Основные принципы объектно-ориентированного программирования (ООП) включают:

**Инкапсуляция** в программировании является объединением данных и кода, работающего с этими данными, в большинстве случае это сводится к тому, чтобы не давать доступа к важным данным напрямую. Вместо этого мы создаем ограниченный набор методов, с помощью которых можно работать с нашими данными

class Dog

def initialize(name)

@name = name

end

def speak

puts "#{@name} говорит: Гав!"

end

def name=(name)

puts "Имя менять нельзя"

end

end

**Наследование**  
*Возможность создания новых классов на основе существующих, что позволяет повторно использовать код и создавать иерархии классов.*

class HeroesOfMightAndMagic3

    def play()

        puts "Герои запущены"

    end

    end

    private def  say\_hello\_player(player)

        puts "Привет, игрок #{player}"

    end

    class HeroesOfMightAndMagic3HD < HeroesOfMightAndMagic3

    def playHD(player)

    puts "Герои HD запущены"

    say\_hello\_player(player)

    end

    end

    heroesDef = HeroesOfMightAndMagic3.new

    heroesHd = HeroesOfMightAndMagic3HD.new

    heroesDef.play

    heroesHd.play

    heroesHd.playHD("Vovs")

    heroesHd.say\_hello\_player("vovs") #Ошибка

Но метод say\_hello\_player(player) нельзя вызвать вне публичных методов из-за модификаторов доступа

Еще пример

class HeroesOfMightAndMagic3

    def play()

        puts "Герои запущены"

    end

    protected def say\_hello\_player()

        puts "Привет, игрок!"

    end

end

class HeroesOfMightAndMagic3HD < HeroesOfMightAndMagic3

    def play(obj)

        puts "Герои HD запущены"

        obj.say\_hello\_player()

        say\_hello\_player()

    end

end

heroesDef = HeroesOfMightAndMagic3.new

heroesHd = HeroesOfMightAndMagic3HD.new

heroesDef.play

heroesHd.play(heroesHd) #можно

heroesHd.say\_hello\_player() #нельзя

**Полиморфизм**  
*Возможность объектов с одинаковым интерфейсом выполнять разные действия в зависимости от их типа.*

class HeroesOfMightAndMagic3

    def play()

        puts "Герои запущены"

    end

end

class HeroesOfMightAndMagic3HD < HeroesOfMightAndMagic3

    def play()

        puts "Герои HD запущены"

    end

end

**Абстракция***Выделение основных характеристик объекта, которые важны для решения задачи, и игнорирование несущественных деталей.*

**КОД:**

**Абстрактный класс Animal**

class Animal

    def initialize(name)

      @name = name

    end

    def speak

      raise NotImplementedError, "Метод 'speak' должен быть реализован в подклассе"

    end

  end

  class Dog < Animal

    def speak

      puts "#{@name} говорит: Гав!"

    end

  end

  dog = Dog.new("Бобик")

  dog.speak

  animal = Animal.new("52")

  animal.speak

Наследование и полиморфизм еще

class Animal

    def initialize(name)

      @name = name

    end

    def speak

      puts "52 #{@name}"

    end

end

class Dog < Animal

    def speak

        puts "53 #{@name}"

      end

end

  dog = Dog.new("Бобик")

  dog.speak

  animal = Animal.new("52")

  animal.speak

  [animal, dog].each do |a| a.speak end

Пример нарушения принципов ООП

Если нарушить принцип инкапсуляции, например, сделать переменную @name публичной:

class Animal

attr\_accessor :name # Нарушение инкапсуляции

def initialize(name)

@name = name

end

end

dog = Animal.new("Бобик")

dog.name = "Неправильное имя" # Прямое изменение данных

puts dog.name # Неправильное имя

Этот код нарушает принцип инкапсуляции, так как данные объекта могут быть изменены напрямую, без контроля со стороны объекта.

class Animal

    def initialize(name)

      @name = name

    end

    def speak

      puts "52"

    end

end

class Dog

    #дублирование

    def initialize(name)

        @name = name

    end

    def speak

      puts "#{@name} говорит: Гав!"

    end

end

  dog = Dog.new("Бобик")

  dog.speak

  animal = Animal.new("52")

  animal.speak

  [animal, dog].each do |a| a.speak end

# Какие отношения между классами могут быть? Опишите, что они из себя представляют в предметной области, в коде и на диаграмме классов. Покажите пример кода с реализацией этих отношений.

# Наследование

Наследование является базовым принципом ООП и позволяет одному классу (наследнику) унаследовать функционал другого класса (родительского). Нередко отношения наследования еще называют генерализацией или обобщением. Наследование определяет отношение **IS A**, то есть "является"

Пример в предметной области:  
Класс Animal (животное) является родительским классом для классов Dog (собака) и Cat (кошка). Собака и кошка — это животные.

Пример кода:

class Animal

    def speak

      puts "Some generic sound"

    end

  end

  class Dog < Animal

    def speak

      puts "Woof!"

    end

  end

  class Cat < Animal

  end

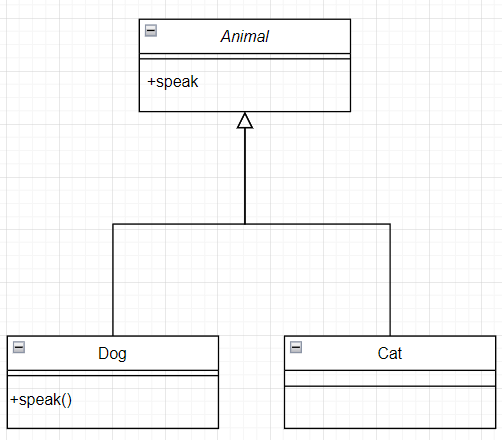
  dog = Dog.new

  dog.speak # Woof!

  cat = Cat.new

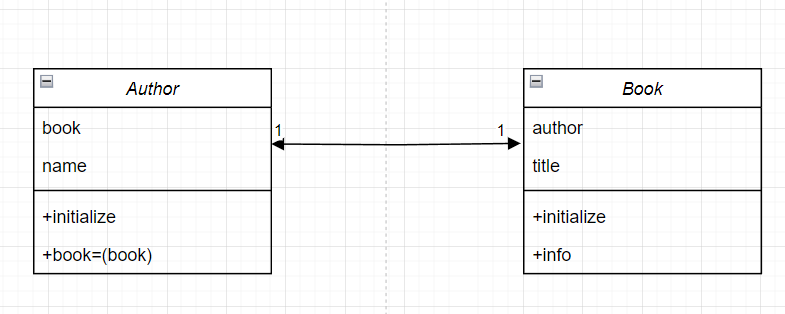
  cat.speak # Some generic sound

Диаграмма классов:



2. Ассоциация (Association)

Описание:

Внутри одного класса в полях находится экземпляр другого класса  
Ассоциация — это отношение "использует" (uses). Один класс использует другой класс, но они не зависят друг от друга жизненным циклом. 

