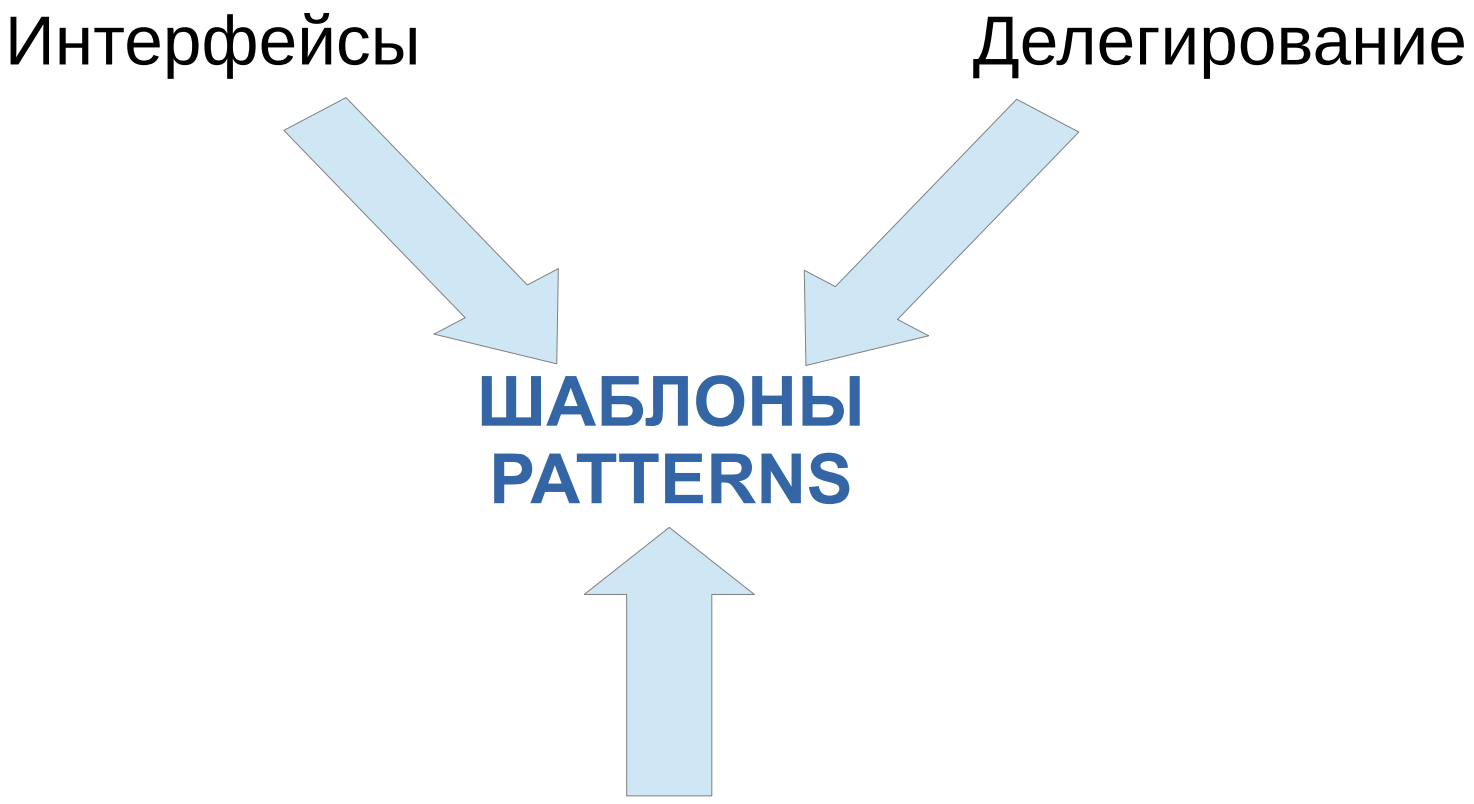
- Делегирование
  - динамическое отношение
  - роль

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

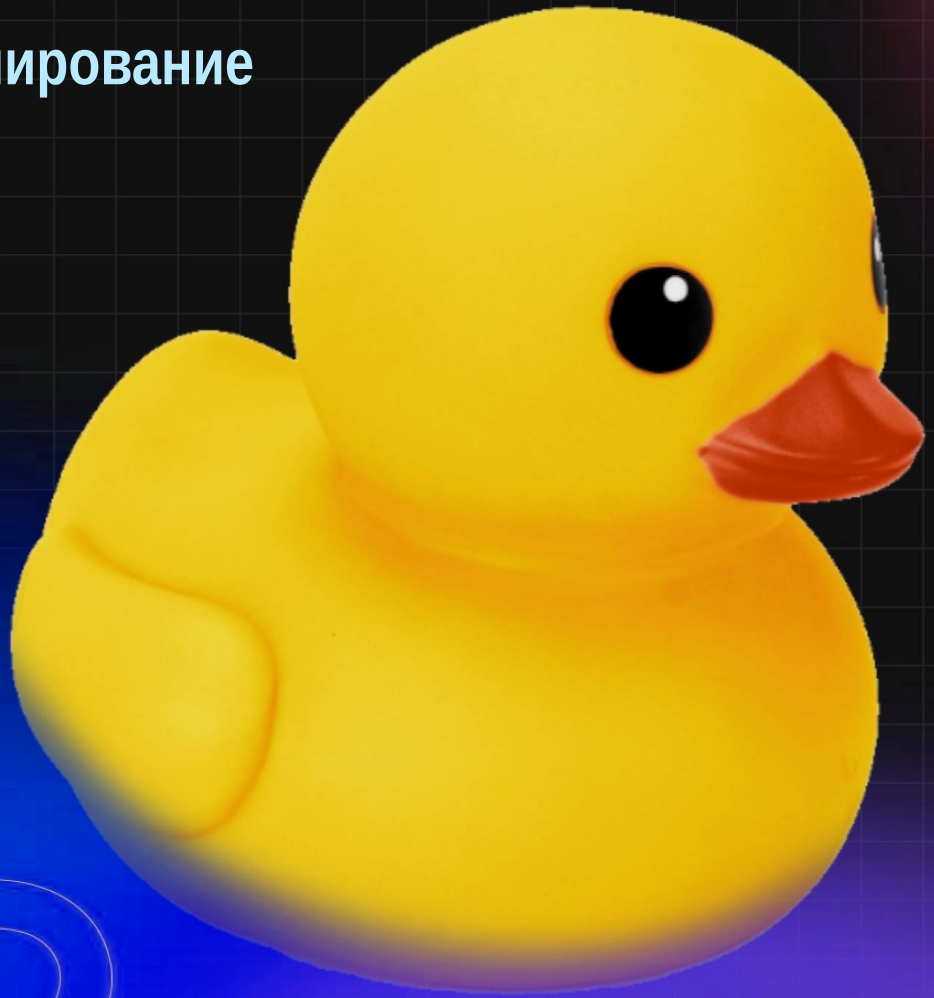
Инкапсуляция изменений

- Отделить изменяющееся от постоянного
- Инкапсулировать изменяющееся

Как справиться с изменениями?



Программирование
2 семестр
2025



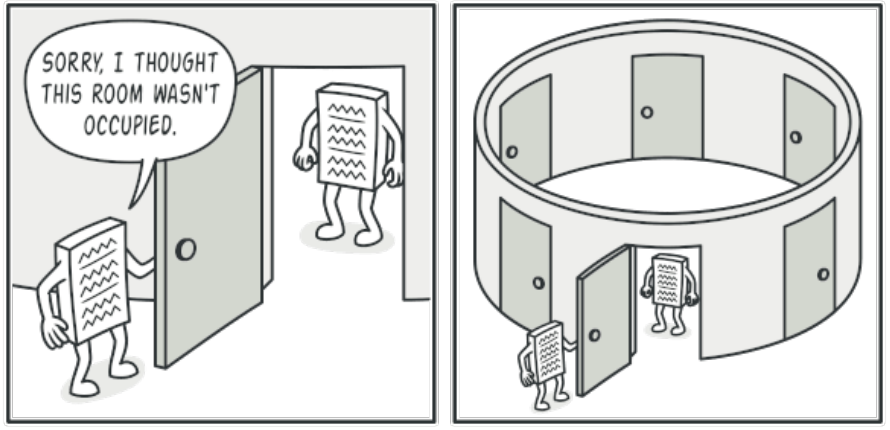
Порождающие шаблоны (Creational)

Порождающие шаблоны

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

Singleton

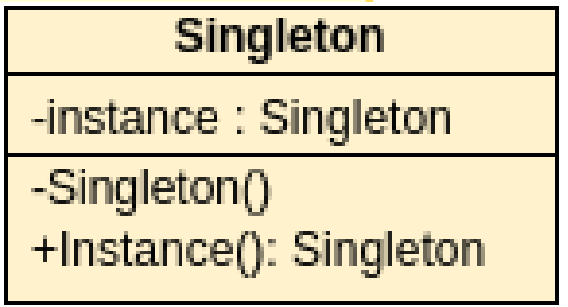
- Одиночка
  - Гарантирует наличие единственного экземпляра класса
  - Предоставляет глобальную точку доступа



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

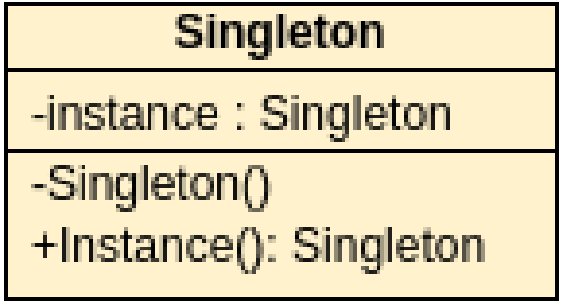
Singleton

```
class Singleton {
  private final static Singleton INSTANCE = new Singleton();

  private Singleton() { }

  public static Singleton getInstance() {
    return INSTANCE;
  }
}
```



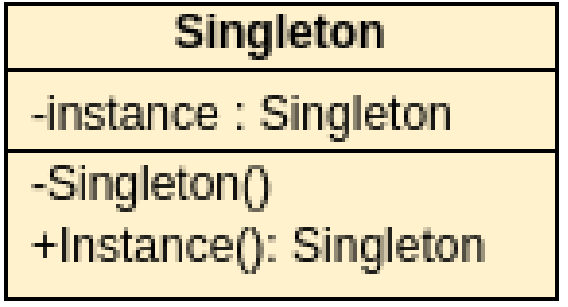
Singleton

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



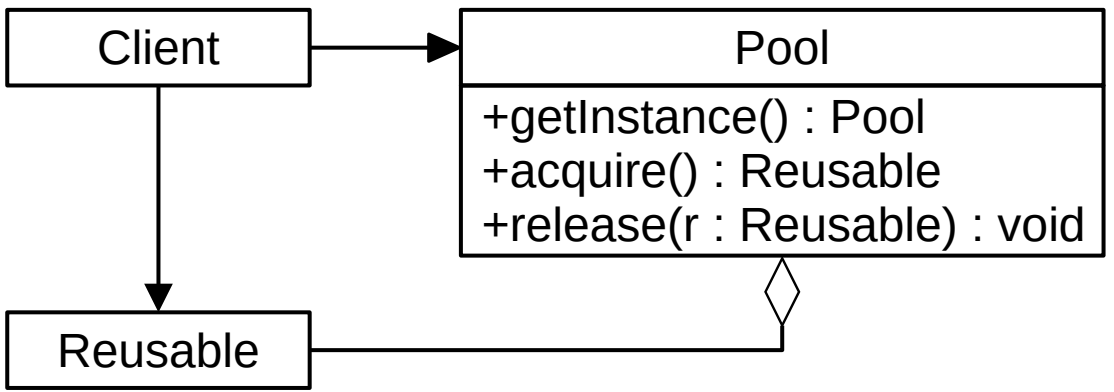
Singleton с ленивой инициализацией

```
class Singleton {
  private static Singleton instance;

  private Singleton() { }

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



Object Pool

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



Simple Factory

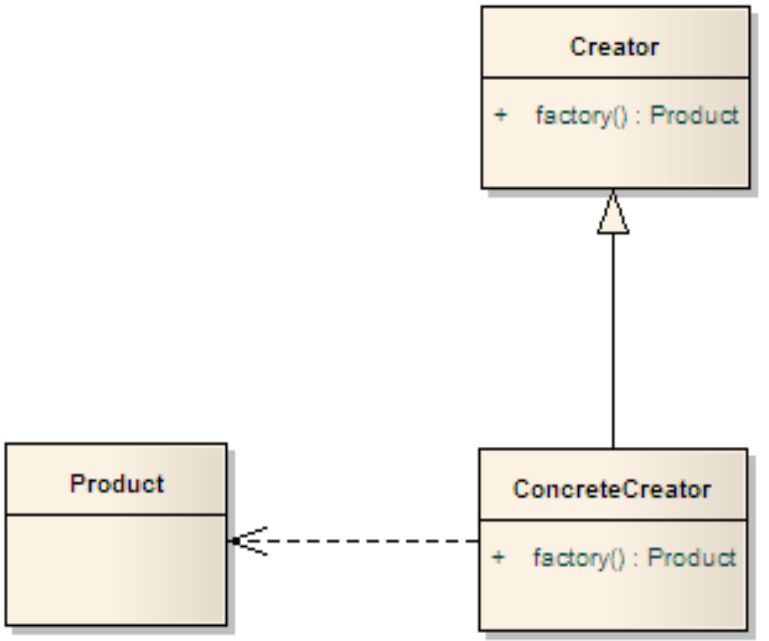
- Простая фабрика (не совсем паттерн)
  - Объект, создающий другие объекты

```
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

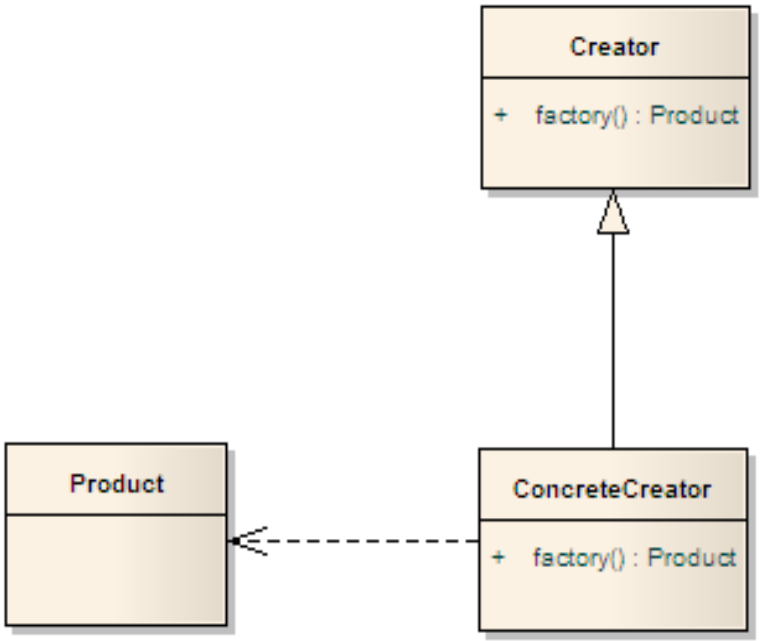
```
class AnimalFactory {
  public Animal getAnimal(int type) {
    return switch(type) {
      case 1 -> new Dog();
      case 2 -> new Cat();
  }
}
```



Simple Factory

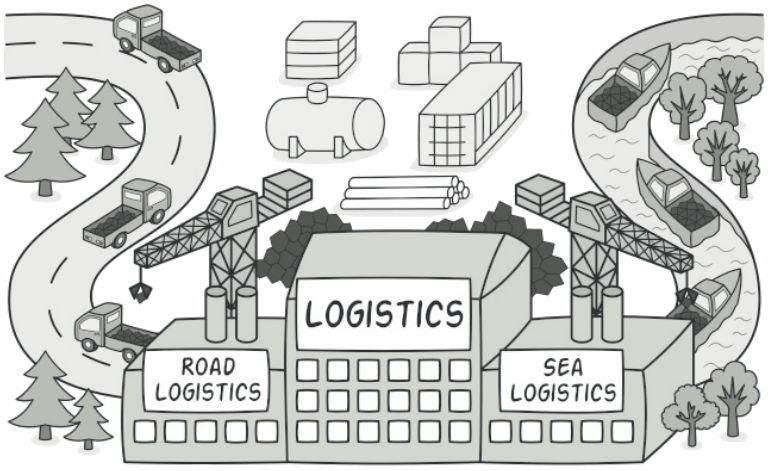
- Простая фабрика (не совсем паттерн)
  - Объект, создающий другие объекты

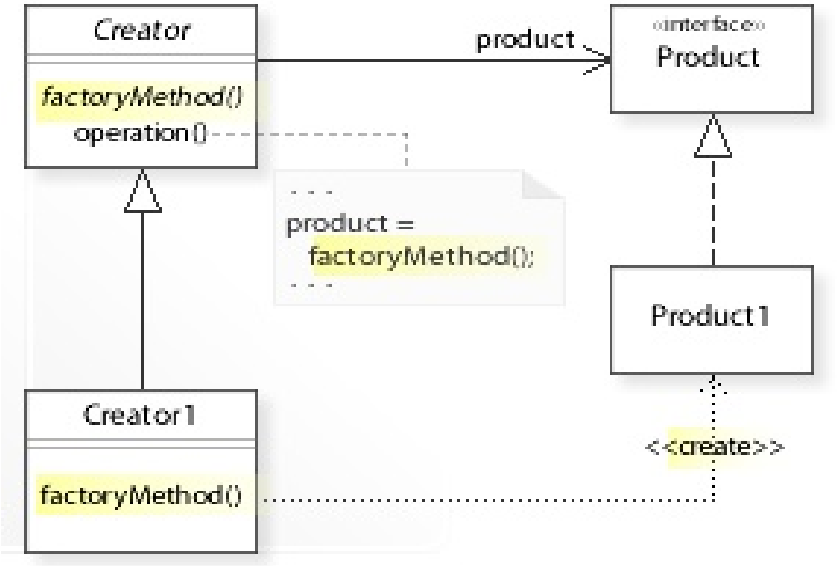
```
Animal[] animals = new Animal[2];
var factory = new AnimalFactory();
animals[0] = factory.getAnimal(1);
animals[1] = factory.getAnimal(2);
for (Animal a : animals) {
  a.sleep();
  a.makeSound();
}
```



Factory Method

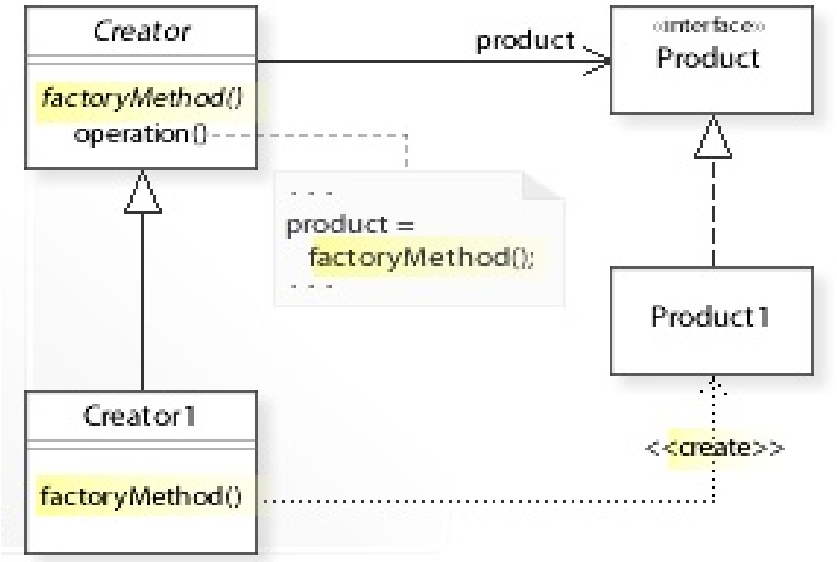
- Фабричный метод
  - Созданием объектов занимаются подклассы