Пример в предметной области:  
class Author

    attr\_reader :name, :book

    def initialize(name)

      @name = name

    end

    def book=(book)

      @book = book

      book.author = self

    end

  end

  class Book

    attr\_accessor :title, :author

    def initialize(title)

      @title = title

      @author = nil

    end

    def info

      if author

        "#{title} (Автор: #{author.name})"

      else

        "#{title} (Автор неизвестен)"

      end

    end

  end

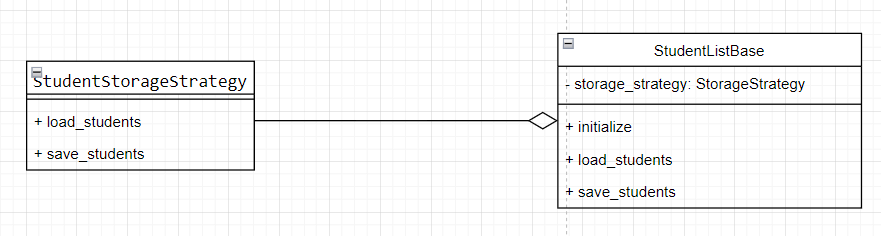
  author = Author.new("Лев Толстой")

  book1 = Book.new("Война и мир")

  author.book=book1

  puts author.book.info

3. Агрегация (Aggregation)



class StudentsListBase

    #Мы не создаем внутри класса StudentStorageStrategy -> агрегация(передаем из вне)

    attr\_writer :storage\_strategy

#Передаем готовый объект

    def initialize(storage\_strategy)

      @storage\_strategy = storage\_strategy

    end

    #Делегация

    def load\_students(file\_path)

      @students = @storage\_strategy.load\_students(file\_path)

    end

    #Делегация

    def save\_students(file\_path)

      @storage\_strategy.save\_students(file\_path, @students)

    end

end

class StudentStorageStrategy

    def load\_students(file\_path)

      raise NotImplementedError, 'This method should be overridden in subclasses'

    end

    def save\_students(file\_path, students)

      raise NotImplementedError, 'This method should be overridden in subclasses'

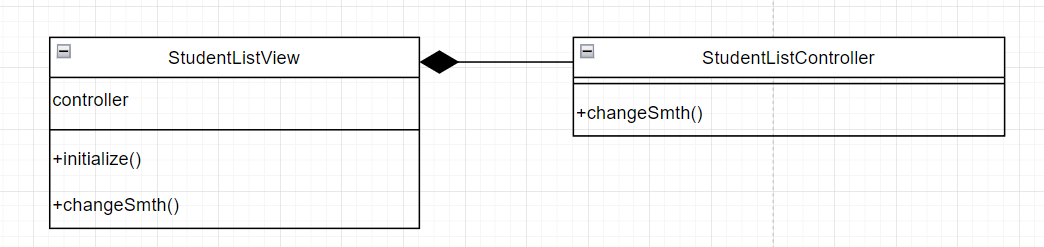
    end

end

Описание:  
Агрегация — это частный случай ассоциации, где один класс является частью другого, но может существовать независимо. Это отношение "имеет" (has-a).

4. Композиция (Composition)

Описание:  
Композиция — это более строгая форма агрегации, где один класс является частью другого и не может существовать независимо. Это отношение "содержит" (contains-a).



class StudentListView

    def initialize()

        @controller = StudentListController.new()

    end

    def changeSmth()

        @controller.changeSmth()

    end

end

class StudentListController

    def changeSmth()

        puts "Im doing smth"

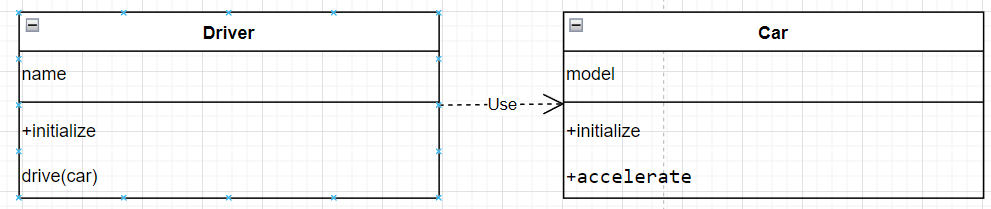
    end

end

Пример в предметной области:

5. Зависимость (Dependency)

Описание:  
Зависимость — это отношение, при котором один класс использует другой класс временно, но они не связаны напрямую. Это отношение "использует" (uses).

Пример в предметной области:  


Пример кода:

class Driver

    attr\_reader :name

    def initialize(name)

      @name = name

    end

    def drive(car)

      puts "#{name} управляет автомобилем #{car.model}"

      car.accelerate

    end

  end

  class Car

    attr\_reader :model

     def initialize(model)

      @model = model

    end

    def accelerate

      puts "#{model} ускоряется!"

    end

  end

  driver = Driver.new("Иван")

  car = Car.new("Toyota Camry")

  driver.drive(car)

## Паттерны проектирования, использующие ассоциацию

Наблюдатель (Observer)  
Паттерн "Наблюдатель" использует ассоциацию для связи между объектами. Один объект (субъект) уведомляет другие объекты (наблюдатели) об изменениях в своем состоянии.

ruby

Copy

class Subject

attr\_accessor :observers

def initialize

@observers = []

end

def add\_observer(observer)

@observers << observer

end

def notify\_observers

@observers.each { |observer| observer.update(self) }

end

end

class Observer

def update(subject)

puts "Subject state has changed!"

end

end

subject = Subject.new

observer = Observer.new

subject.add\_observer(observer)

subject.notify\_observers

# Subject state has changed!

Посетитель (Visitor)  
Паттерн "Посетитель" использует ассоциацию для добавления новых операций к объектам без изменения их классов.

class Element

def accept(visitor)

raise NotImplementedError

end

end

class ConcreteElement < Element

def accept(visitor)

visitor.visit(self)

end

end

class Visitor

def visit(element)

puts "Visiting #{element.class}"

end

end

element = ConcreteElement.new

visitor = Visitor.new

element.accept(visitor)

# Visiting ConcreteElement

Стратегия (Strategy)  
Паттерн "Стратегия" использует ассоциацию для выбора алгоритма во время выполнения.

ruby

Copy

class Context

attr\_accessor :strategy

def initialize(strategy)

@strategy = strategy

end

def execute\_strategy

@strategy.execute

end

end

class StrategyA

def execute

puts "Executing Strategy A"

end

end

class StrategyB

def execute

puts "Executing Strategy B"

end

end

context = Context.new(StrategyA.new)

context.execute\_strategy

# Executing Strategy A

context.strategy = StrategyB.new

context.execute\_strategy

# Executing Strategy B

Принцип "Предпочитайте делегацию наследованию"

Этот принцип гласит, что вместо создания иерархии классов через наследование лучше использовать делегирование — передачу части ответственности другому классу. Делегирование позволяет создавать более гибкие и поддерживаемые системы, избегая жестких связей, которые часто возникают при наследовании.