Factory Method

- Фабричный метод
  - Созданием объектов занимаются подклассы
- Product - общий интерфейс
- Product1 - конкретный продукт
- Creator - интерфейс фабрики
  - Product factoryMethod()
  - другие методы...
- Creator1 - создает Product1



Factory Method

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

```
class AnimalFactory {
  protected abstract Animal getAnimal();
}

class DogFactory extends AnimalFactory {
  public Animal getAnimal() {
    return new Dog();
  }
}
```

Factory Method - вариант

- Конкретный продукт возвращает статический метод абстрактного класса или интерфейса Product

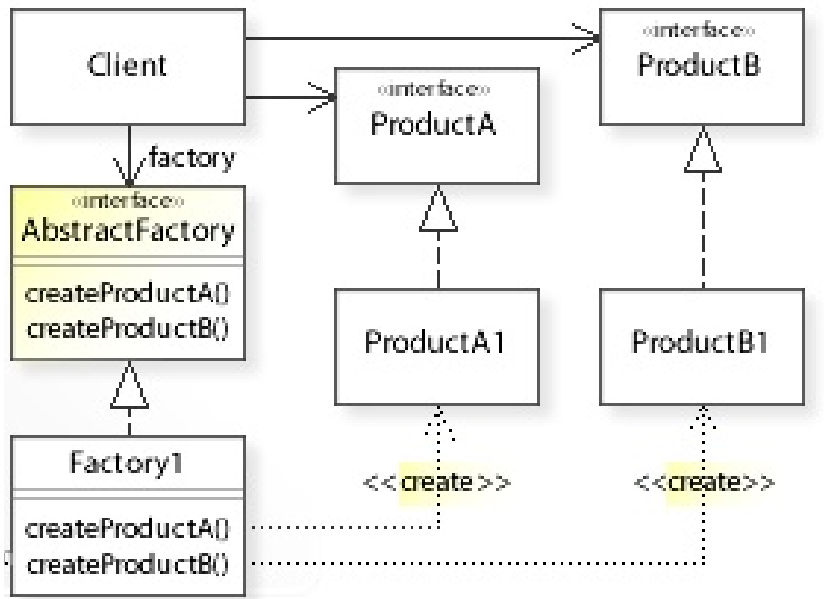
```
Color.getRGB(int red, int green, int blue);
Color.getHSL(int hue, int saturation, int lightness);

NumberFormat.getNumberInstance(Locale.FR);

DriverManager.getConnection(String url);
```

Factory Method

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

Abstract Factory

- Абстрактная фабрика
  - Фабрика для создания семейства продуктов



Abstract Factory

ИТМО

- 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)

Java

Abstract Factory

- Абстрактная фабрика
  - Фабрика для создания семейства продуктов

```
Factory colaFact = Provider.getFactory("Coca-Cola");
Drink sprite = colaFact.getLemonDrink();

Factory pepsiFact = Provider.getFactory("Pepsi");
Drink mirinda = pepsiFact.getOrangeDrink();
```

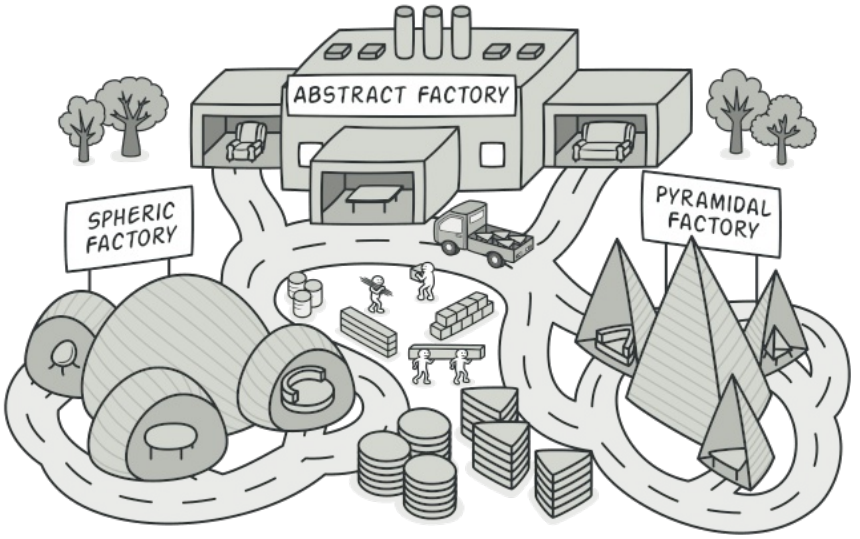
Abstract Factory

- Абстрактная фабрика
  - Connection conn = DriverManager.getConnection(args)

```
Factory colaFact = Provider.getFactory("Coca-Cola");
Drink sprite = colaFact.getLemonDrink();

Factory pepsiFact = Provider.getFactory("Pepsi");
Drink mirinda = pepsiFact.getOrangeDrink();

Factory kindFact = Provider.getFactory("Добрый");
Drink dobryCola = kindFact.getColaDrink();
```

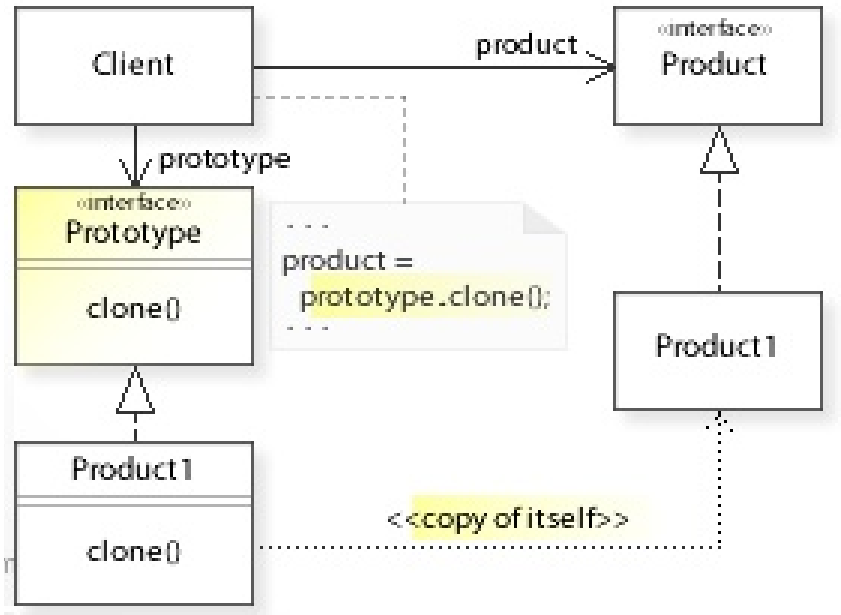
Abstract Factory

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



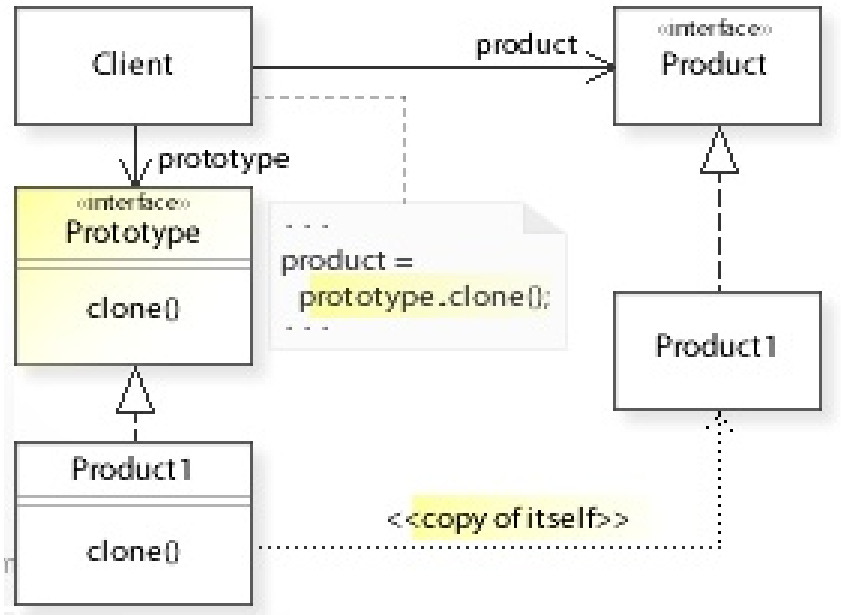
Prototype

- Прототип
  - Клонирование вместо создания



Prototype

- 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



Prototype

ИТМО

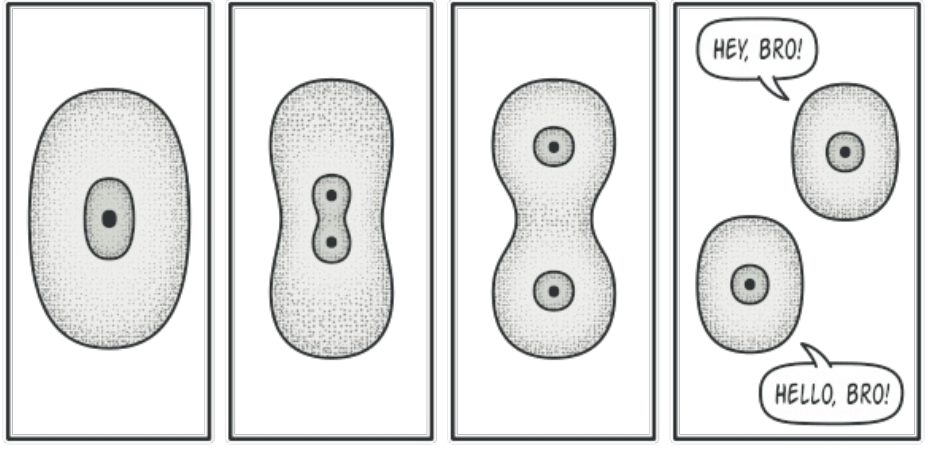
- Глубокая и неглубокая копия (deep / shallow copy)

```
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);
}
```

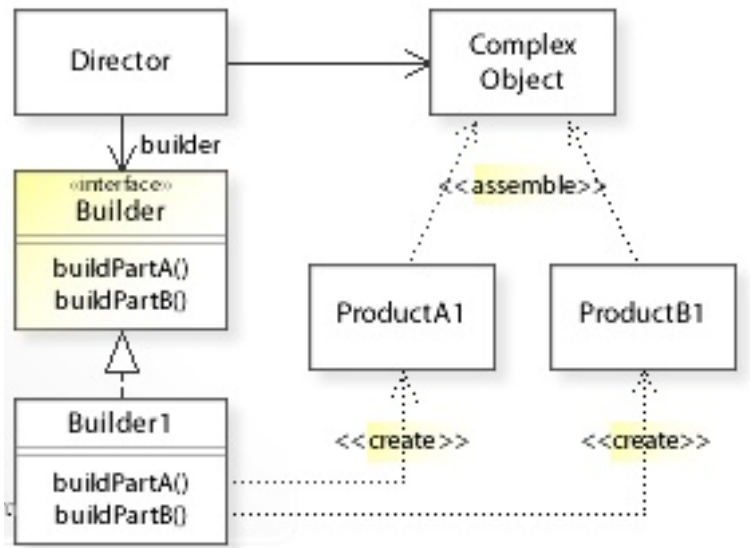
Prototype

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



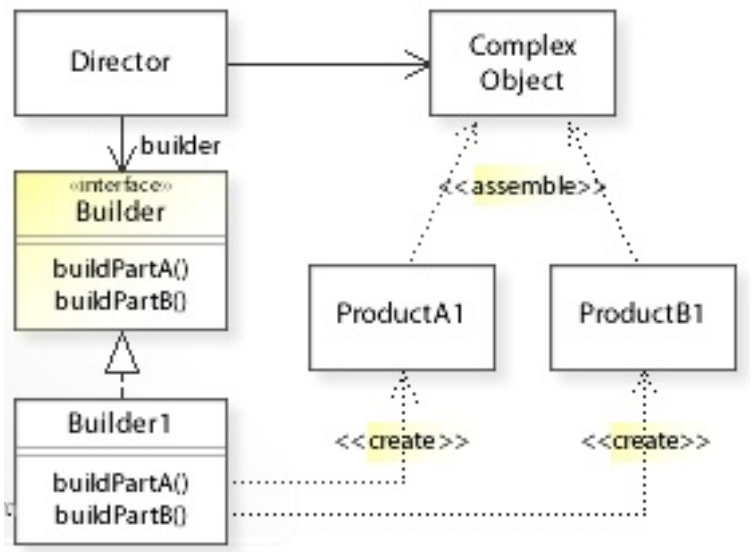
Builder

- Строитель
  - Строит сложный объект по шагам



Builder

- 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



Builder

- 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;
  ...
}
```

Builder

- 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 localtion)
  ...
}
```

Builder

- 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)
```

Builder

- 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)
  ...
}
```

Builder

- 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();
```

Builder

- Строитель создает объект по шагам
- Методы возвращают строителя

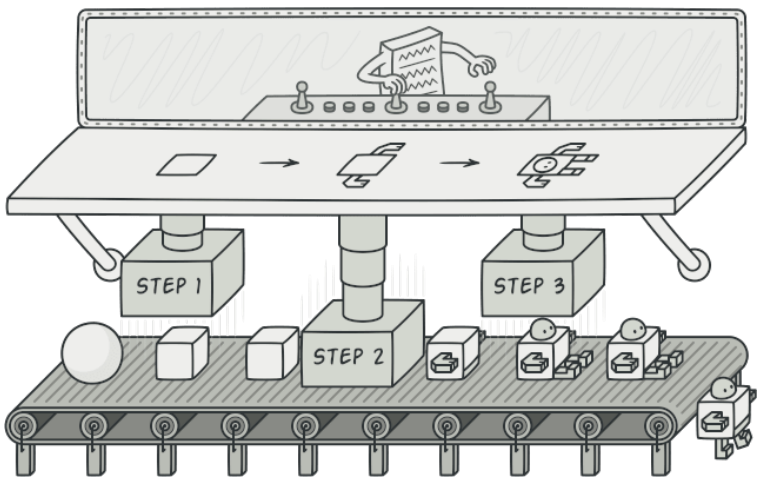
```
Builder b = new Builder()
   .setWhere("Moscow")
   .setTransport("train")
   .setDate(1,7,2023);
   ...

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

- Директор создает объект с помощью строителя одним методом

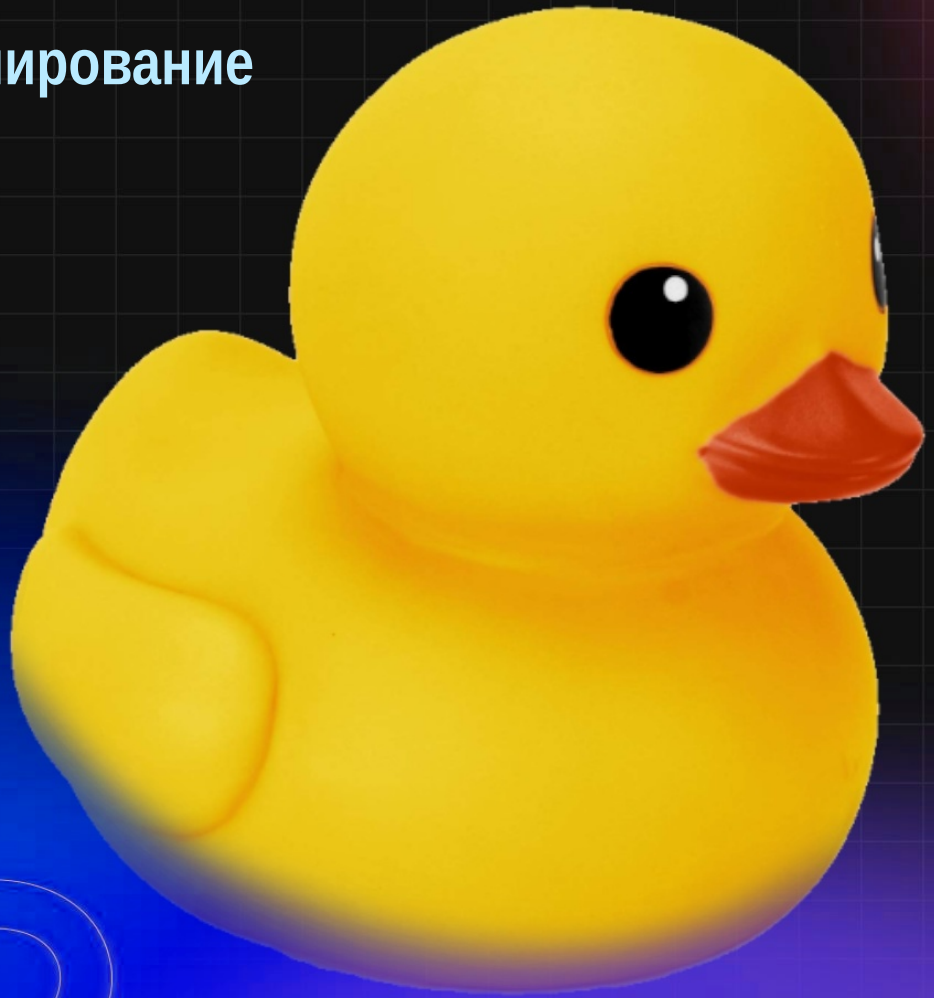
```
Director d = new Director();
Trip trip = d.makeTrainTrip("Moscow", new Date(1,7,2023));
```

Builder

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



Программирование
2 семестр
2025



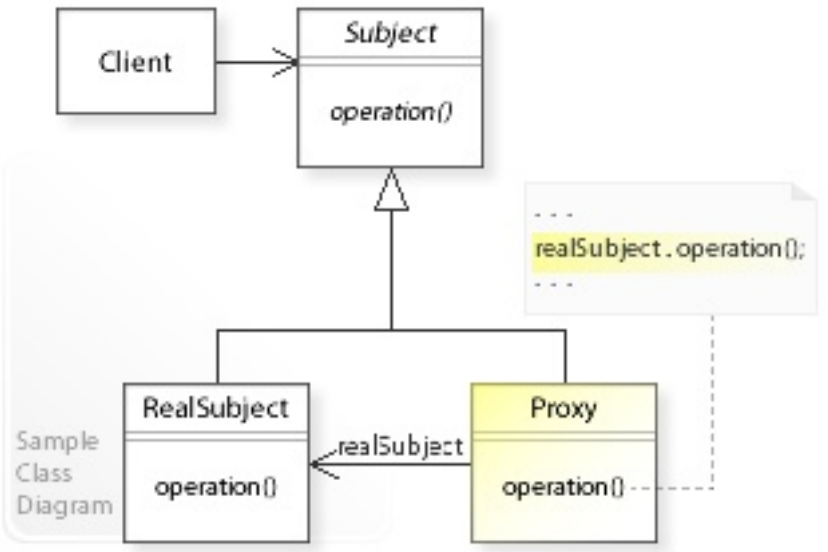
Структурные шаблоны (Structural)

Структурные шаблоны

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

Proxy

- Заместитель - делегирование **с тем же интерфейсом**
- Подмена реального объекта заместителем
- java.lang.reflect.Proxy