Почему делегирование лучше наследования?

Гибкость:  
Делегирование позволяет динамически изменять поведение объекта, передавая ему разные объекты для выполнения задач. Наследование же жестко связывает классы, что затрудняет изменение поведения.

Избегание проблем с иерархией:  
Наследование может привести к сложным иерархиям классов, которые трудно поддерживать (например, проблема "алмаза смерти" в множественном наследовании).

Принцип единственной ответственности:  
Делегирование помогает соблюдать принцип единственной ответственности, так как каждый класс отвечает только за свою часть функциональности.

Примеры кода

Пример 1: Наследование

ruby

Copy

class Animal

def speak

puts "Some generic sound"

end

end

class Dog < Animal

def speak

puts "Woof!"

end

end

dog = Dog.new

dog.speak # Woof!

Проблема:  
Если нам нужно добавить новое поведение (например, способность летать), мы вынуждены либо добавлять методы в базовый класс Animal, либо создавать новые подклассы. Это может привести к раздуванию иерархии.

Пример 2: Делегирование

ruby

Copy

class SoundBehavior

def speak

raise NotImplementedError

end

end

class WoofBehavior < SoundBehavior

def speak

puts "Woof!"

end

end

class MeowBehavior < SoundBehavior

def speak

puts "Meow!"

end

end

class Animal

attr\_accessor :sound\_behavior

def initialize(sound\_behavior)

@sound\_behavior = sound\_behavior

end

def speak

@sound\_behavior.speak

end

end

dog = Animal.new(WoofBehavior.new)

dog.speak # Woof!

cat = Animal.new(MeowBehavior.new)

cat.speak # Meow!

Преимущество:  
Мы можем легко изменить поведение объекта, передав ему другой объект SoundBehavior. Например, если собака научится мяукать:

ruby

Copy

dog.sound\_behavior = MeowBehavior.new

dog.speak # Meow!

Когда использовать наследование, а когда делегирование?

Наследование:

Используйте, когда между классами есть четкое отношение "является" (is-a).

Когда подкласс является специализацией родительского класса.

Пример: Dog является Animal.

Делегирование:

Используйте, когда нужно разделить ответственность между классами.

Когда поведение может изменяться динамически.

Пример: Animal делегирует воспроизведение звука объекту SoundBehavior.

Паттерны, иллюстрирующие делегирование

Стратегия (Strategy)  
Паттерн "Стратегия" позволяет динамически изменять поведение объекта, передавая ему разные стратегии.

ruby

Copy

class Strategy

def execute

raise NotImplementedError

end

end

class StrategyA < Strategy

def execute

puts "Executing Strategy A"

end

end

class StrategyB < Strategy

def execute

puts "Executing Strategy B"

end

end

class Context

attr\_accessor :strategy

def initialize(strategy)

@strategy = strategy

end

def execute\_strategy

@strategy.execute

end

end

context = Context.new(StrategyA.new)

context.execute\_strategy # Executing Strategy A

context.strategy = StrategyB.new

context.execute\_strategy # Executing Strategy B

Декоратор (Decorator)  
Паттерн "Декоратор" добавляет новое поведение объекту, не изменяя его класс.

ruby

Copy

class Component

def operation

raise NotImplementedError

end

end

class ConcreteComponent < Component

def operation

puts "ConcreteComponent operation"

end

end

class Decorator < Component

attr\_accessor :component

def initialize(component)

@component = component

end

def operation

@component.operation

end

end

class ConcreteDecoratorA < Decorator

def operation

super

puts "ConcreteDecoratorA operation"

end

end

component = ConcreteComponent.new

decorator = ConcreteDecoratorA.new(component)

decorator.operation

# ConcreteComponent operation

# ConcreteDecoratorA operation

Компоновщик (Composite)  
Паттерн "Компоновщик" использует делегирование для работы с древовидными структурами.

ruby

Copy

class Component

def operation

raise NotImplementedError

end

end

class Leaf < Component

def operation

puts "Leaf operation"

end

end

class Composite < Component

def initialize

@children = []

end

def add(child)

@children << child

end

def operation

@children.each { |child| child.operation }

end

end

composite = Composite.new

composite.add(Leaf.new)

composite.add(Leaf.new)

composite.operation

# Leaf operation

# Leaf operation

5. Способы создания экземпляров классов

Создание экземпляров классов — это фундаментальная часть объектно-ориентированного программирования. Существует несколько подходов к созданию объектов, и выбор подхода зависит от контекста и требований проекта. Рассмотрим основные способы:

1. Прямое создание через конструктор

Самый простой и распространенный способ — использование конструктора класса.

Пример:

ruby

Copy

class Product

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

end

product = Product.new("Laptop", 1000)

puts product.name # Laptop

puts product.price # 1000

Проблема:  
Если у класса много параметров, конструктор может стать слишком сложным. Это особенно актуально, если некоторые параметры необязательные.

2. Использование фабричных методов

Фабричные методы — это методы, которые создают и возвращают экземпляры класса. Они могут быть статическими или принадлежать объекту.

Пример:

ruby

Copy

class Product

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

# Фабричный метод

def self.create\_with\_discount(name, price, discount)

new(name, price - discount)

end

end

product = Product.create\_with\_discount("Laptop", 1000, 100)

puts product.price # 900

Преимущество:  
Фабричные методы инкапсулируют логику создания объектов, делая код более читаемым и поддерживаемым.

3. Использование паттерна "Строитель" (Builder)

Паттерн "Строитель" позволяет создавать сложные объекты пошагово.

Пример:

ruby

Copy

class Product

attr\_accessor :name, :price, :category

def initialize

@name = nil

@price = nil

@category = nil

end

end

class ProductBuilder

def initialize

@product = Product.new

end

def set\_name(name)

@product.name = name

self

end

def set\_price(price)

@product.price = price

self

end

def set\_category(category)

@product.category = category

self

end

def build

@product

end

end

builder = ProductBuilder.new

product = builder.set\_name("Laptop").set\_price(1000).set\_category("Electronics").build

puts product.name # Laptop

puts product.price # 1000

puts product.category # Electronics

Преимущество:  
Паттерн "Строитель" полезен, когда объект имеет много параметров, и некоторые из них необязательные.