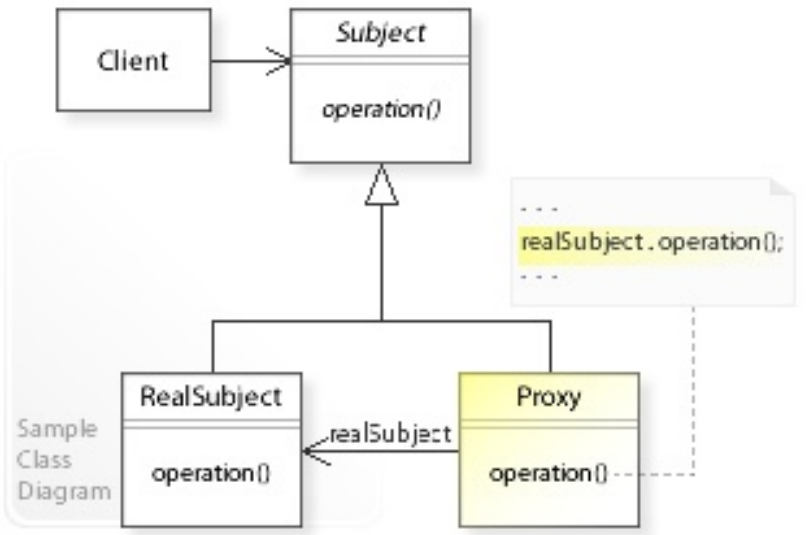


Proxy

- Заместитель

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

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



Proxy

- Заместитель

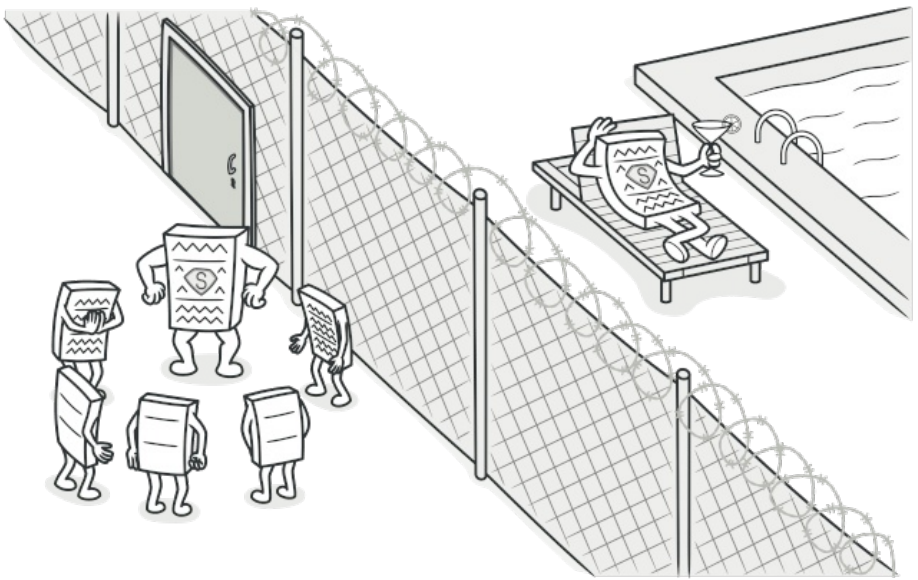
```
class Proxy implements Subject {
  RealSubject real;
  public request() {
    if (real == null) {
      real = new RealSubject();
    }
    return real.request();
  }
}
```

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

- Разновидности Proxy
  - Virtual - ленивое создание
  - Remote - удаленный объект
  - Security - контроль доступа
  - Caching - кэширование
  - Smart - логирование

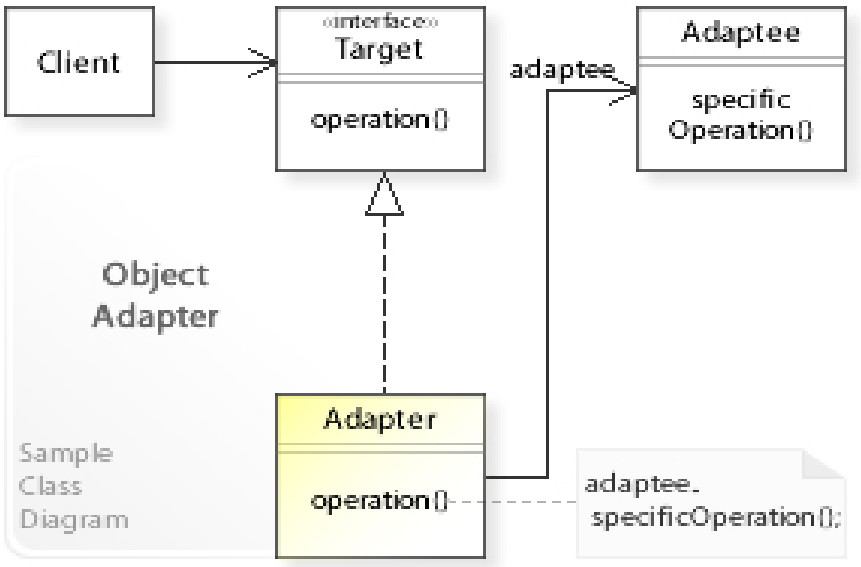
Proxy

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



Adapter

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



Adapter

- Адаптирует один интерфейс к другому

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

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

```
EUPlug myPlug = myDevice.getPlug();
EUSocket roomSocket = room.getSocket();

myPlug.connect(roomSocket);
```

Adapter

- Адаптирует один интерфейс к другому

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

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

```
EUPlug myPlug = myDevice.getPlug();
UKSocket roomSocket = room.getSocket();

myPlug.connect(roomSocket);
```

Adapter

- Адаптирует один интерфейс к другому

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

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

```
EUPlug myPlug = myDevice.getPlug();
UKSocket roomSocket = room.getSocket();

myPlug.connect(roomSocket);
```

Adapter

- Адаптирует один интерфейс к другому

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


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

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

```
EUPlug myPlug = myDevice.getPlug();
UKSocket roomSocket = room.getSocket();

myPlug.connect(roomSocket);
```

Adapter

- Адаптирует один интерфейс к другому

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

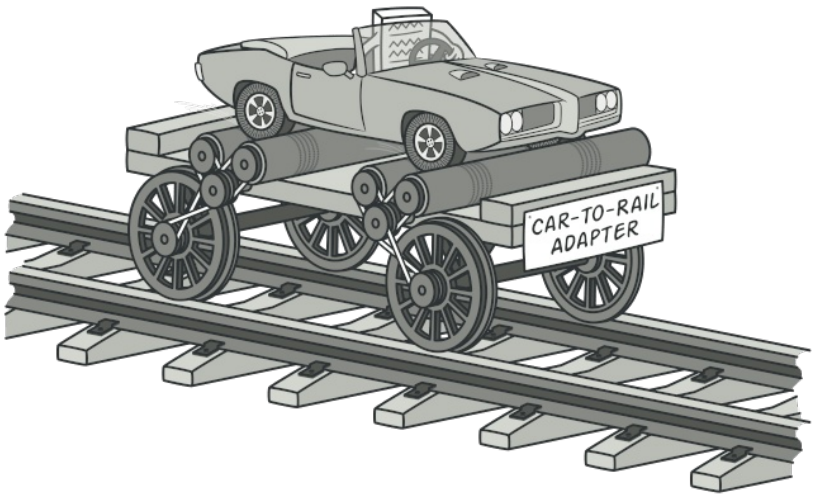
```
class Adapter implements EUSocket {
  UKSocket uks;
  Adapter(UKSocket s) { uks = s; }
  getEUPower() { uks.getUKPower(); }
}
```

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

```
EUPlug myPlug = myDevice.getPlug();
UKSocket roomSocket = room.getSocket();
EUSocket adapter = new Adapter(roomSocket);
myPlug.connect(adapter);
```

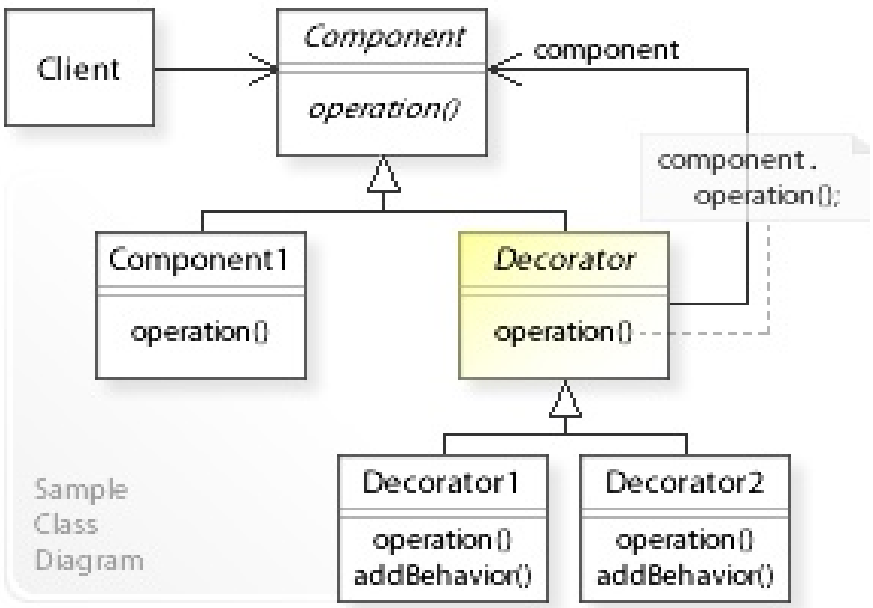
Adapter

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



Decorator

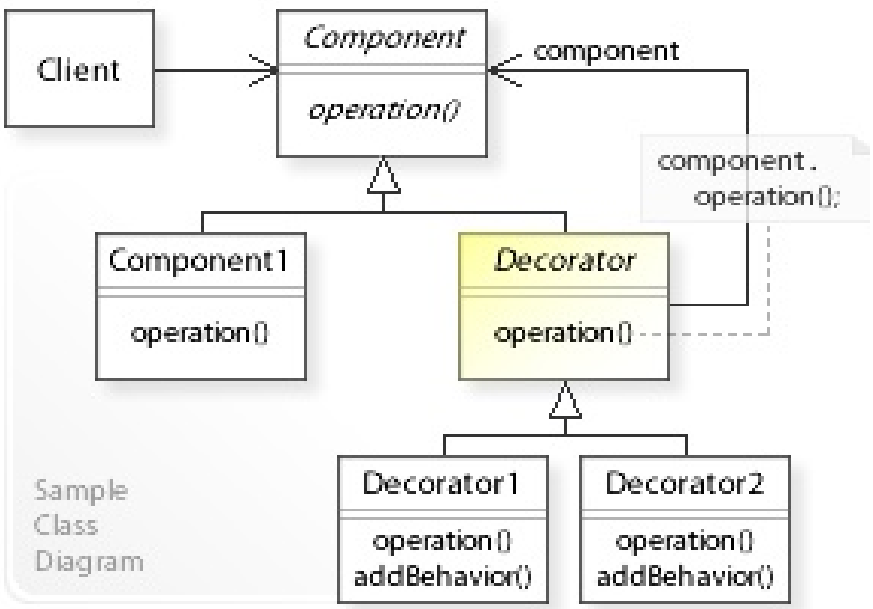
- Декоратор - делегирование **с новой функциональностью**
- Блины с добавками в Теремке
- InputStream
  - BufferedInputStream,
  - LineNumberInputStream,
  - ...



Decorator

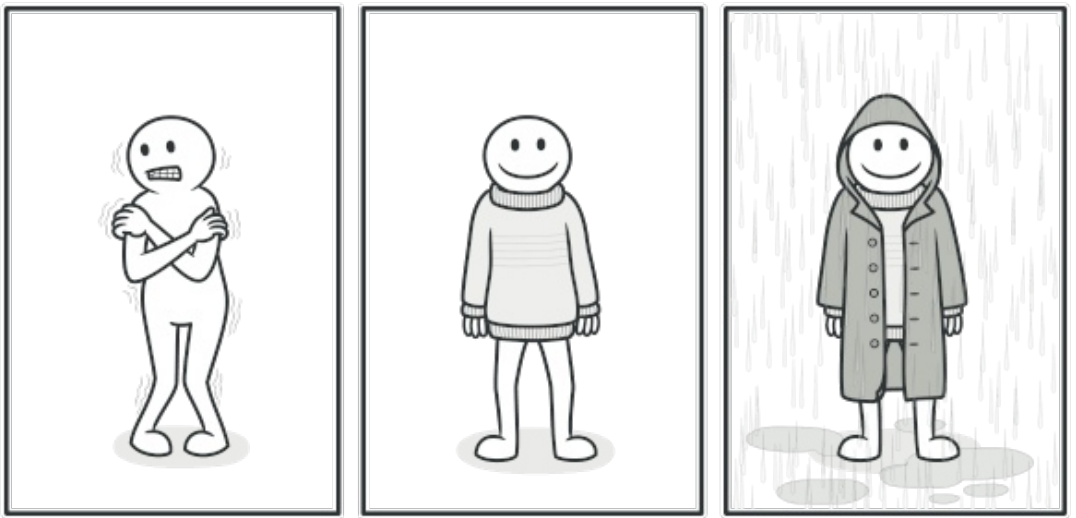
- Делегирование с расширением функциональности

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



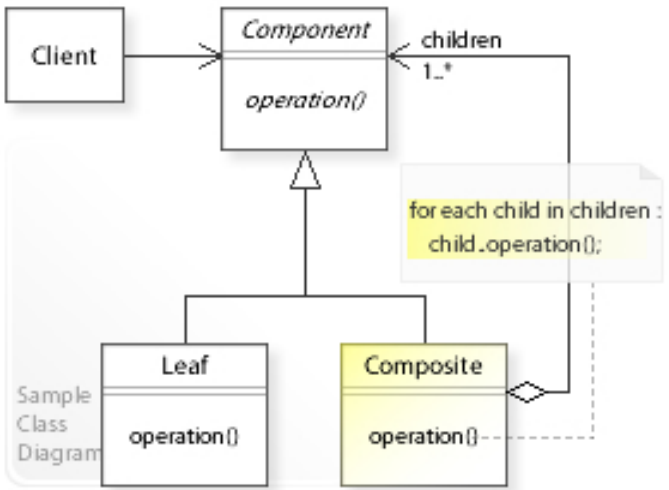
Decorator

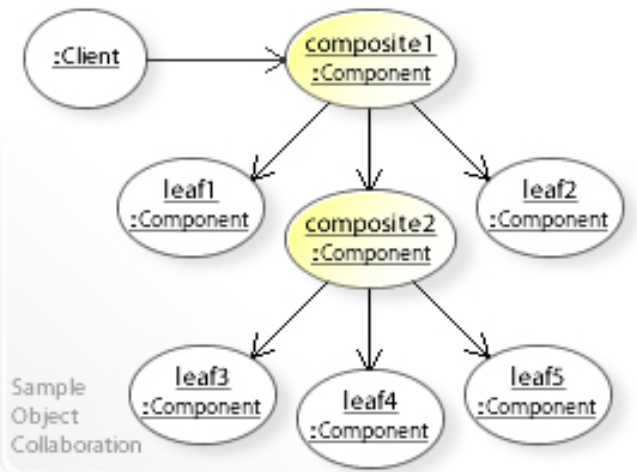
- Позволяет добавлять функциональность динамически
- Вместо большой иерархии - несколько декораторов
- Сложность конфигурирования



Composite

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





Composite

ИТМО

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

```
class Component {
  public operation() {
    // что-то делаем
  }
}
```

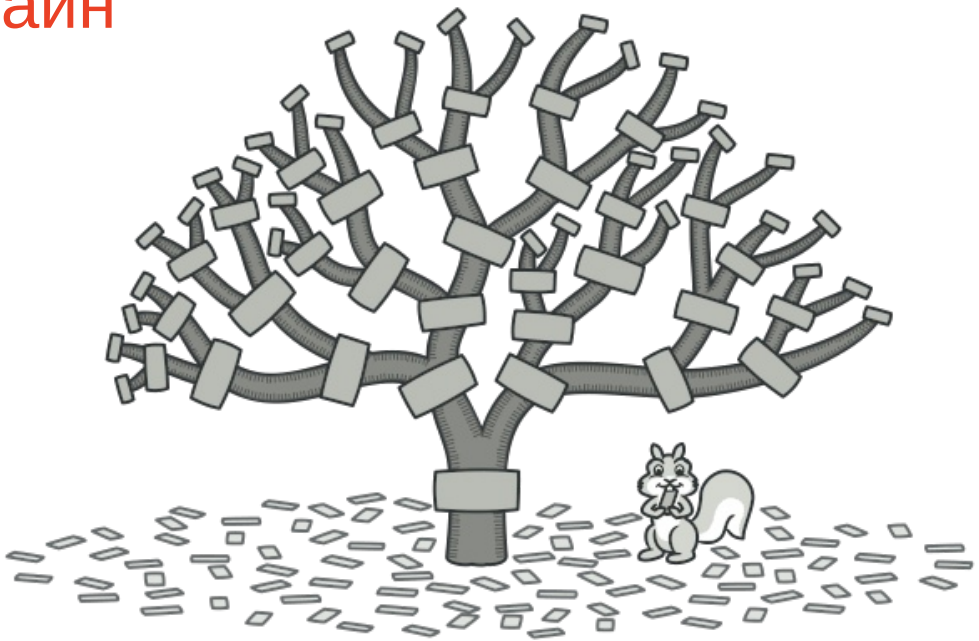
```
class Composite extends Component {
  List<Component> children;
  public operation() {
    for (var c : children) {
      c.operation();
    }
  }
}
```

Composite

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

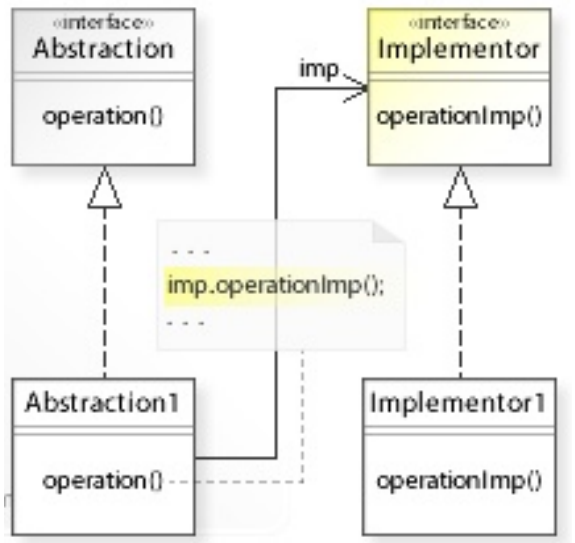
Composite

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



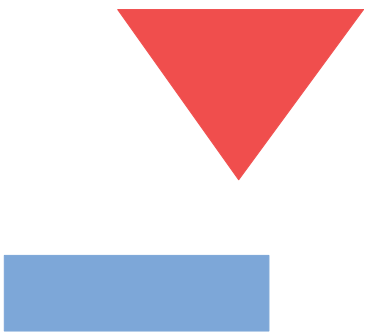
Bridge

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



Bridge

```
class Triangle
class Rectangle
```



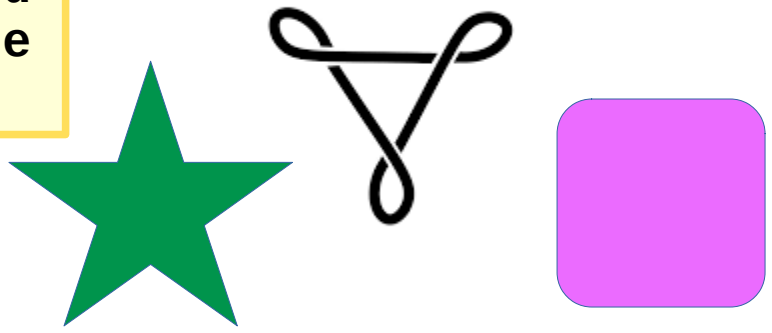
Bridge

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

Bridge

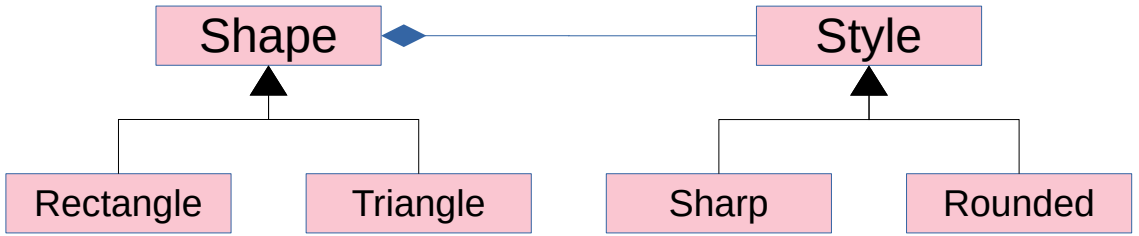
```
class SharpTriangle
class RoundedTriangle
class SharpRectangle
class RoundedRectangle
```