4. Использование паттерна "Фабрика" (Factory)

Паттерн "Фабрика" инкапсулирует создание объектов в отдельном классе.

Пример:

ruby

Copy

class Product

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

end

class ProductFactory

def self.create\_product(type)

case type

when :laptop

Product.new("Laptop", 1000)

when :phone

Product.new("Phone", 500)

else

raise "Unknown product type"

end

end

end

product = ProductFactory.create\_product(:laptop)

puts product.name # Laptop

puts product.price # 1000

Преимущество:  
Фабрика позволяет централизовать логику создания объектов, что упрощает поддержку и расширение.

5. Использование паттерна "Одиночка" (Singleton)

Паттерн "Одиночка" гарантирует, что у класса будет только один экземпляр.

Пример:

ruby

Copy

require 'singleton'

class Logger

include Singleton

def log(message)

puts "Log: #{message}"

end

end

logger = Logger.instance

logger.log("Test message") # Log: Test message

Преимущество:  
Паттерн "Одиночка" полезен, когда нужно гарантировать, что объект существует в единственном экземпляре (например, логгер, конфигурация).

6. Использование Dependency Injection (Внедрение зависимостей)

Внедрение зависимостей — это передача зависимостей объекта извне, а не создание их внутри класса.

Пример:

ruby

Copy

class ProductService

attr\_accessor :logger

def initialize(logger)

@logger = logger

end

def create\_product(name, price)

@logger.log("Creating product: #{name}")

Product.new(name, price)

end

end

class Logger

def log(message)

puts "Log: #{message}"

end

end

logger = Logger.new

service = ProductService.new(logger)

product = service.create\_product("Laptop", 1000)

# Log: Creating product: Laptop

Преимущество:  
Внедрение зависимостей делает код более гибким и тестируемым.

Некачественный подход

Пример некачественного кода:

ruby

Copy

class Product

attr\_accessor :name, :price

def initialize(name, price, category = nil, discount = nil, warranty = nil)

@name = name

@price = price

@category = category

@discount = discount

@warranty = warranty

end

end

product = Product.new("Laptop", 1000, "Electronics", 100, "2 years")

Проблема:

Конструктор перегружен параметрами.

Код сложно читать и поддерживать.

Невозможно гибко изменять параметры.

Качественный подход

Пример качественного кода с использованием паттерна "Строитель":

ruby

Copy

class Product

attr\_accessor :name, :price, :category, :discount, :warranty

end

class ProductBuilder

def initialize

@product = Product.new

end

def set\_name(name)

@product.name = name

self

end

def set\_price(price)

@product.price = price

self

end

def set\_category(category)

@product.category = category

self

end

def set\_discount(discount)

@product.discount = discount

self

end

def set\_warranty(warranty)

@product.warranty = warranty

self

end

def build

@product

end

end

builder = ProductBuilder.new

product = builder.set\_name("Laptop")

.set\_price(1000)

.set\_category("Electronics")

.set\_discount(100)

.set\_warranty("2 years")

.build

puts product.name # Laptop

puts product.price # 1000

puts product.category # Electronics

puts product.discount # 100

puts product.warranty # 2 years

Преимущество:

Код читаем и поддерживаем.

Параметры можно задавать гибко.

Легко добавлять новые параметры.

Паттерны проектирования, связанные с созданием объектов

Фабричный метод (Factory Method)  
Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.

Абстрактная фабрика (Abstract Factory)  
Предоставляет интерфейс для создания семейств связанных объектов.

Строитель (Builder)  
Позволяет создавать сложные объекты пошагово.

Одиночка (Singleton)  
Гарантирует, что у класса будет только один экземпляр.

Прототип (Prototype)  
Позволяет копировать существующие объекты без создания новых через конструктор.

ООП структура Ruby

Ruby — это полностью объектно-ориентированный язык программирования. В Ruby всё является объектом, включая примитивные типы данных (числа, строки) и даже классы. Основные элементы ООП в Ruby:

Классы (Classes)  
Классы — это шаблоны для создания объектов. Они определяют свойства (атрибуты) и поведение (методы) объектов.

Объекты (Objects)  
Объекты — это экземпляры классов. Каждый объект имеет состояние (значения атрибутов) и поведение (методы).

Наследование (Inheritance)  
Ruby поддерживает единичное наследование. Класс может наследовать методы и атрибуты от одного родительского класса.

Модули (Modules)  
Модули используются для группировки методов, констант и классов. Они также позволяют реализовать множественное наследование через примешивание (mixins).

Инкапсуляция (Encapsulation)  
Ruby предоставляет модификаторы доступа (public, private, protected) для управления видимостью методов и атрибутов.

Полиморфизм (Polymorphism)  
Ruby поддерживает полиморфизм через переопределение методов и утиную типизацию.

Основные методы, которые принято переопределять

В Ruby есть несколько методов, которые часто переопределяются в пользовательских классах для изменения поведения объектов. Рассмотрим их:

1. initialize

Метод initialize — это конструктор объекта. Он вызывается при создании нового экземпляра класса.

Пример:

ruby

Copy

class Product

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

end

product = Product.new("Laptop", 1000)

puts product.name # Laptop

puts product.price # 1000

Когда переопределять:  
Когда нужно инициализировать объект с определенными значениями или выполнить дополнительные действия при создании объекта.

2. to\_s

Метод to\_s возвращает строковое представление объекта. По умолчанию он возвращает имя класса и идентификатор объекта.

Пример:

ruby

Copy

class Product

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

def to\_s

"Product: #{@name}, Price: #{@price}"

end

end

product = Product.new("Laptop", 1000)

puts product # Product: Laptop, Price: 1000

Когда переопределять:  
Когда нужно предоставить удобочитаемое строковое представление объекта.

3. inspect

Метод inspect возвращает строку, которая используется для отладки. По умолчанию он возвращает информацию о классе и атрибутах объекта.

Пример:

ruby

Copy

class Product

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

def inspect

"#<Product name: #{@name}, price: #{@price}>"

end

end

product = Product.new("Laptop", 1000)

puts product.inspect # #<Product name: Laptop, price: 1000>

Когда переопределять:  
Когда нужно предоставить более информативное представление объекта для отладки.

4. == и eql?

Методы == и eql? используются для сравнения объектов. По умолчанию они сравнивают идентификаторы объектов.

Пример:

ruby

Copy

class Product

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

def ==(other)

@name == other.name && @price == other.price

end

def eql?(other)

self == other

end

end

product1 = Product.new("Laptop", 1000)

product2 = Product.new("Laptop", 1000)

puts product1 == product2 # true

Когда переопределять:  
Когда нужно определить, что два объекта считаются равными на основе их атрибутов.