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



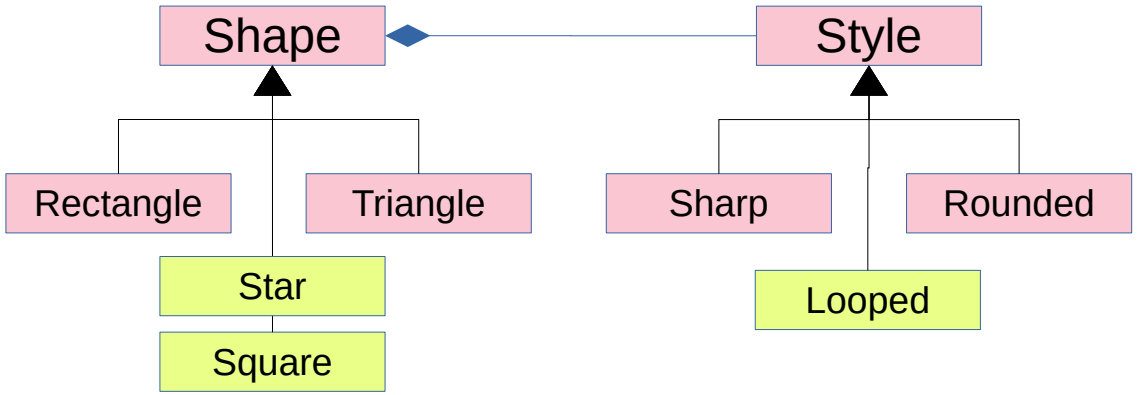
```
                +
class LoopedTriangle
class RoundedSquare
class SharpStar
                ...
```

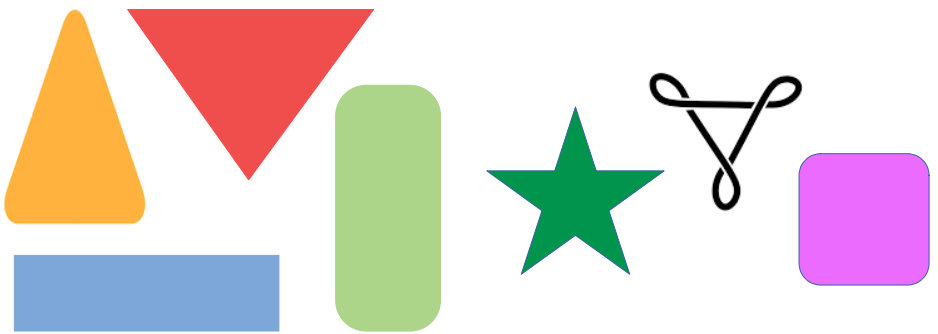
Bridge



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

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

Bridge

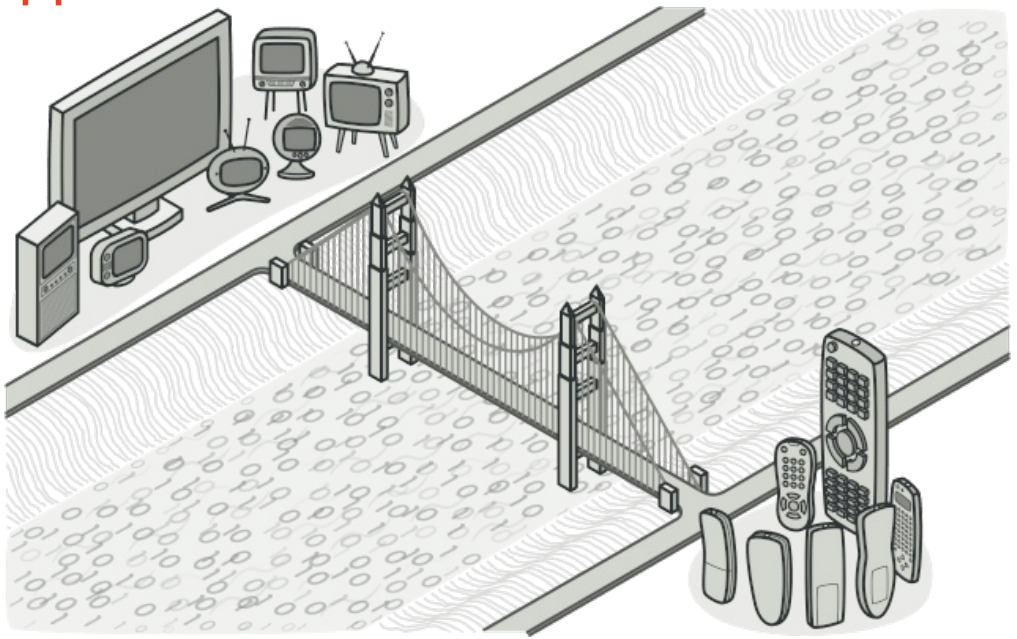




```
abstract class Shape {
  Style style;
  setStyle(Style style) {
    this.style = style;
  }
  abstract draw() {
    ...
    style.drawCorner();
    ...
  }
}

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

Bridge

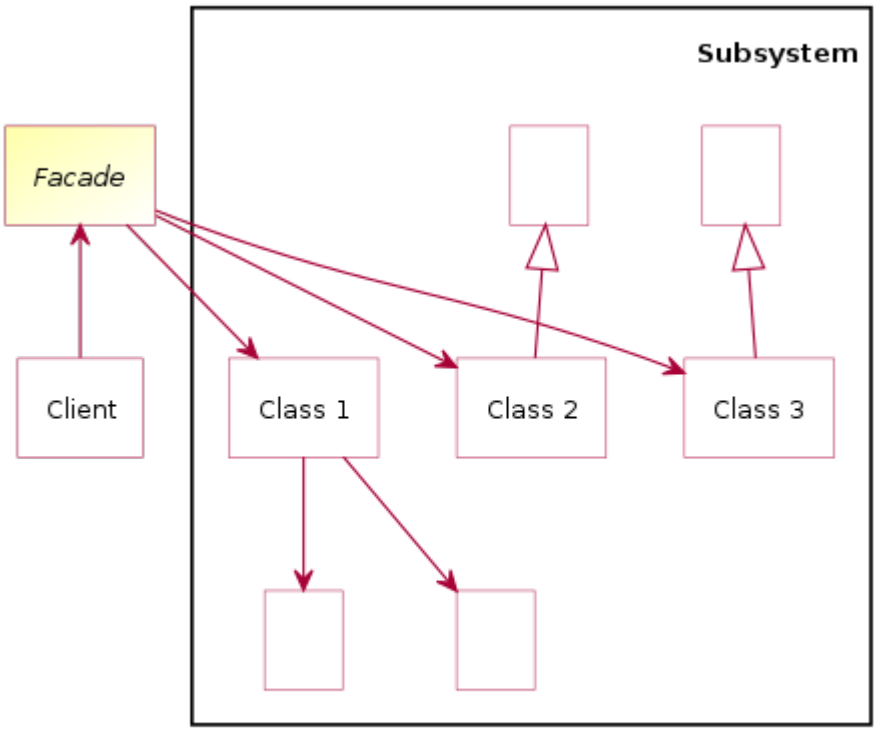
- Повышение расширяемости
- Устраняет сложность поддержки нескольких иерархий
- Появление дополнительных классов



Facade

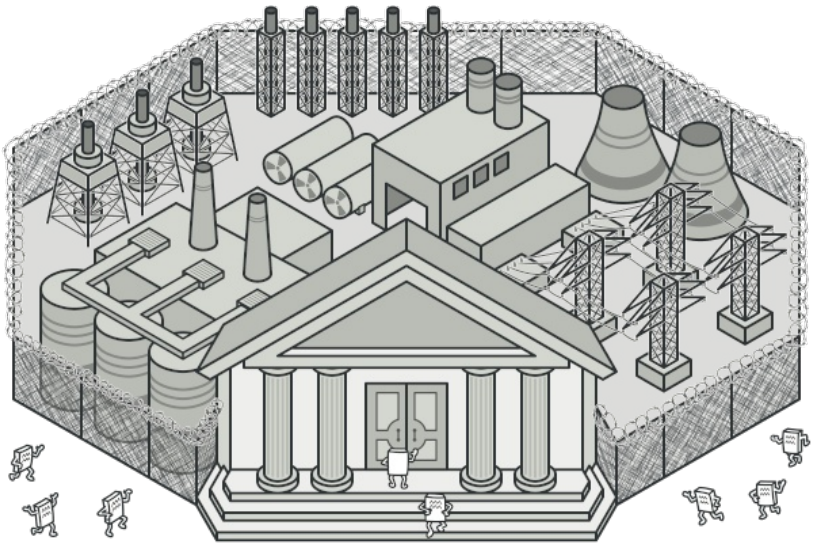
- Фасад - простой интерфейс к сложным системам
- Система может меняться
- Доступ через фасад

```
class Facade {
  public go() {
    c0.connect(c1,c2,c3,c42);
    c10.activate(c11,c42,c6);
    c24.switch(c5,c3,c2,c42);
    c999.check(c7,c42,c1,c2);
    c0.start(c42,c2025,c999);
  }
}
```



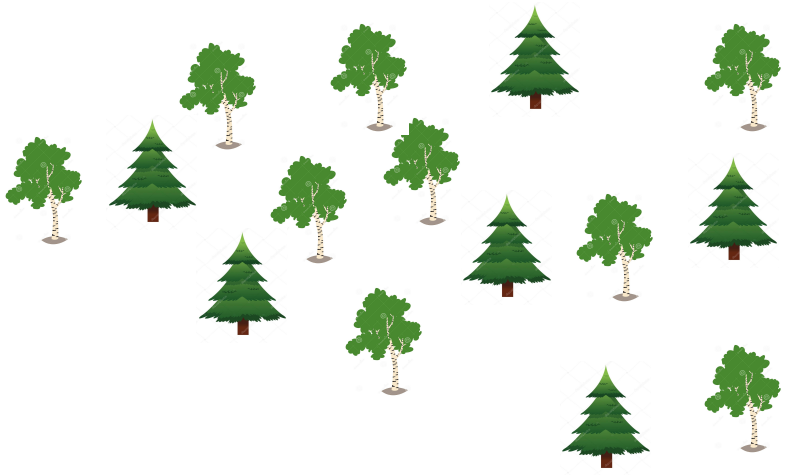
Facade

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



Flyweight

- Легковес - экономия памяти при большом числе объектов



Flyweight

- 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');
```

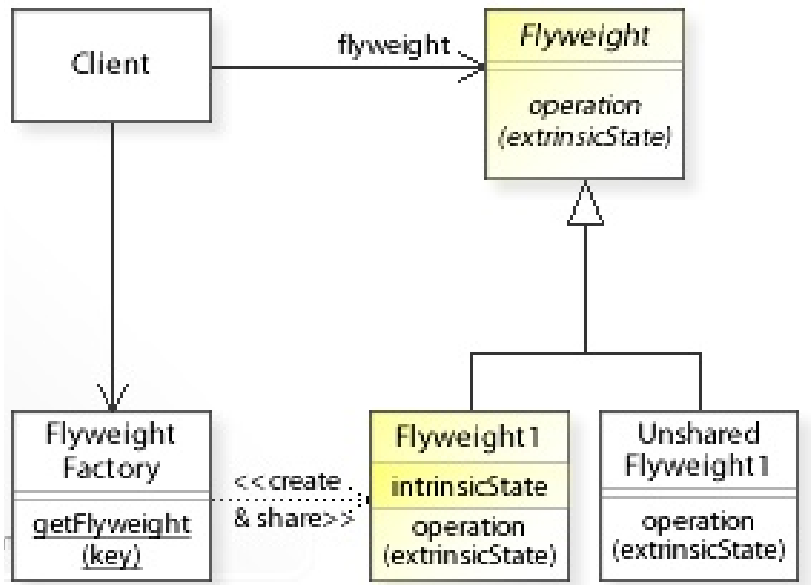
Flyweight

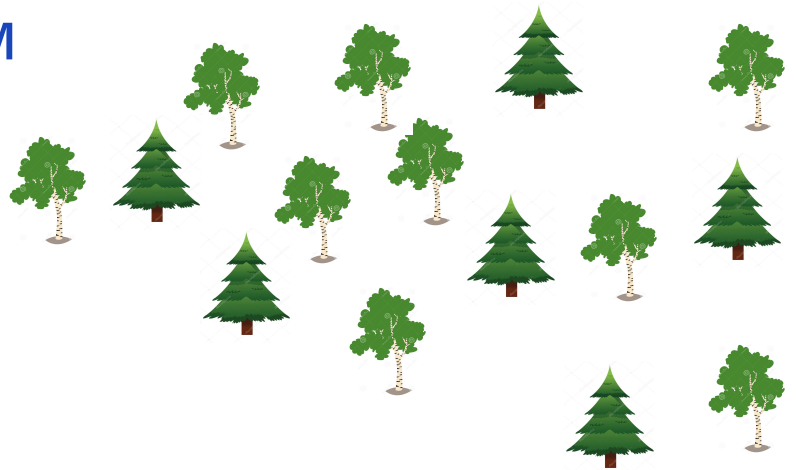
- Легковес - экономия памяти при большом числе объектов
- Каждый объект - свое состояние
- Большой расход памяти

```
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');
```

Flyweight

- Уникальное состояние - храним для каждого объекта
- Общее состояние - не дублируем

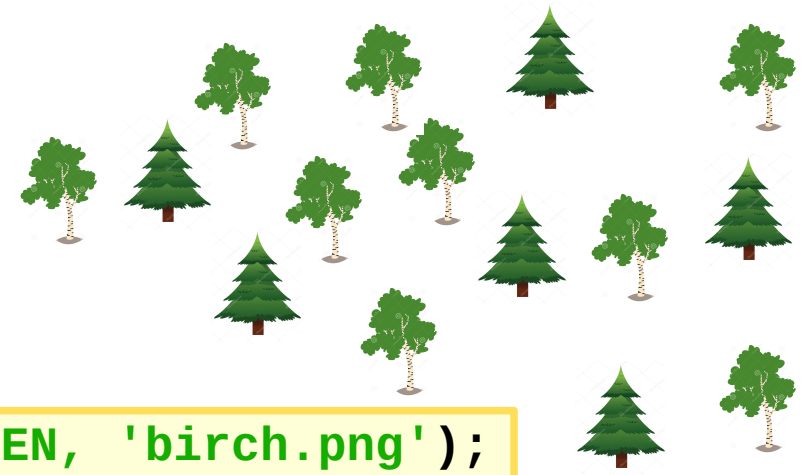




Flyweight

```
class TreeType {
  Image image;
  Color color;
}
```

```
class Tree {
  int x, y;
  TreeType type;
}
```

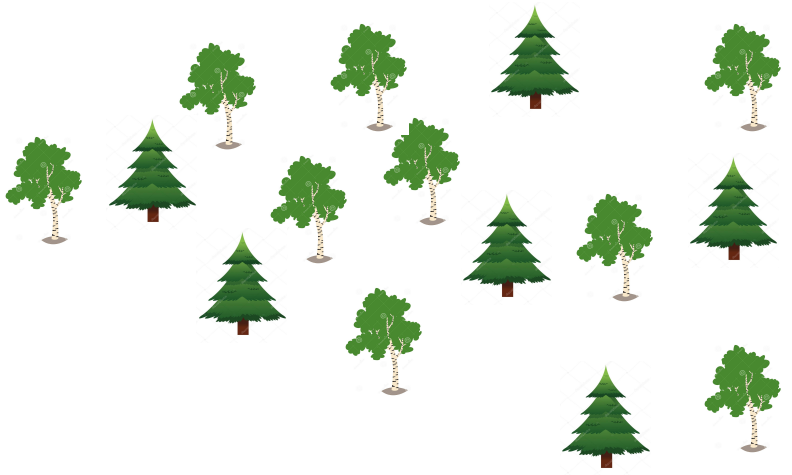


```
TreeType birch = new TreeType(Color.GREEN, 'birch.png');
TreeType pine = new TreeType(Color.GREEN, 'pine.png');
Tree t1 = new Tree(15,35,birch);
Tree t1 = new Tree(27,31,pine);
Tree t1 = new Tree(20,40,birch);
Tree t1 = new Tree(32,14,pine);
```

Flyweight

```
class TreeType {
  Image image;
  Color color;
  draw(int x, int y, Color c);
}
```

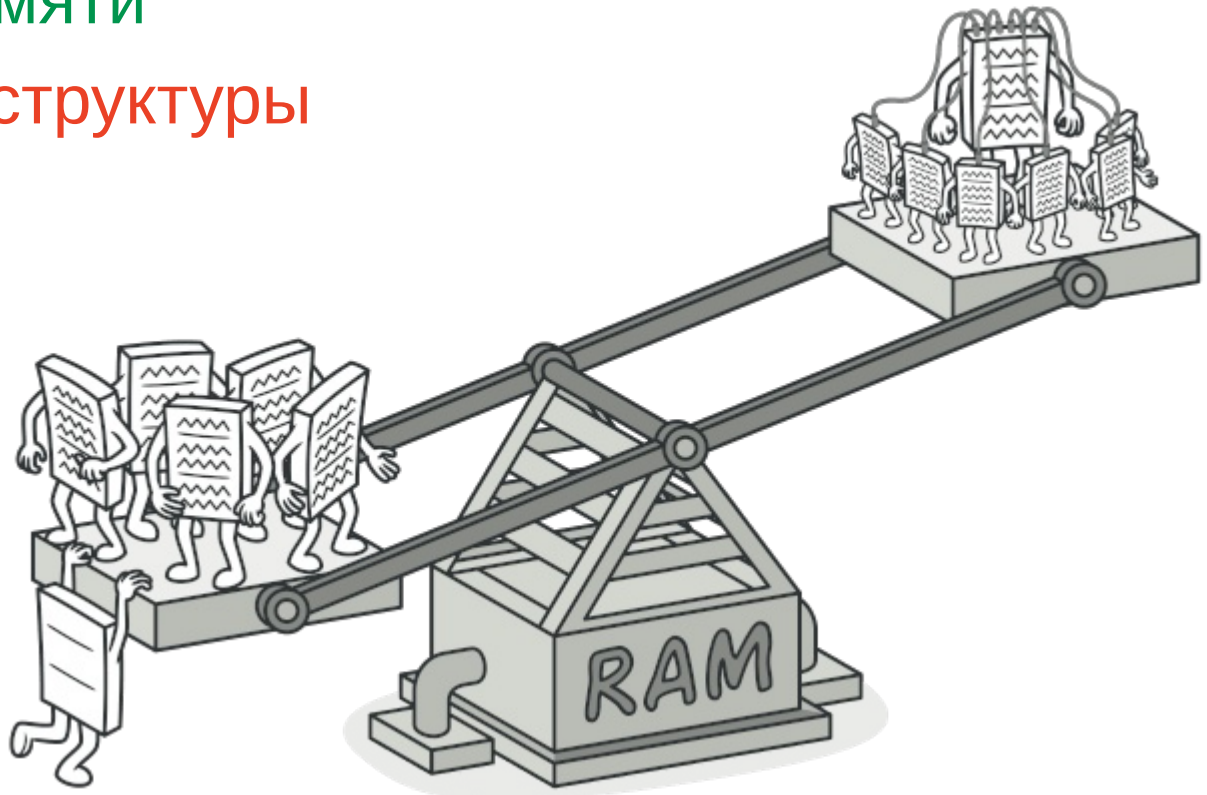
```
class Trees {
  TreeType tree;
  int[][] coords;
  drawAllTrees() {
    for (var c : coords) {
      tree.draw(c[0],c[1]);
    }
  }
}
```