5. hash

Метод hash используется для вычисления хэш-кода объекта. Он должен быть согласован с методом eql?.

Пример:

ruby

Copy

class Product

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

def ==(other)

@name == other.name && @price == other.price

end

def eql?(other)

self == other

end

def hash

[@name, @price].hash

end

end

product1 = Product.new("Laptop", 1000)

product2 = Product.new("Laptop", 1000)

puts product1.hash == product2.hash # true

Когда переопределять:  
Когда объект используется как ключ в хэше, и нужно обеспечить корректное поведение.

6. <=> (оператор сравнения)

Метод <=> используется для сравнения объектов. Он возвращает -1, 0 или 1 в зависимости от того, меньше, равен или больше текущий объект другого объекта.

Пример:

ruby

Copy

class Product

include Comparable

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

def <=>(other)

@price <=> other.price

end

end

product1 = Product.new("Laptop", 1000)

product2 = Product.new("Phone", 500)

puts product1 > product2 # true

Когда переопределять:  
Когда нужно реализовать сортировку или сравнение объектов.

7. method\_missing

Метод method\_missing вызывается, когда вызывается несуществующий метод.

Пример:

ruby

Copy

class Product

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

def method\_missing(method\_name, \*args)

if method\_name.to\_s.start\_with?("set\_")

attribute = method\_name.to\_s[4..-1]

instance\_variable\_set("@#{attribute}", args[0])

else

super

end

end

end

product = Product.new("Laptop", 1000)

product.set\_price(900)

puts product.price # 900

Когда переопределять:  
Когда нужно реализовать динамическое поведение, например, обработку вызовов несуществующих методов.

Паттерны проектирования, связанные с переопределением методов

Шаблонный метод (Template Method)  
Определяет скелет алгоритма, позволяя подклассам переопределять отдельные шаги.

Декоратор (Decorator)  
Добавляет новое поведение объекту, не изменяя его класс.

Адаптер (Adapter)  
Преобразует интерфейс одного класса в интерфейс, ожидаемый клиентом.

Наблюдатель (Observer)  
Позволяет объектам уведомлять другие объекты об изменениях.

Паттерны проектирования (design patterns) — это типичные решения часто встречающихся проблем в проектировании программного обеспечения. Они представляют собой готовые шаблоны, которые можно адаптировать под конкретные задачи. Паттерны помогают создавать гибкий, поддерживаемый и масштабируемый код.

Классификация паттернов проектирования

Паттерны проектирования обычно классифицируются на три основные категории:

Порождающие паттерны (Creational Patterns)  
Решают проблемы создания объектов. Примеры:

Фабричный метод (Factory Method)

Абстрактная фабрика (Abstract Factory)

Строитель (Builder)

Прототип (Prototype)

Одиночка (Singleton)

Структурные паттерны (Structural Patterns)  
Решают проблемы композиции классов и объектов. Примеры:

Адаптер (Adapter)

Мост (Bridge)

Компоновщик (Composite)

Декоратор (Decorator)

Фасад (Facade)

Приспособленец (Flyweight)

Заместитель (Proxy)

Поведенческие паттерны (Behavioral Patterns)  
Решают проблемы взаимодействия объектов и распределения ответственности. Примеры:

Цепочка обязанностей (Chain of Responsibility)

Команда (Command)

Итератор (Iterator)

Посредник (Mediator)

Снимок (Memento)

Наблюдатель (Observer)

Состояние (State)

Стратегия (Strategy)

Шаблонный метод (Template Method)

Посетитель (Visitor)

Взаимосвязь паттернов проектирования, архитектурных паттернов и фреймворков

Паттерны проектирования  
Решают локальные задачи на уровне классов и объектов. Например, как создать объект, как организовать взаимодействие между объектами.

Архитектурные паттерны  
Решают глобальные задачи на уровне всей системы. Они определяют структуру приложения, разделение на компоненты и их взаимодействие. Примеры:

MVC (Model-View-Controller)

MVVM (Model-View-ViewModel)

Микросервисы (Microservices)

Слоистая архитектура (Layered Architecture)

Фреймворки  
Это готовые решения, которые реализуют определенные архитектурные паттерны и включают в себя множество паттернов проектирования. Например:

Ruby on Rails (реализует MVC)

Angular (реализует MVVM)

Spring (реализует Dependency Injection и другие паттерны)

Примеры паттернов проектирования

1. Фабричный метод (Factory Method)

Задача: Создание объектов без указания конкретного класса.

Пример:

ruby

Copy

class Product

attr\_accessor :name, :price

end

class ProductFactory

def self.create\_product(type)

case type

when :laptop

Product.new.tap { |p| p.name = "Laptop"; p.price = 1000 }

when :phone

Product.new.tap { |p| p.name = "Phone"; p.price = 500 }

else

raise "Unknown product type"

end

end

end

product = ProductFactory.create\_product(:laptop)

puts product.name # Laptop

puts product.price # 1000

2. Декоратор (Decorator)

Задача: Динамическое добавление поведения объекту.

Пример:

ruby

Copy

class Product

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

end

class ProductDecorator

def initialize(product)

@product = product

end

def name

"Premium #{@product.name}"

end

def price

@product.price \* 1.1

end

end

product = Product.new("Laptop", 1000)

decorated\_product = ProductDecorator.new(product)

puts decorated\_product.name # Premium Laptop

puts decorated\_product.price # 1100

3. Наблюдатель (Observer)

Задача: Уведомление объектов об изменениях.

Пример:

ruby

Copy

class Subject

attr\_accessor :observers

def initialize

@observers = []

end

def add\_observer(observer)

@observers << observer

end

def notify\_observers

@observers.each { |observer| observer.update(self) }

end

end

class Observer

def update(subject)

puts "Subject state has changed!"

end

end

subject = Subject.new

observer = Observer.new

subject.add\_observer(observer)

subject.notify\_observers # Subject state has changed!

Пример архитектурного паттерна: MVC

MVC (Model-View-Controller) — это архитектурный паттерн, который разделяет приложение на три компонента:

Model — отвечает за данные и бизнес-логику.

View — отвечает за отображение данных.

Controller — управляет взаимодействием между Model и View.

Пример в Ruby on Rails:

ruby

Copy

# Model (app/models/product.rb)

class Product < ApplicationRecord

end

# Controller (app/controllers/products\_controller.rb)

class ProductsController < ApplicationController

def index

@products = Product.all

end

end

# View (app/views/products/index.html.erb)

<h1>Products</h1>

<ul>

<% @products.each do |product| %>

<li><%= product.name %> - <%= product.price %></li>

<% end %>

</ul>

Взаимосвязь паттернов, архитектур и фреймворков

Паттерны проектирования используются внутри фреймворков для решения локальных задач. Например, Rails использует:

ActiveRecord (реализует паттерн "Адаптер" для работы с базой данных).