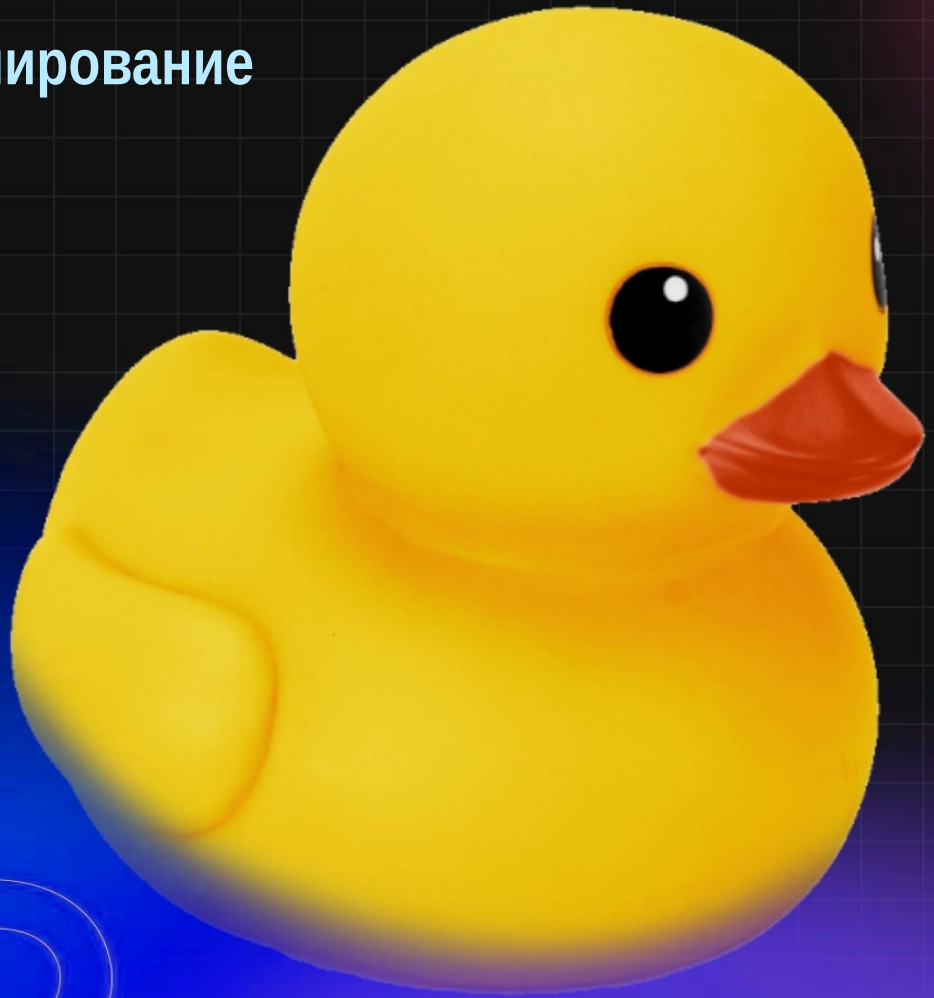
```
TreeType birch = new TreeType(Color.GREEN,'birch.png');
TreeType pine = new TreeType(Color.GREEN,'pine.png');
Trees pines = new Trees(pine,coords1);
Trees birches = new Trees(birch,coords2);
```

Flyweight

- Экономия памяти
- Усложнение структуры



Программирование
2 семестр
2025



Поведенческие шаблоны (Behavioural)

Поведенческие шаблоны

- 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

Chain of Responsibility

- 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

Chain of Responsibility

ИТМО

- 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

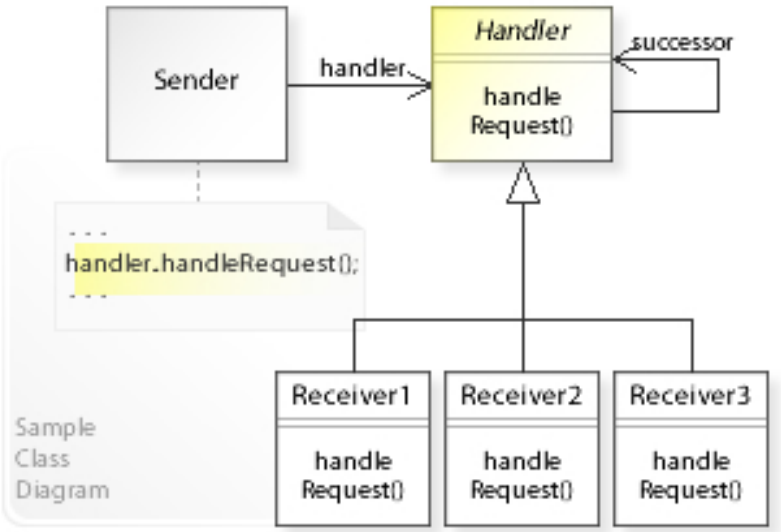
Chain of Responsibility

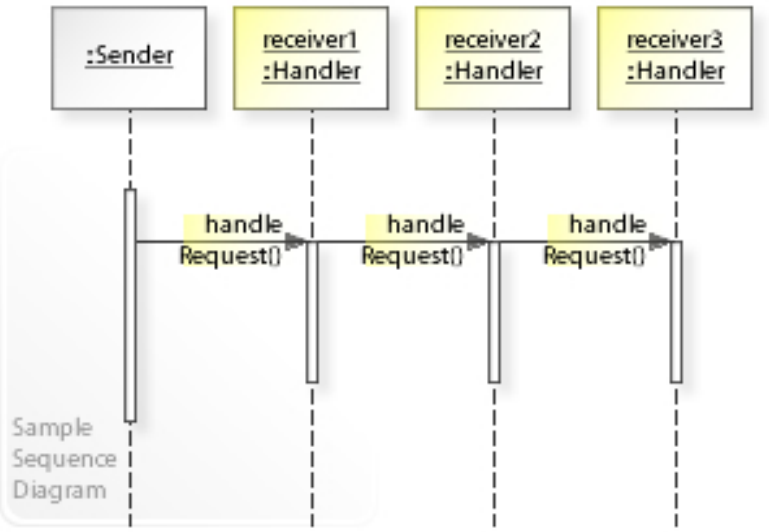
ИТМО

- 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

Chain of Responsibility

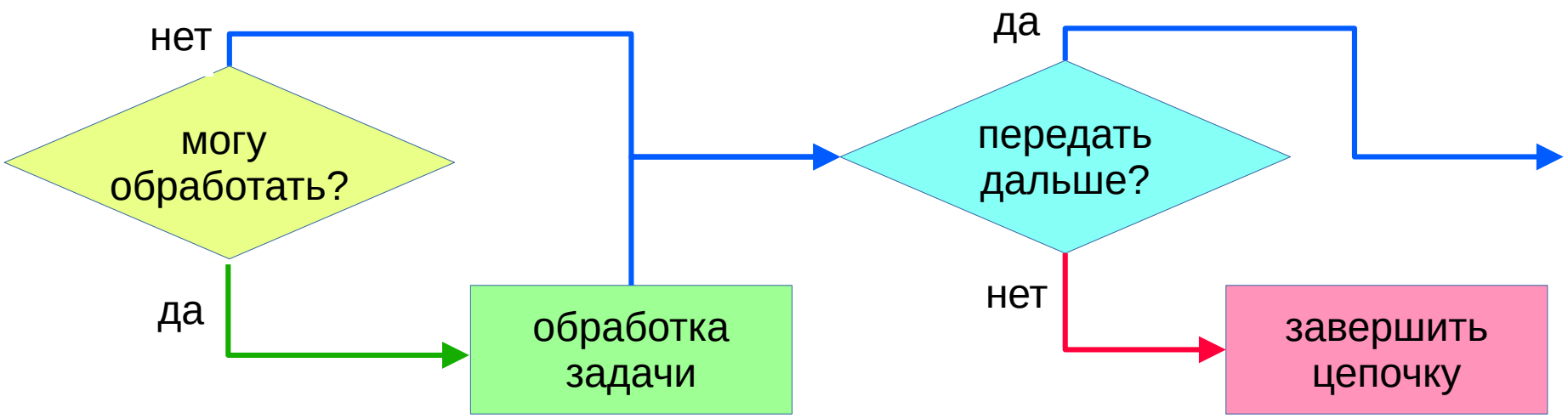
- Передача запроса по цепочке обработчиков





Chain of Responsibility

- Передача запроса по цепочке обработчиков
- Варианты действий



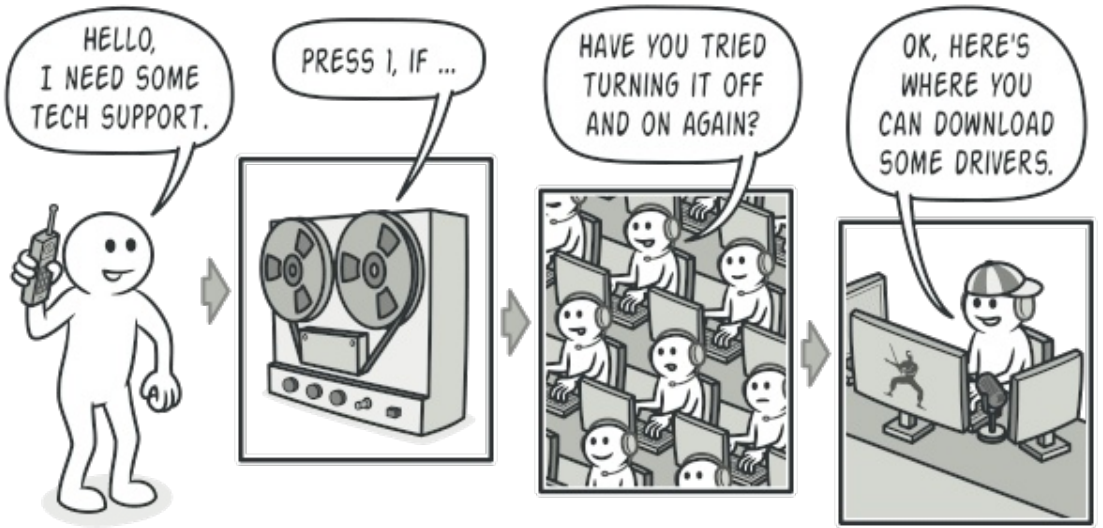
Chain of Responsibility

```
abstract class Handler {
  Handler next;
  setNext(Handler h) { next = h; }
  abstract handleRequest(int id);
}
```

```
class ConcreteHandler1 extends Handler {
  handleRequest(int id) {
    if (id > 3) {
      next.handleRequest(id);
    } else {
      makeActions();
    }
  }
}
```

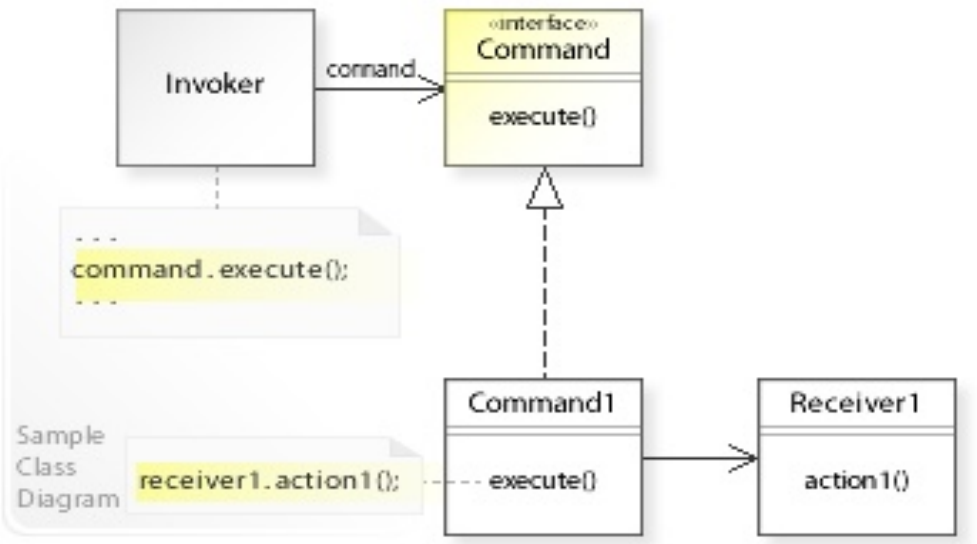
Chain of Responsibility

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



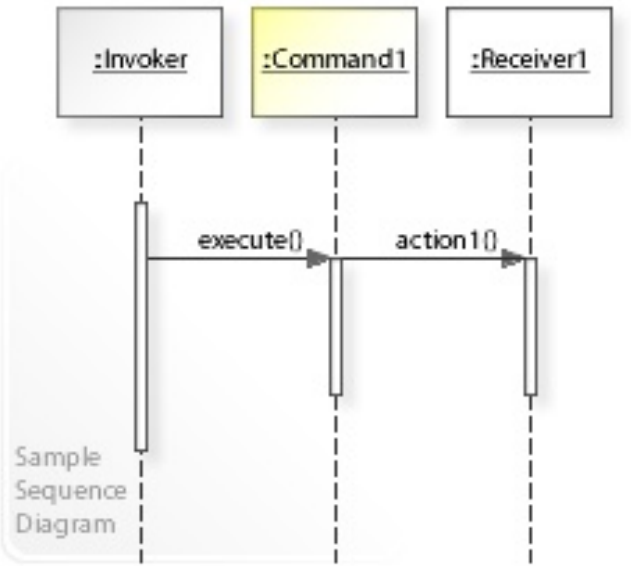
Command

- Команда - разделение вызова и исполнения команд
- Invoker - хранит ссылку на команду
- Command - команда
- Receiver - исполнитель



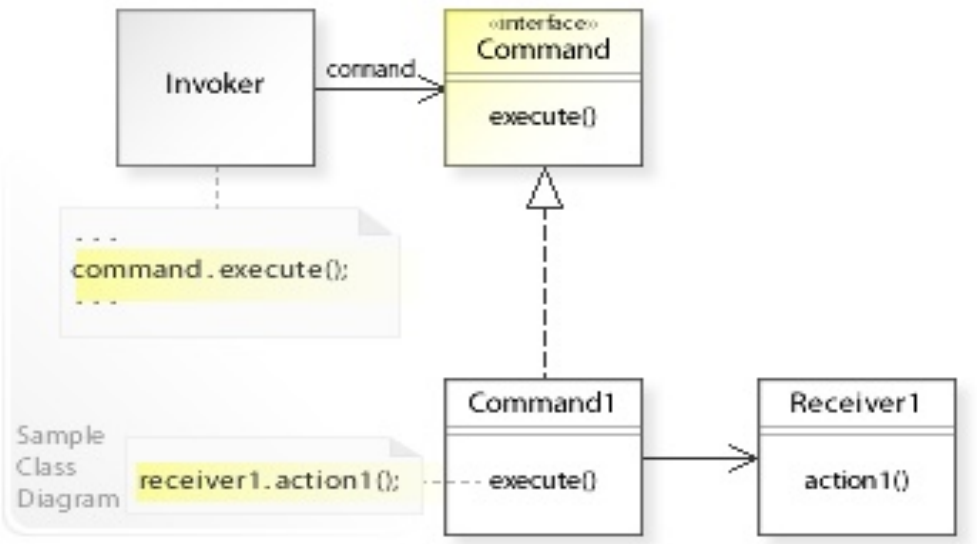
Command

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



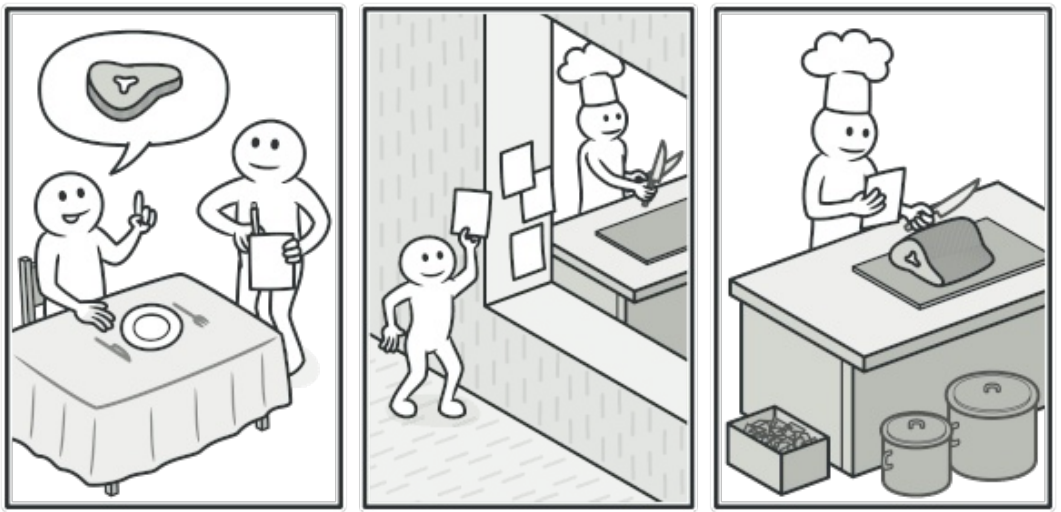
Command

- Команда - разделение вызова и исполнения команд
- Дополнительные возможности
  - Очередь команд
  - Макрокоманды
  - История команд