Observer (реализует паттерн "Наблюдатель" для уведомлений).

Архитектурные паттерны определяют структуру фреймворков. Например:

Rails использует MVC.

Angular использует MVVM.

Фреймворки объединяют паттерны проектирования и архитектурные паттерны в готовые решения для разработки приложений.

**Диаграмма классов**

Диаграмма классов — это графическое представление структуры системы, которое показывает классы, их атрибуты, методы и отношения между ними. Она является частью языка UML (Unified Modeling Language) и используется для визуализации объектно-ориентированных систем.

**Основные элементы диаграммы классов**

1. **Класс (Class)**  
   Представляет собой шаблон для создания объектов. На диаграмме класс изображается в виде прямоугольника, разделенного на три части:
   * **Имя класса** (верхняя часть).
   * **Атрибуты** (средняя часть).
   * **Методы** (нижняя часть).

**Пример:**

Copy

+---------------------+

| Product |

+---------------------+

| - name: String |

| - price: Float |

+---------------------+

| + getName(): String |

| + setPrice(price: Float) |

+---------------------+

1. **Атрибуты (Attributes)**  
   Это переменные, которые хранят состояние объекта. Они указываются в формате:

Copy

- visibility name: type

* + visibility: Модификатор доступа (+ — public, - — private, # — protected).
  + name: Имя атрибута.
  + type: Тип данных (например, String, Integer, Float).

1. **Методы (Methods)**  
   Это функции, которые определяют поведение объекта. Они указываются в формате:

Copy

+ visibility name(parameters): return\_type

* + visibility: Модификатор доступа.
  + name: Имя метода.
  + parameters: Список параметров (например, price: Float).
  + return\_type: Тип возвращаемого значения.

1. **Отношения (Relationships)**  
   Показывают, как классы взаимодействуют друг с другом. Основные типы отношений:
   * **Ассоциация (Association)**: Простая связь между классами.
   * **Агрегация (Aggregation)**: Отношение "часть-целое", где часть может существовать независимо.
   * **Композиция (Composition)**: Более строгая форма агрегации, где часть не может существовать без целого.
   * **Наследование (Inheritance)**: Отношение "является" (is-a).
   * **Зависимость (Dependency)**: Один класс зависит от другого.

**Пример диаграммы классов**

Рассмотрим пример системы для управления продуктами и заказами.

**Диаграмма:**

Copy

+---------------------+ +---------------------+

| Product | | Order |

+---------------------+ +---------------------+

| - name: String | | - id: Integer |

| - price: Float |<-------->| - products: List |

+---------------------+ +---------------------+

| + getName(): String | | + addProduct(p: Product) |

| + setPrice(price: Float) | | + calculateTotal(): Float |

+---------------------+ +---------------------+

^ ^

| |

+---------------------+ +---------------------+

| DiscountedProduct| | Customer |

+---------------------+ +---------------------+

| - discount: Float | | - name: String |

+---------------------+ | - orders: List |

| + getPrice(): Float | +---------------------+

+---------------------+ | + placeOrder(o: Order) |

+---------------------+

**Объяснение:**

* Класс Product имеет атрибуты name и price, а также методы getName и setPrice.
* Класс DiscountedProduct наследует Product и добавляет атрибут discount.
* Класс Order связан с Product через ассоциацию (один заказ может содержать несколько продуктов).
* Класс Customer связан с Order через агрегацию (один клиент может иметь несколько заказов).

**Взаимосвязь диаграммы классов и кода**

Диаграмма классов — это визуальное представление структуры кода. Каждый элемент диаграммы соответствует элементу кода.

**Пример кода для диаграммы выше:**

ruby

Copy

# Класс Product

class Product

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

def get\_name

@name

end

def set\_price(price)

@price = price

end

end

# Класс DiscountedProduct

class DiscountedProduct < Product

attr\_accessor :discount

def initialize(name, price, discount)

super(name, price)

@discount = discount

end

def get\_price

@price - (@price \* @discount / 100)

end

end

# Класс Order

class Order

attr\_accessor :id, :products

def initialize(id)

@id = id

@products = []

end

def add\_product(product)

@products << product

end

def calculate\_total

@products.sum(&:price)

end

end

# Класс Customer

class Customer

attr\_accessor :name, :orders

def initialize(name)

@name = name

@orders = []

end

def place\_order(order)

@orders << order

end

end

**Соответствие диаграмме:**

* Классы Product, DiscountedProduct, Order и Customer соответствуют прямоугольникам на диаграмме.
* Атрибуты и методы классов соответствуют разделам атрибутов и методов на диаграмме.
* Отношения между классами (наследование, ассоциация, агрегация) отражены в коде через наследование (<) и использование объектов других классов.

**Как составлять диаграмму классов**

1. **Определите классы:**  
   Выделите основные сущности системы (например, Product, Order, Customer).
2. **Добавьте атрибуты и методы:**  
   Для каждого класса укажите атрибуты и методы, которые описывают его состояние и поведение.
3. **Определите отношения:**  
   Укажите, как классы связаны друг с другом (ассоциация, наследование, агрегация и т.д.).
4. **Используйте инструменты:**  
   Для создания диаграмм можно использовать инструменты, такие как:
   * **Lucidchart**
   * **Draw.io**
   * **PlantUML**
   * **Visual Paradigm**

ВСЕ О ПАТТЕРНАХ

КЛАССИФИКАЦИЯ

* Порождающие отвечают за удобное и безопасное создание новых объектов
* Поведенческие решают задачи эффективного и безопасного взаимодействия между объектами
* Структурные отвечают за построение удобных в поддержке иерархий классов

ПОРОЖДАЮЩИЕ

# ФАБРИКА(Фабричный метод)

ПРОБЛЕМА

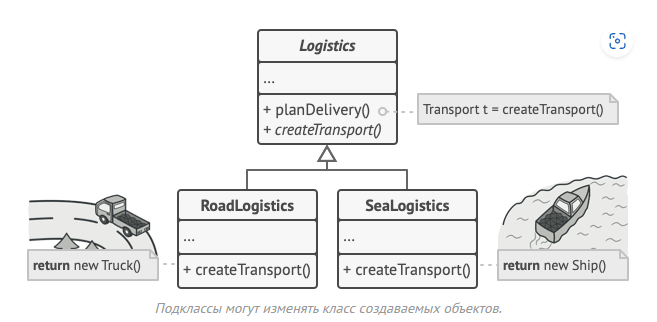
Представьте, что вы создаёте программу управления грузовыми перевозками. Сперва вы рассчитываете перевозить товары только на автомобилях. Поэтому весь ваш код работает с объектами класса Грузовик.