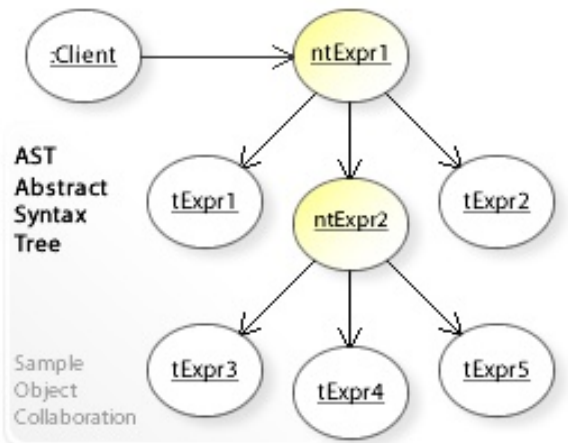
Command

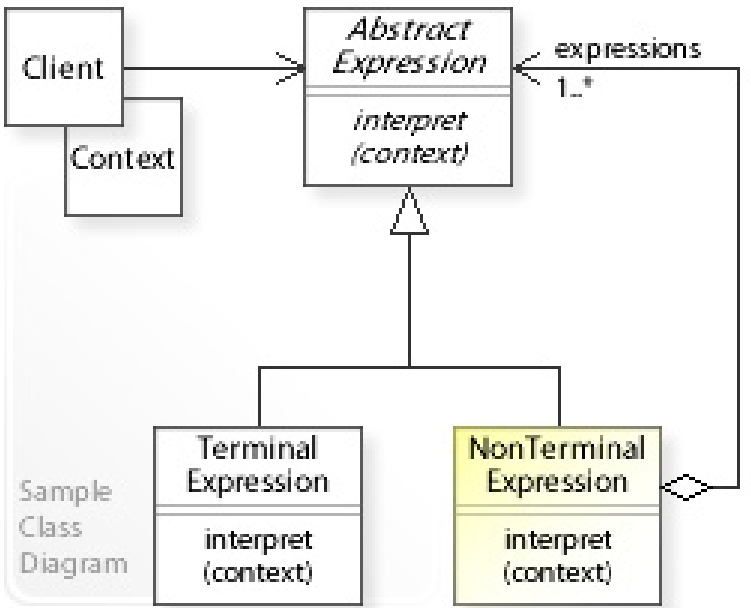
- Отделение кода исполнения команд от вызова
- Дополнительные возможности управления
- Усложнение кода (дополнительные классы)



Interpreter

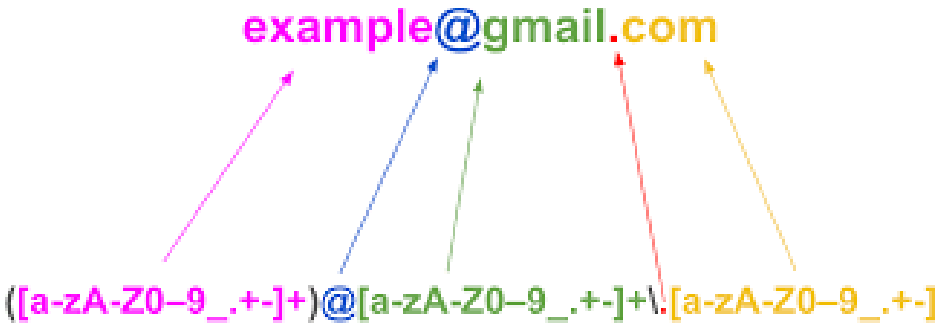
- Интерпретатор языка для управления поведением
  - Регулярные выражения
  - Форматирование строк



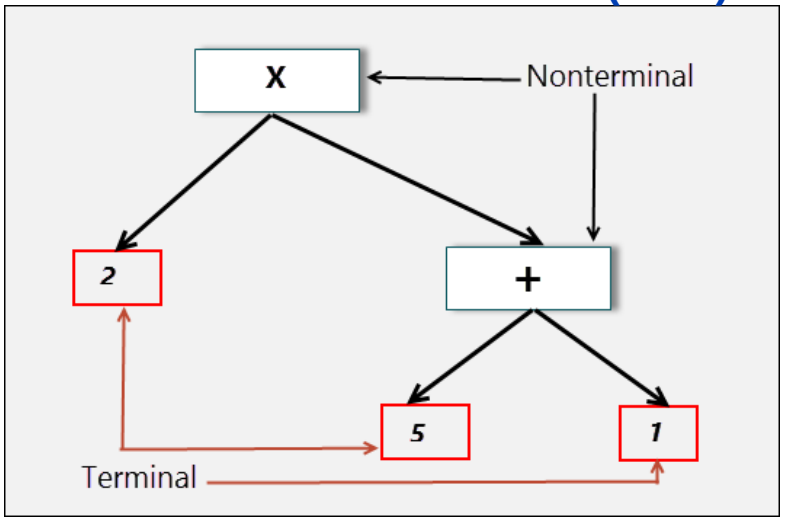


Interpreter

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

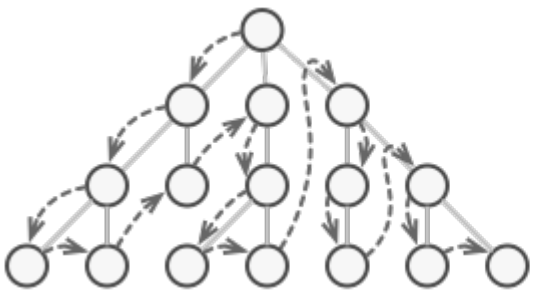


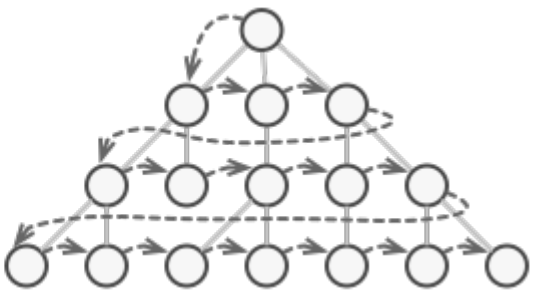
2 × (5 + 1)



Iterator

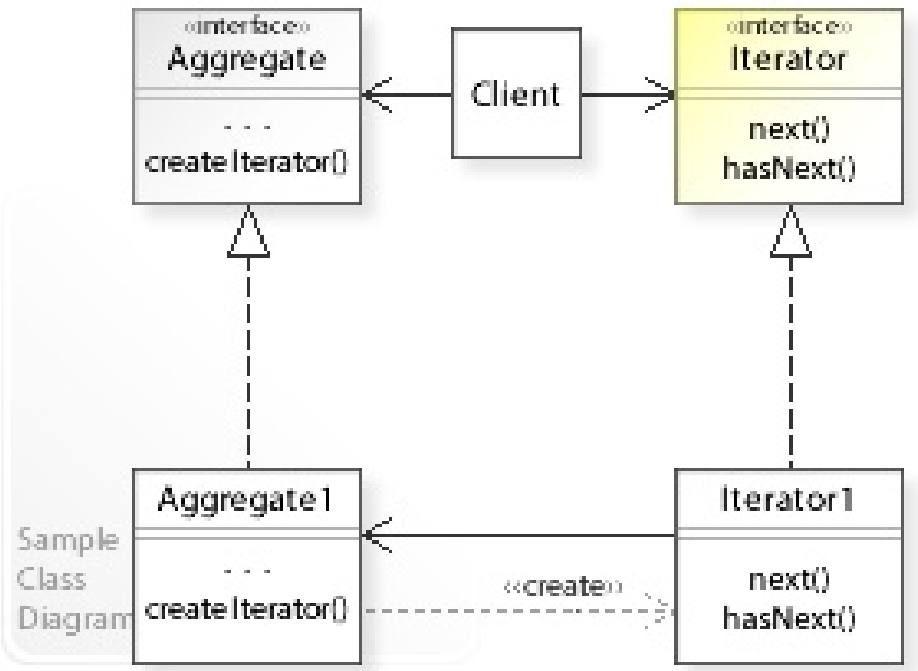
- Последовательный доступ к элементам коллекции
- Итератор знает внутреннюю структуру коллекции
- Разные способы обхода





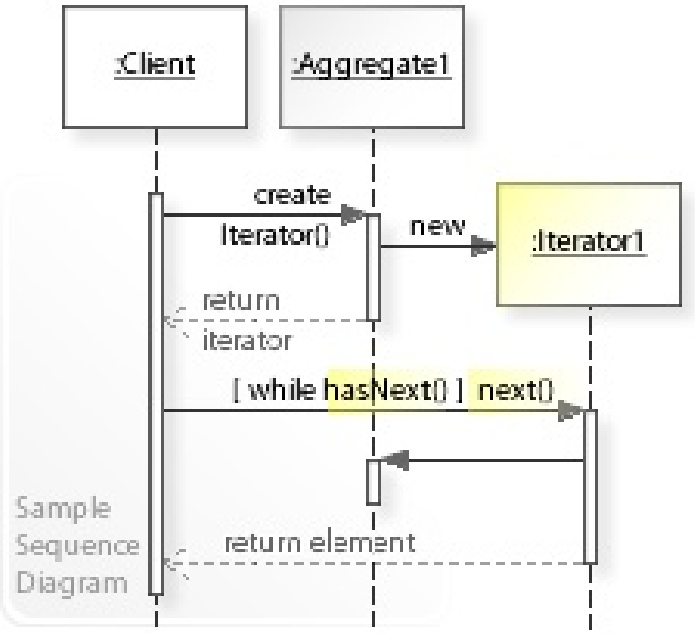
Iterator

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



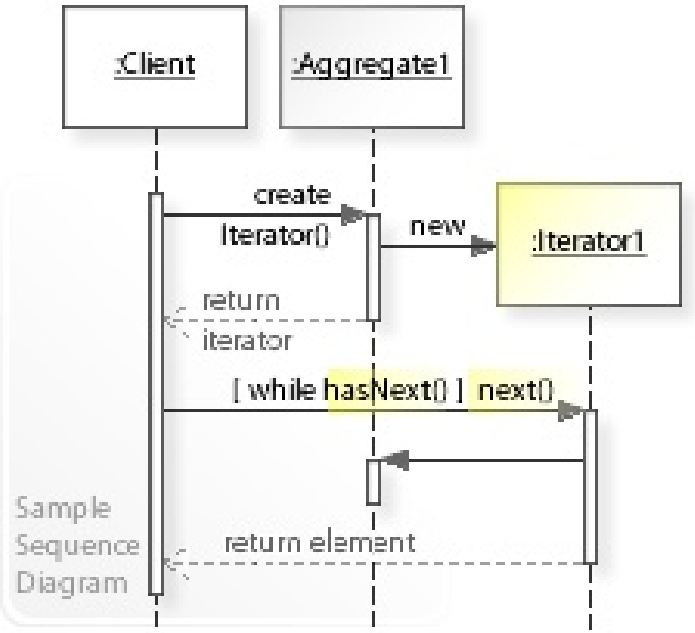
Iterator

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



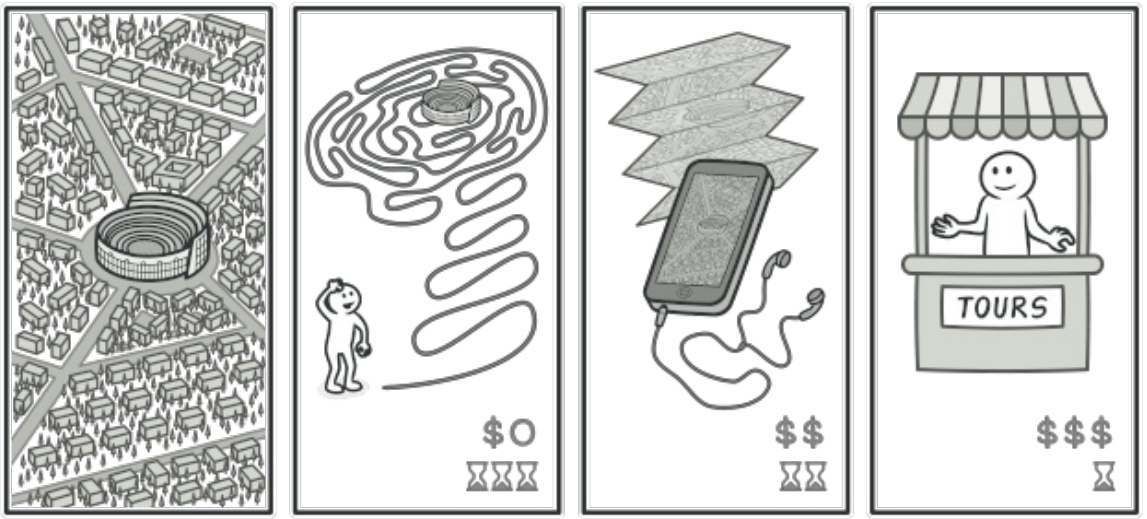
Iterator

```
class Array {
  Object[] array;
  Iterator iterator() {
    return this.new Iterator();
  }
  class Iterator {
    int current = 0;
    boolean hasNext() {
      return current < array.length;
    }
    Object next() {
      return array[current++];
    }
  }
}
```



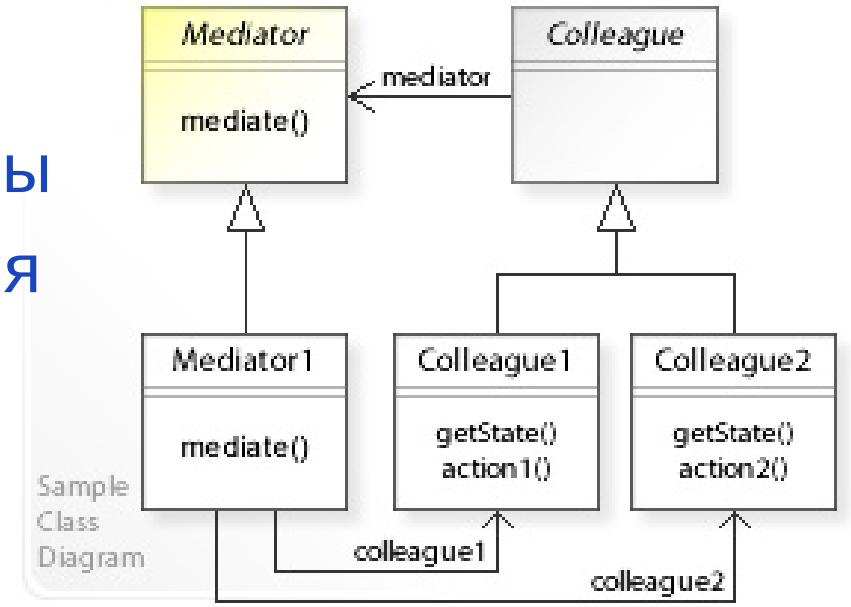
Iterator

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



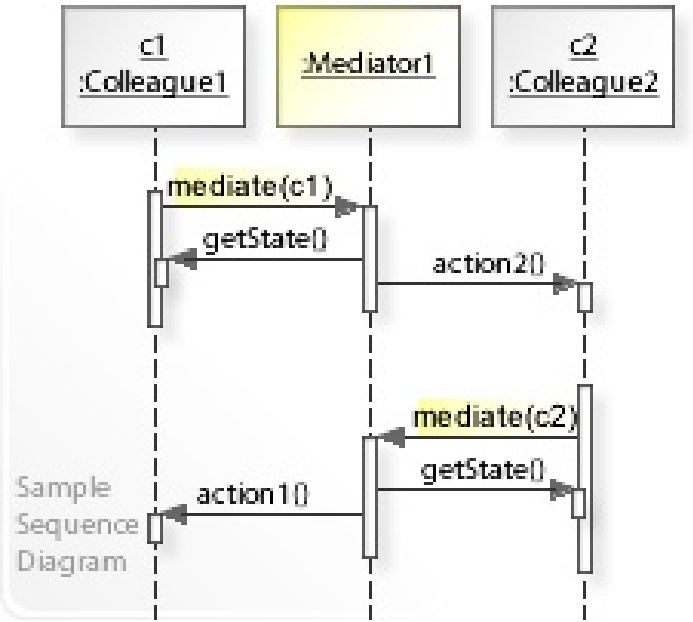
Mediator

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



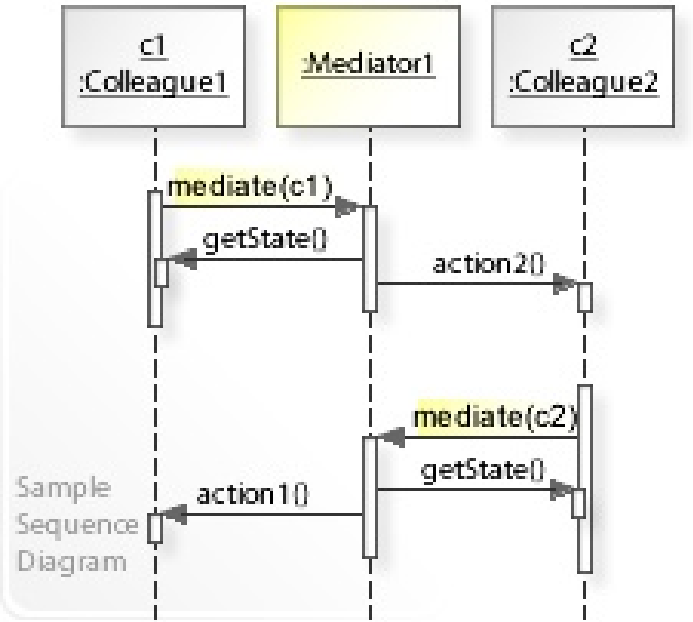
Mediator

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



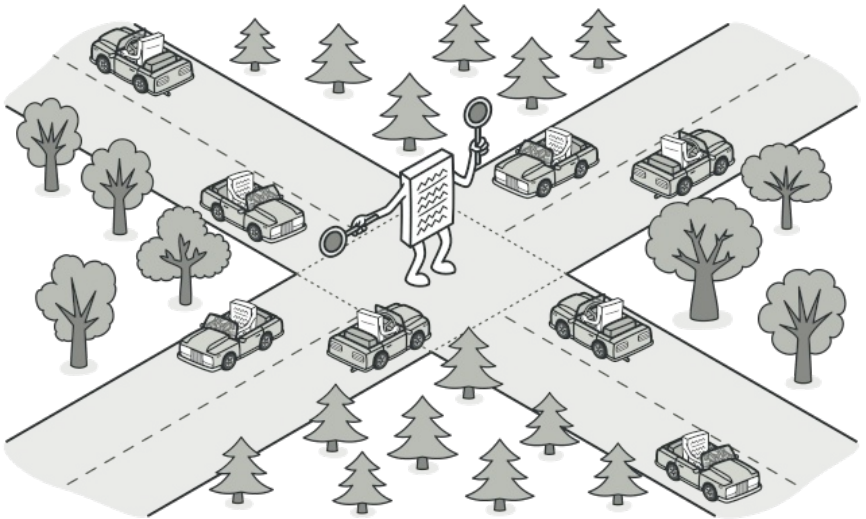
Mediator

```
class Mediator1 extends Mediator {
  Colleague c1;
  Colleague c2;
  void mediate(Colleague sender,
               String msg) {
    if (sender == c1) {
      c2.action(c1.getState(), msg);
    }
    if (sender == c2) {
      c1.action(c2.getState(), "stop");
    }
  }
}
```



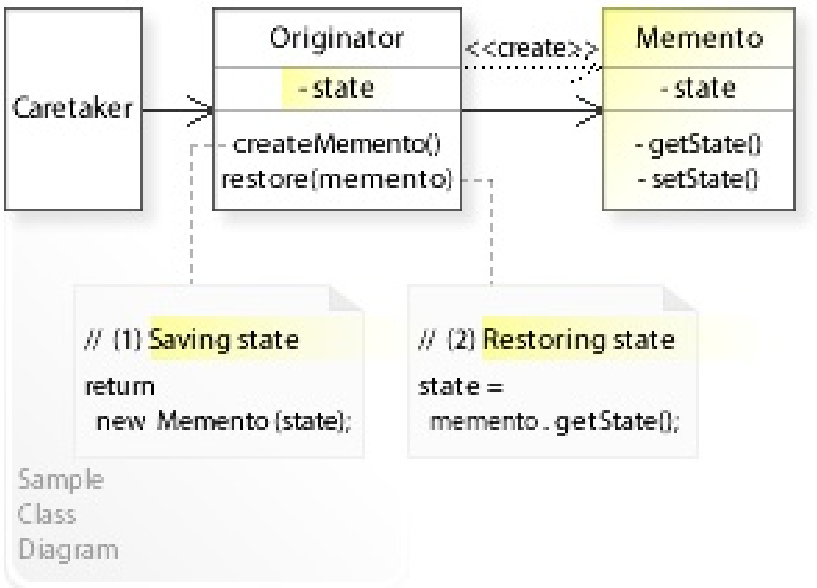
Mediator

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



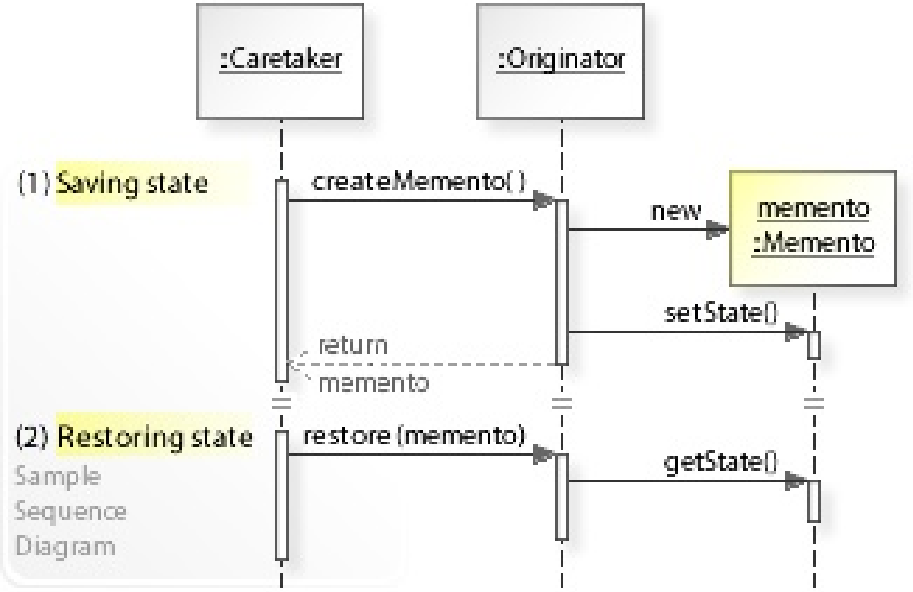
Memento

- 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



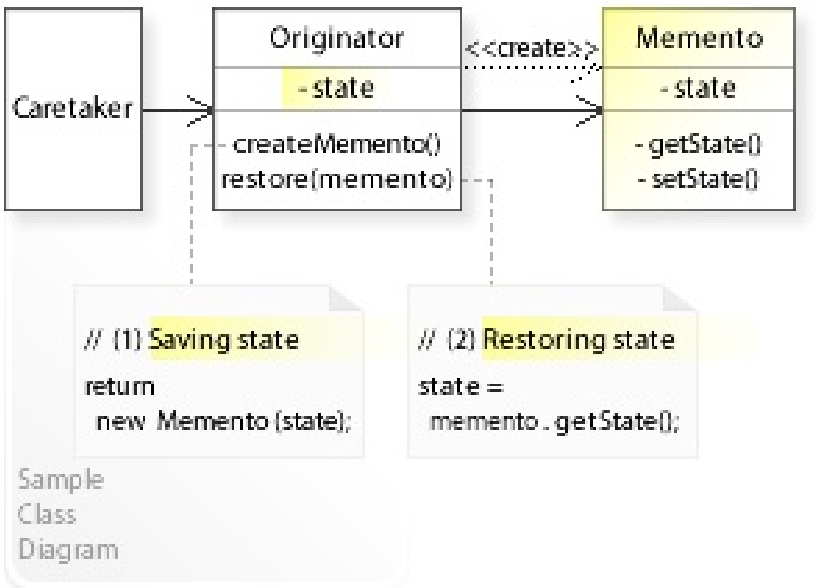
Memento

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



Memento

```
class Originator {
  private Object state;
  public Memento save() {
    return new this.Memento();
  }
  public void restore(Memento old) {
    state = old.getState();
  }
  class Memento {
    private Object s = state;
    private Object getState() {
      return s;
    }
    public String getID() { ... }
  }
}
```

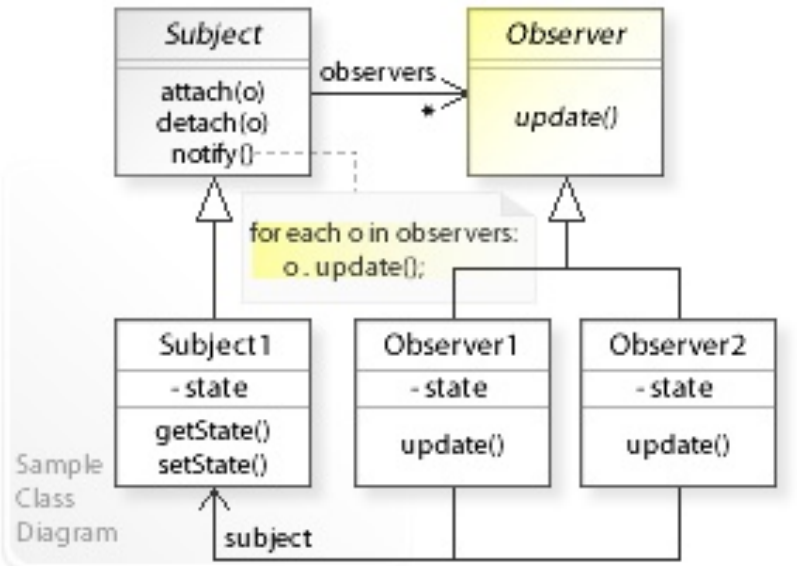


Memento

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

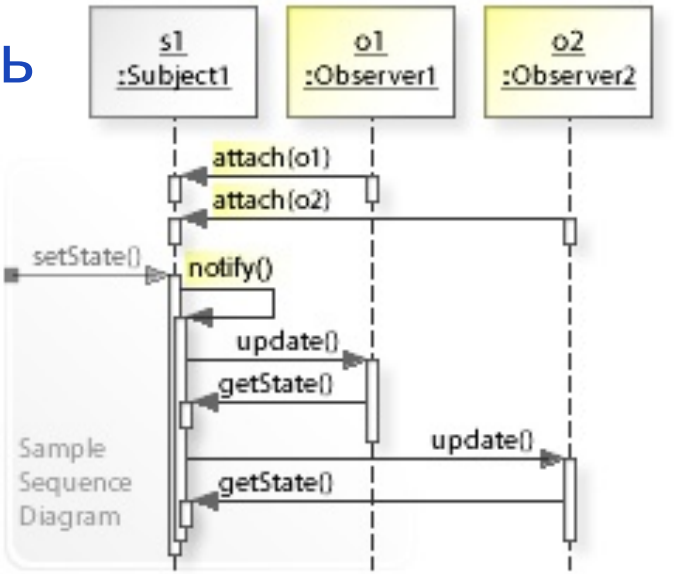
Observer

- 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



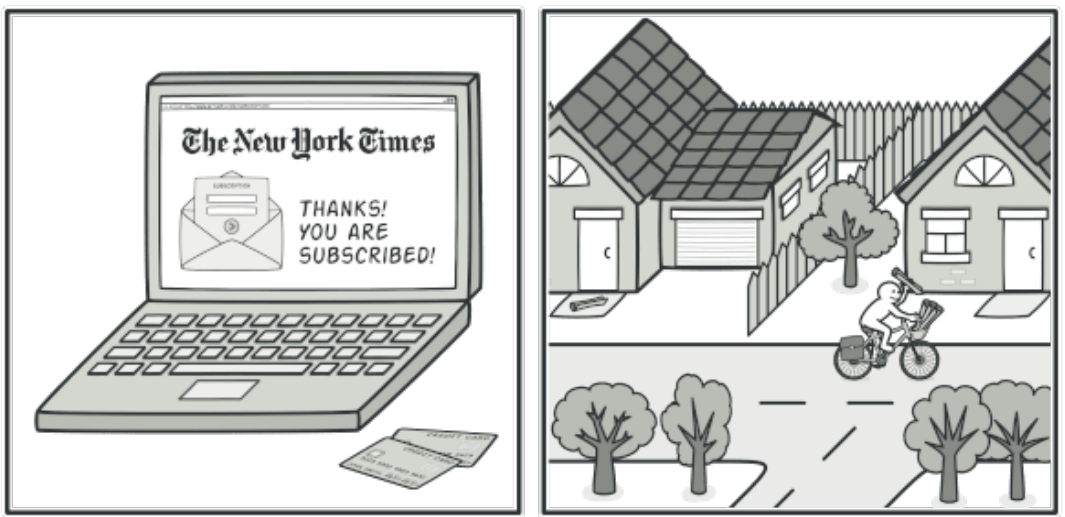
Observer

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



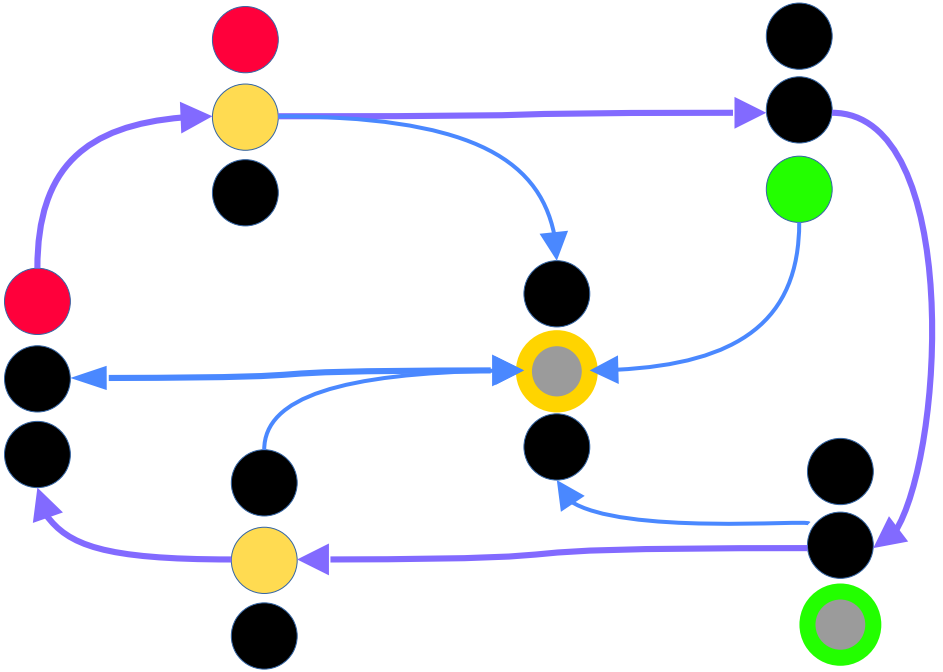
Observer

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



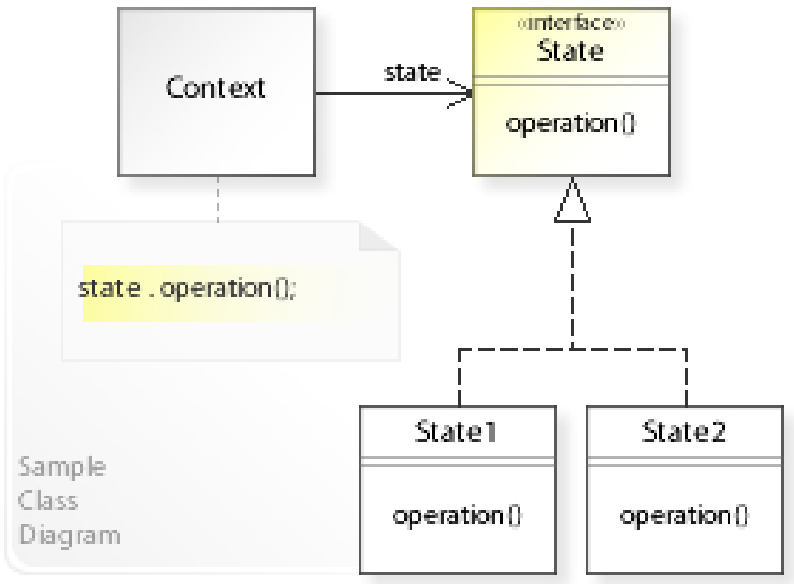
State

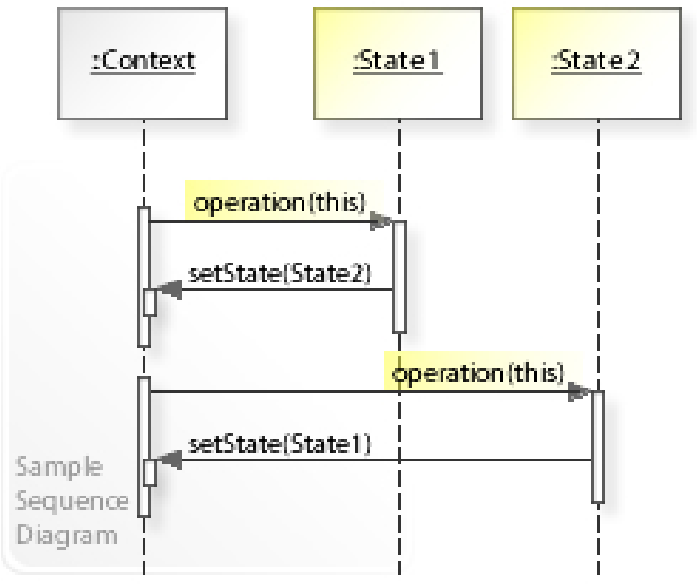
- Изменяет поведение объекта в зависимости от состояния — реализация конечного автомата



State

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





State

- Код без шаблона

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

State

- Код с шаблоном

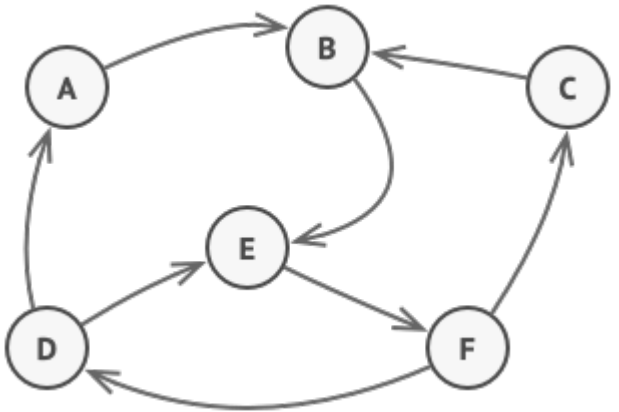
```
class Yellow implements State {
  State operation() {
    yellowOn(10);
    return new Red();
  }
}
```

```
class Red implements State {
  State operation() {
    redOn(30);
    return new RedYellow();
  }
}
```

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

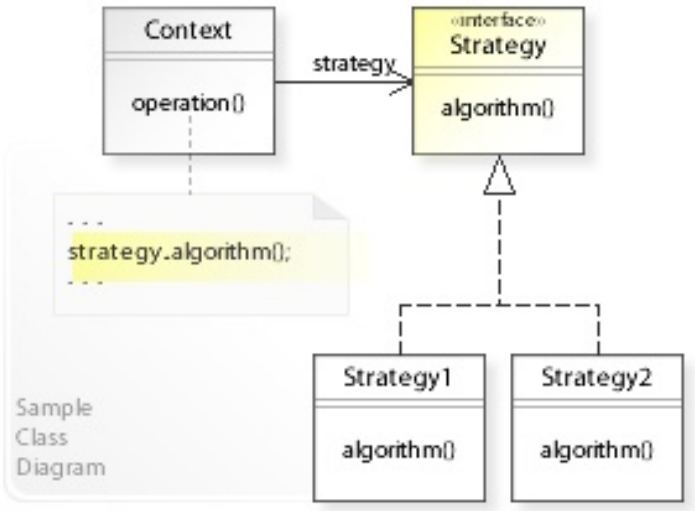
State

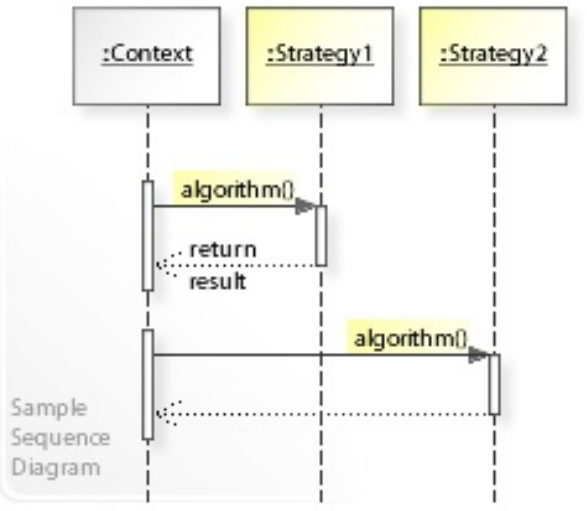
- Избавляет от множества операторов if
- Выделяет код для каждого состояния в одном месте
- В некоторых случаях слишком сложный код



Strategy

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





Strategy

```
class Taxi implements Strategy {
  void findRoute() {
    allowedStart(ROAD);
    allowedFinish(ROAD);
    setTariff(BY_KM);
    setTime(now() + 15*60);
    navigate();
}
```

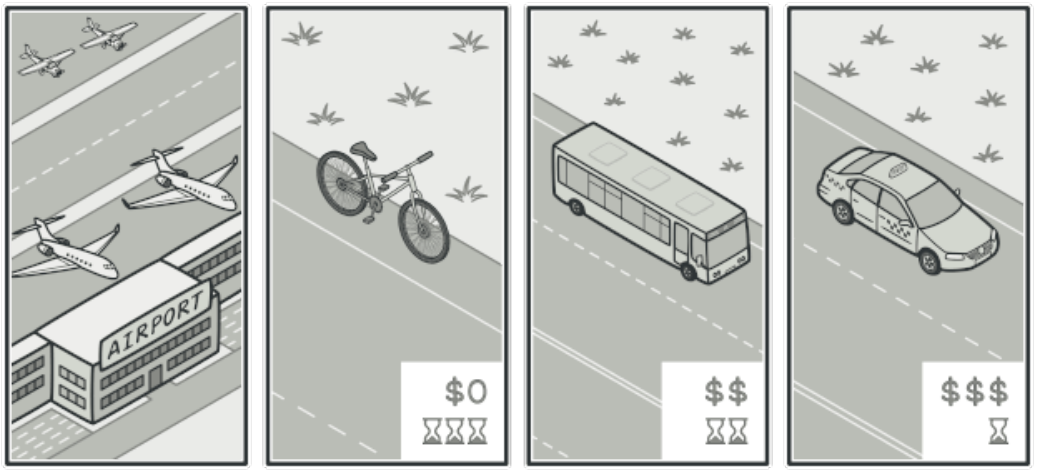
```
class Walk implements Strategy {
  void findRoute() {
    allowedStart(ANY);
    allowedFinish(ANY);
    setTariff(ZERO);
    setTime(now());
    navigate();
}
```

```
class Bus implements Strategy {
  void findRoute() {
    allowedStart(STOPS);
    allowedFinish(STOPS);
    setTariff(FIXED);
    setTime(getTimeTable());
    navigate();
}
```

```
Strategy st = getStrategy();
st.findRoute();
```