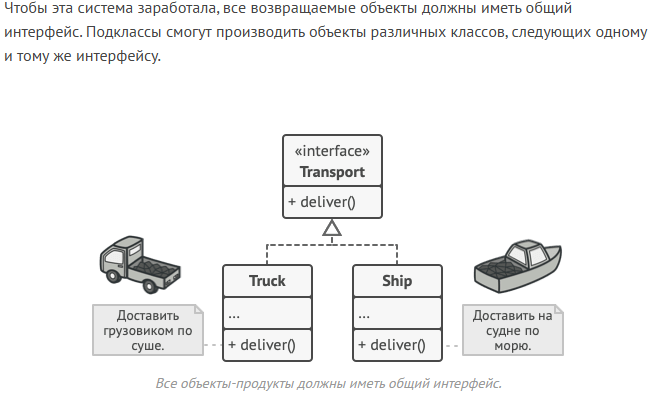
В какой-то момент ваша программа становится настолько известной, что морские перевозчики выстраиваются в очередь и просят добавить поддержку морской логистики в программу.

Отличные новости, правда?! Но как насчёт кода? Большая часть существующего кода жёстко привязана к классам Грузовиков. Чтобы добавить в программу классы морских Судов, понадобится перелопатить всю программу. Более того, если вы потом решите добавить в программу ещё один вид транспорта, то всю эту работу придётся повторить.

В итоге вы получите ужасающий код, наполненный условными операторами, которые выполняют то или иное действие, в зависимости от класса транспорта.

РЕШЕНИЕ





# Класс Создатель объявляет фабричный метод, который должен возвращать объект

# класса Продукт. Подклассы Создателя обычно предоставляют реализацию этого

# метода.

#

# @abstract

**class** **Creator**

# Обратите внимание, что Создатель может также обеспечить реализацию

# фабричного метода по умолчанию.

#

# @abstract

**def** **factory\_method**

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

# Также заметьте, что, несмотря на название, основная обязанность Создателя не

# заключается в создании продуктов. Обычно он содержит некоторую базовую

# бизнес-логику, которая основана на объектах Продуктов, возвращаемых

# фабричным методом. Подклассы могут косвенно изменять эту бизнес-логику,

# переопределяя фабричный метод и возвращая из него другой тип продукта.

#

# @return [String]

**def** **some\_operation**

# Вызываем фабричный метод, чтобы получить объект-продукт.

product = factory\_method

# Далее, работаем с этим продуктом.

"Creator: The same creator's code has just worked with #{product.operation}"

**end**

**end**

# Конкретные Создатели переопределяют фабричный метод для того, чтобы изменить

# тип результирующего продукта.

**class** **ConcreteCreator1** < **Creator**

# Обратите внимание, что сигнатура метода по-прежнему использует тип

# абстрактного продукта, хотя фактически из метода возвращается конкретный

# продукт. Таким образом, Создатель может оставаться независимым от конкретных

# классов продуктов.

#

# @return [ConcreteProduct1]

**def** **factory\_method**

**ConcreteProduct1**.new

**end**

**end**

**class** **ConcreteCreator2** < **Creator**

# @return [ConcreteProduct2]

**def** **factory\_method**

**ConcreteProduct2**.new

**end**

**end**

# Интерфейс Продукта объявляет операции, которые должны выполнять все конкретные

# продукты.

#

# @abstract

**class** **Product**

# return [String]

**def** **operation**

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

**end**

# Конкретные Продукты предоставляют различные реализации интерфейса Продукта.

**class** **ConcreteProduct1** < **Product**

# @return [String]

**def** **operation**

'{Result of the ConcreteProduct1}'

**end**

**end**

**class** **ConcreteProduct2** < **Product**

# @return [String]

**def** **operation**

'{Result of the ConcreteProduct2}'

**end**

**end**

# Клиентский код работает с экземпляром конкретного создателя, хотя и через его

# базовый интерфейс. Пока клиент продолжает работать с создателем через базовый

# интерфейс, вы можете передать ему любой подкласс создателя.

#

# @param [Creator] creator

**def** **client\_code**(creator)

print "Client: I'm not aware of the creator's class, but it still works.\n"\

"#{creator.some\_operation}"

**end**

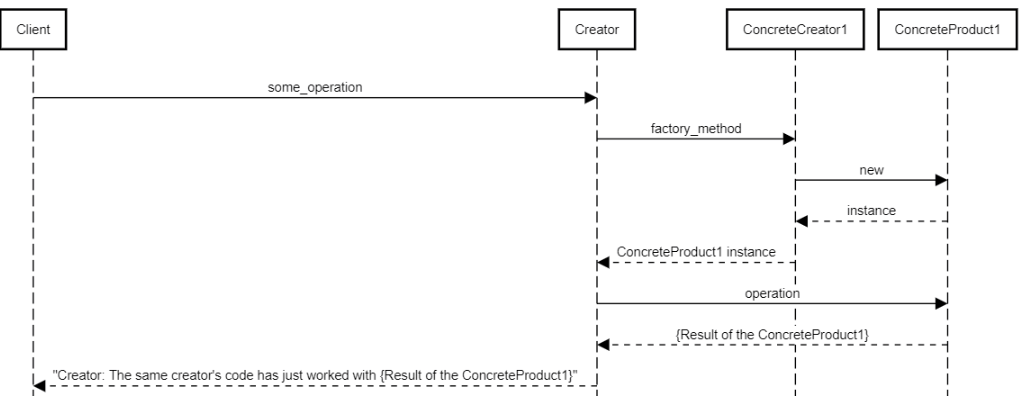
puts 'App: Launched with the ConcreteCreator1.'

client\_code(**ConcreteCreator1**.new)

puts "\n\n"

puts 'App: Launched with the ConcreteCreator2.'

client\_code(**ConcreteCreator2**.new)

Абстрактная фабрика

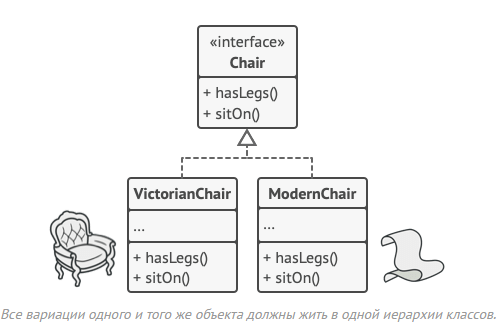
**Абстрактная фабрика** — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