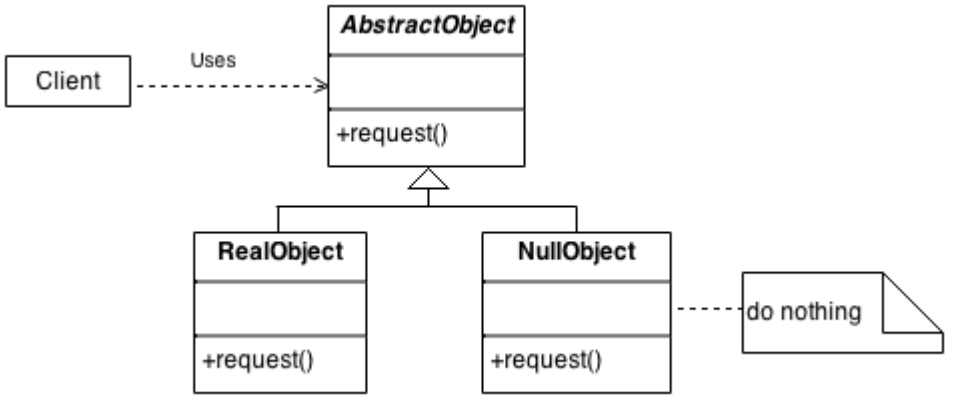
Strategy

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



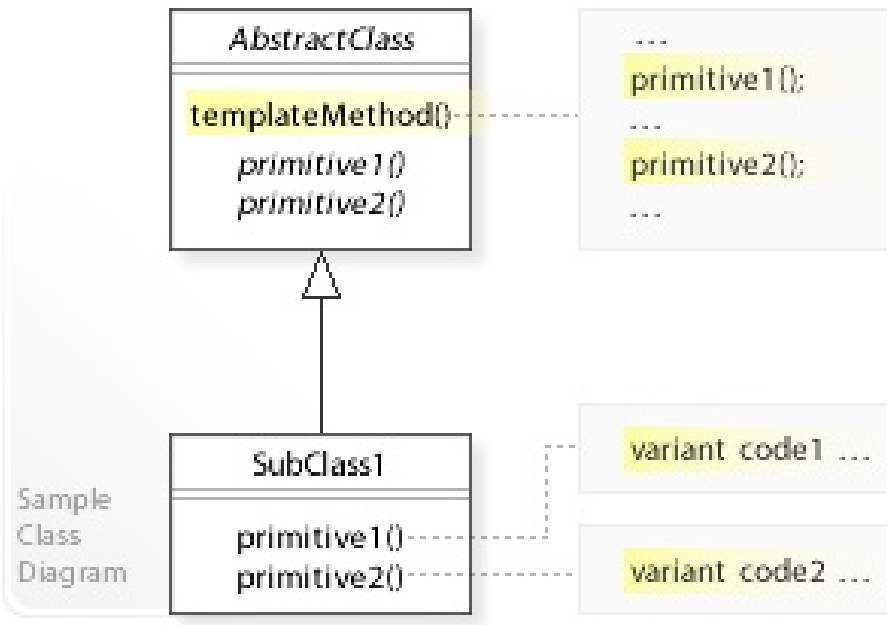
NullObject

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



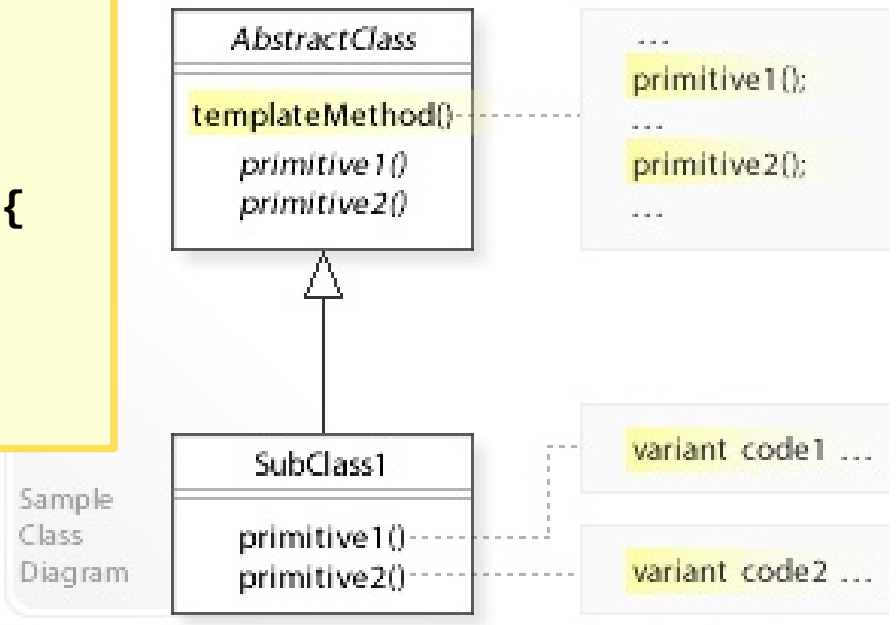
Template Method

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



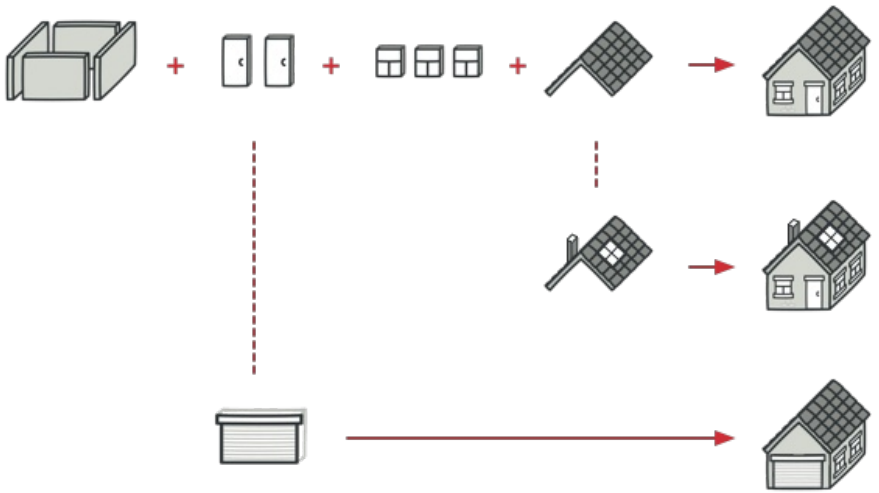
Template Method

```
class Pokemon {
  void attack() {
    calculateDamage();
    applySpecialEffects();
    fight();
  }
}

class MyPowerPokemon extends Pokemon {
  void applySpecialEffects() {
    destroyAllPokemons() { ... }
  }
}
```



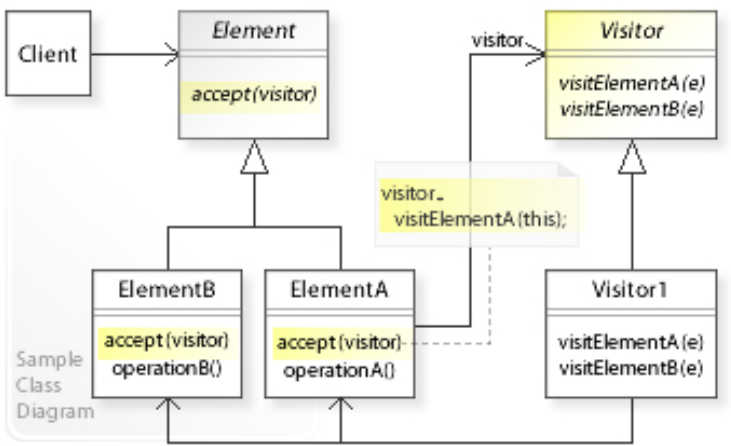
Template Method

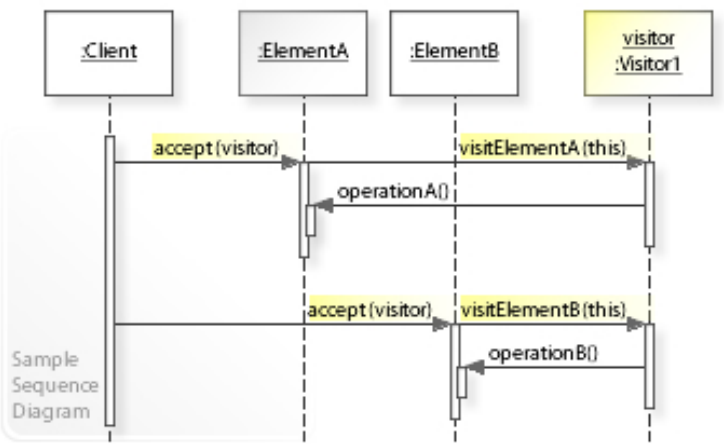
- Упрощает повторное использование кода
- Жестко задает последовательность действий
- При большом количестве действий сложно поддерживать



Visitor

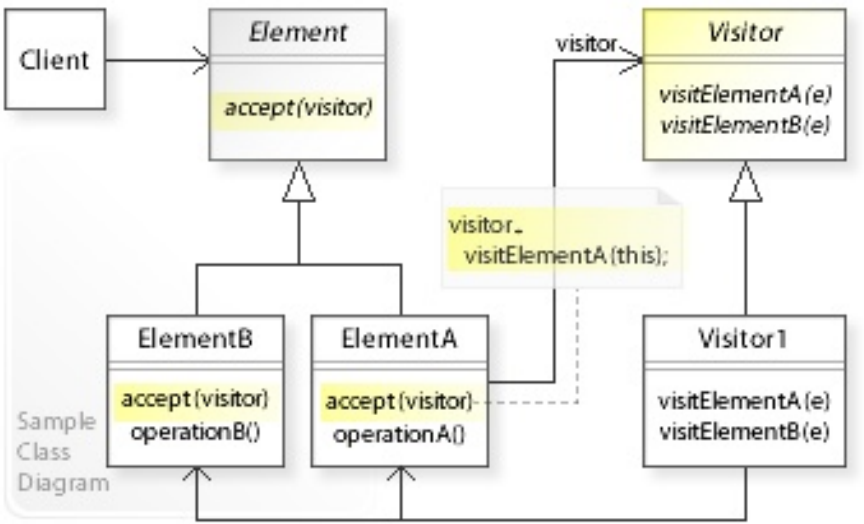
- Позволяет сгруппировать операции, выполняемые над структурой объектов
- Применяется если структура элементов более стабильна, чем набор операций





Visitor

- 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); }



Двойная диспетчеризация

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

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

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

Двойная диспетчеризация

```
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);
}
```

Двойная диспетчеризация

```
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

Двойная диспетчеризация

```
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);
}
```

Двойная диспетчеризация

```
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);
}
```

Двойная диспетчеризация

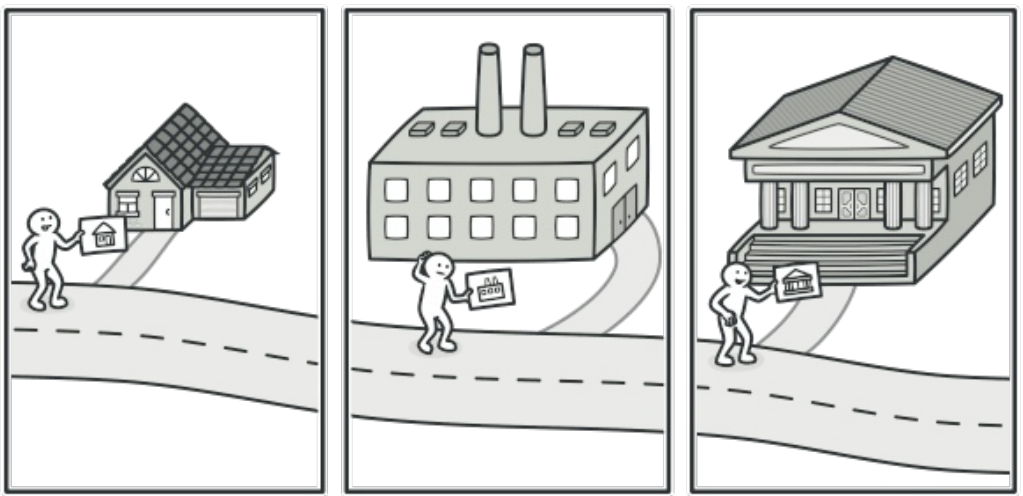
```
class Animal {
  void accept(Feeder f) {
    f.feed(this);
  }
}

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);
}
```

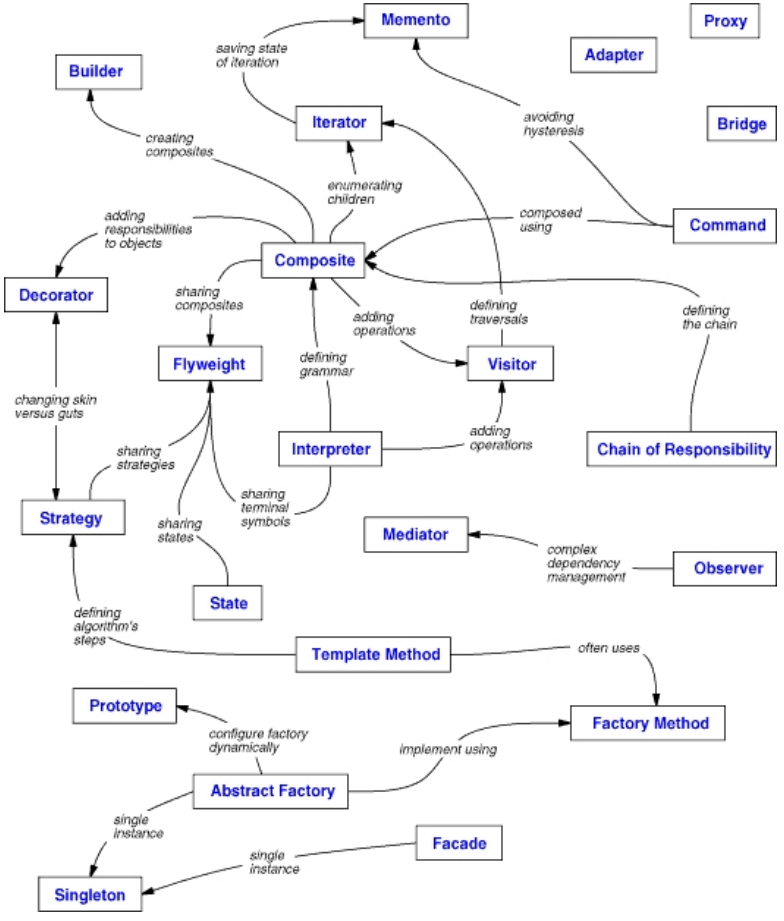
Feeding a cat

Visitor

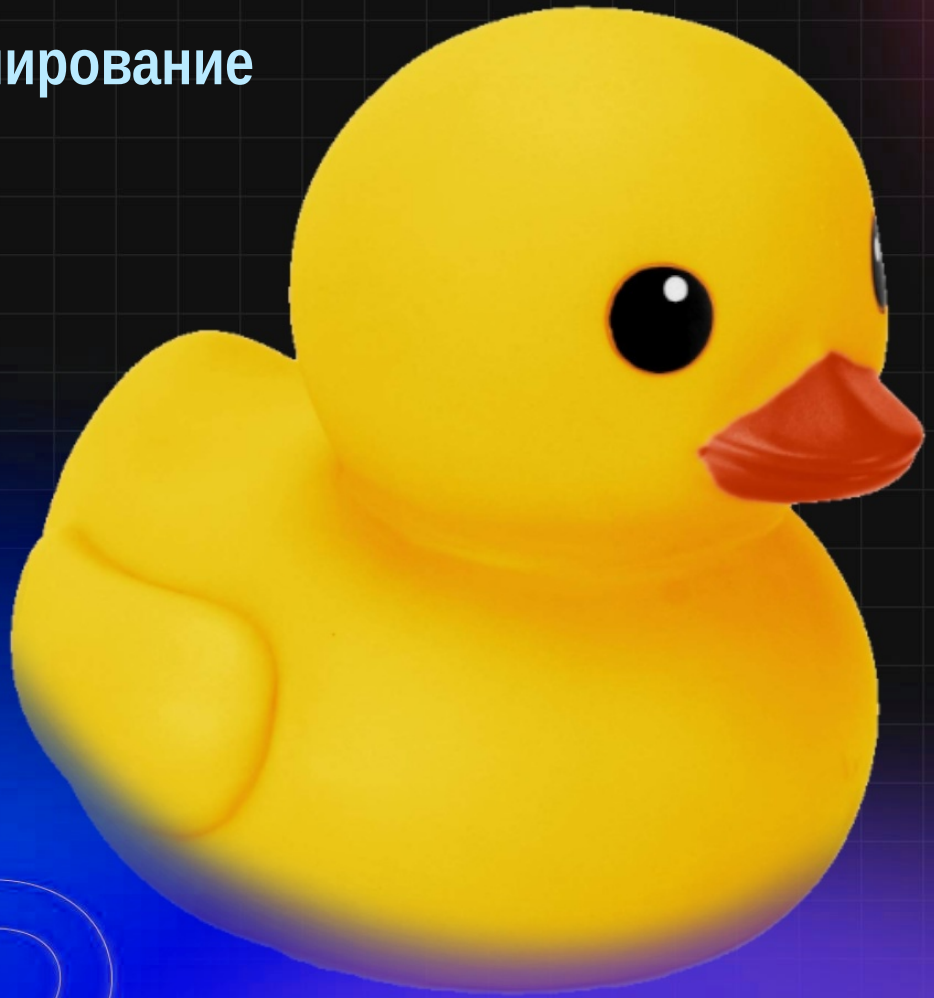
- Упрощает добавление новых операций
- Отделяет код операций от структуры элементов
- Усложняет изменение структуры



Все вместе


Memento
Proxy
saving state of iteration
Adapter
Builder
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
sharing strategies
Interpreter
adding operations
Chain of Responsibility
sharing terminal symbols
Strategy
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
configure factory dynamically
Factory Method
implement using
Abstract Factory
single instance
Facade
single instance
Singleton

Программирование
2 семестр
2025



ИТМО

Чистый код

Что такое хороший код?

ИТМО

- Метрики качества кода
- Код для человека (и обычно не для себя)
- Должно быть легко
  - читать
  - понимать
  - изменять
  - поддерживать
- Роберт Мартин. "Чистый код"

Зачем?

- Меньше затраты времени на поддержку
- Меньше потенциальных ошибок
- Командная работа
- Профессионализм

Общие принципы

- Самодокументируемость
- Информативность
- Принцип наименьшего удивления
- Единый стиль, особенно в команде

Именование

- Принципы именования:
  - осмысленные и понятные имена
  - содержательные имена
  - имена, которые проще искать
- Соглашения по именованию
  - модули/пакеты - обратное имя домена строчными буквами
  - классы/интерфейсы - существительные (CamelCase)
  - методы - глаголы (camelCase, как и переменные)
  - константы - SNAKE_CASE (заглавными буквами)

Функции и методы

- Короткие
- Единственная обязанность
- Меньше аргументов
- Меньше вложенных блоков
- Без побочных эффектов

Классы

- Небольшие
- Единственная обязанность
- Высокая связность (high cohesion) - внутри класса
- Низкая связанность (low coupling) - с другими классами

Комментарии

- Как надо:
  - Лучший комментарий - когда его нет
  - Информативные
  - Пояснения, предупреждения и TODO
  - JavaDoc
- Как НЕ надо
  - Избыточные комментарии и "Капитан Очевидность"
  - Закомментированный код

Форматирование

- Отступы и пробелы
- Единообразие
- Группировка и упорядочивание кода
- Ограничение длины строк

Обработка ошибок

- Ошибки надо обрабатывать
- Не игнорировать
- Информативные сообщения об ошибках
- Логирование
- Не передавать и не возвращать null

Тестирование

- Код должен работать
- Тесты должны выполняться
  - TDD - test-driven development