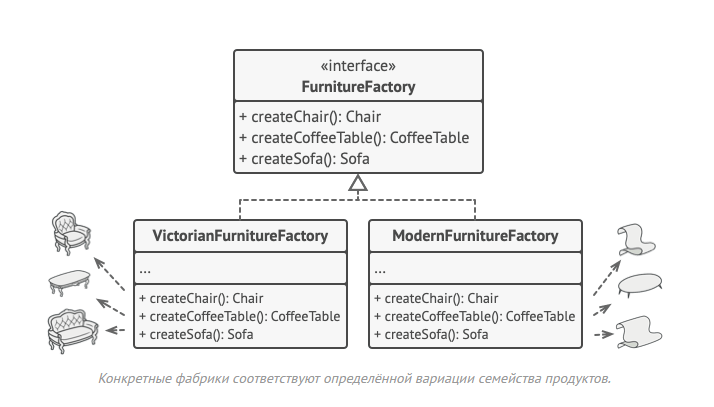
Вам нужен такой способ создавать объекты продуктов, чтобы они сочетались с другими продуктами того же семейства. Это важно, так как клиенты расстраиваются, если получают несочетающуюся мебель. Кроме того, вы не хотите вносить изменения в существующий код при добавлении новых продуктов или семейcтв в программу. Поставщики часто обновляют свои каталоги, и вы бы не хотели менять уже написанный код каждый раз при получении новых моделей мебели.

## Решение

Для начала паттерн Абстрактная фабрика предлагает выделить общие интерфейсы для отдельных продуктов, составляющих семейства. Так, все вариации кресел получат общий интерфейс Кресло, все диваны реализуют интерфейс Диван и так далее.



Далее вы создаёте абстрактную фабрику — общий интерфейс, который содержит методы создания всех продуктов семейства (например, создатьКресло, создатьДиван и создатьСтолик). Эти операции должны возвращать **абстрактные** типы продуктов, представленные интерфейсами, которые мы выделили ранее — Кресла, Диваны и Столики.

 Как насчёт вариаций продуктов? Для каждой вариации семейства продуктов мы должны создать свою собственную фабрику, реализовав абстрактный интерфейс. Фабрики создают продукты одной вариации. Например, ФабрикаМодерн будет возвращать только КреслаМодерн,ДиваныМодерн и СтоликиМодерн.

Клиентский код должен работать как с фабриками, так и с продуктами только через их общие интерфейсы. Это позволит подавать в ваши классы любой тип фабрики и производить любые продукты, ничего не ломая.

РЕАЛИЗАЦИЯ

# Интерфейс Абстрактной Фабрики объявляет набор методов, которые возвращают

# различные абстрактные продукты. Эти продукты называются семейством и связаны

# темой или концепцией высокого уровня. Продукты одного семейства обычно могут

# взаимодействовать между собой. Семейство продуктов может иметь несколько

# вариаций, но продукты одной вариации несовместимы с продуктами другой.

#

# @abstract

**class** **AbstractFactory**

# @abstract

**def** **create\_product\_a**

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

# @abstract

**def** **create\_product\_b**

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

**end**

# Конкретная Фабрика производит семейство продуктов одной вариации. Фабрика

# гарантирует совместимость полученных продуктов. Обратите внимание, что

# сигнатуры методов Конкретной Фабрики возвращают абстрактный продукт, в то

# время как внутри метода создается экземпляр конкретного продукта.

**class** **ConcreteFactory1** < **AbstractFactory**

**def** **create\_product\_a**

**ConcreteProductA1**.new

**end**

**def** **create\_product\_b**

**ConcreteProductB1**.new

**end**

**end**

# Каждая Конкретная Фабрика имеет соответствующую вариацию продукта.

**class** **ConcreteFactory2** < **AbstractFactory**

**def** **create\_product\_a**

**ConcreteProductA2**.new

**end**

**def** **create\_product\_b**

**ConcreteProductB2**.new

**end**

**end**

# Каждый отдельный продукт семейства продуктов должен иметь базовый интерфейс.

# Все вариации продукта должны реализовывать этот интерфейс.

#

# @abstract

**class** **AbstractProductA**

# @abstract

#

# @return [String]

**def** **useful\_function\_a**

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

**end**

# Конкретные продукты создаются соответствующими Конкретными Фабриками.

**class** **ConcreteProductA1** < **AbstractProductA**

**def** **useful\_function\_a**

'The result of the product A1.'

**end**

**end**

**class** **ConcreteProductA2** < **AbstractProductA**

**def** **useful\_function\_a**

'The result of the product A2.'

**end**

**end**

# Базовый интерфейс другого продукта. Все продукты могут взаимодействовать друг

# с другом, но правильное взаимодействие возможно только между продуктами одной

# и той же конкретной вариации.

#

# @abstract

**class** **AbstractProductB**

# Продукт B способен работать самостоятельно...

#

# @abstract

**def** **useful\_function\_b**

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

# ...а также взаимодействовать с Продуктами A той же вариации.

#

# Абстрактная Фабрика гарантирует, что все продукты, которые она создает,

# имеют одинаковую вариацию и, следовательно, совместимы.

#

# @abstract

#

# @param [AbstractProductA] collaborator

**def** **another\_useful\_function\_b**(\_collaborator)

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

**end**

# Конкретные Продукты создаются соответствующими Конкретными Фабриками.

**class** **ConcreteProductB1** < **AbstractProductB**

# @return [String]

**def** **useful\_function\_b**

'The result of the product B1.'

**end**

# Продукт B1 может корректно работать только с Продуктом A1. Тем не менее, он

# принимает любой экземпляр Абстрактного Продукта А в качестве аргумента.

#

# @param [AbstractProductA] collaborator

#

# @return [String]

**def** **another\_useful\_function\_b**(collaborator)

result = collaborator.useful\_function\_a

"The result of the B1 collaborating with the (#{result})"

**end**

**end**

**class** **ConcreteProductB2** < **AbstractProductB**

# @return [String]

**def** **useful\_function\_b**

'The result of the product B2.'

**end**

# Продукт B2 может корректно работать только с Продуктом A2. Тем не менее, он

# принимает любой экземпляр Абстрактного Продукта А в качестве аргумента.

#

# @param [AbstractProductA] collaborator

**def** **another\_useful\_function\_b**(collaborator)

result = collaborator.useful\_function\_a

"The result of the B2 collaborating with the (#{result})"

**end**

**end**

# Клиентский код работает с фабриками и продуктами только через абстрактные

# типы: Абстрактная Фабрика и Абстрактный Продукт. Это позволяет передавать

# любой подкласс фабрики или продукта клиентскому коду, не нарушая его.

#

# @param [AbstractFactory] factory

**def** **client\_code**(factory)

product\_a = factory.create\_product\_a

product\_b = factory.create\_product\_b

puts product\_b.useful\_function\_b

puts product\_b.another\_useful\_function\_b(product\_a)

**end**

# Клиентский код может работать с любым конкретным классом фабрики.

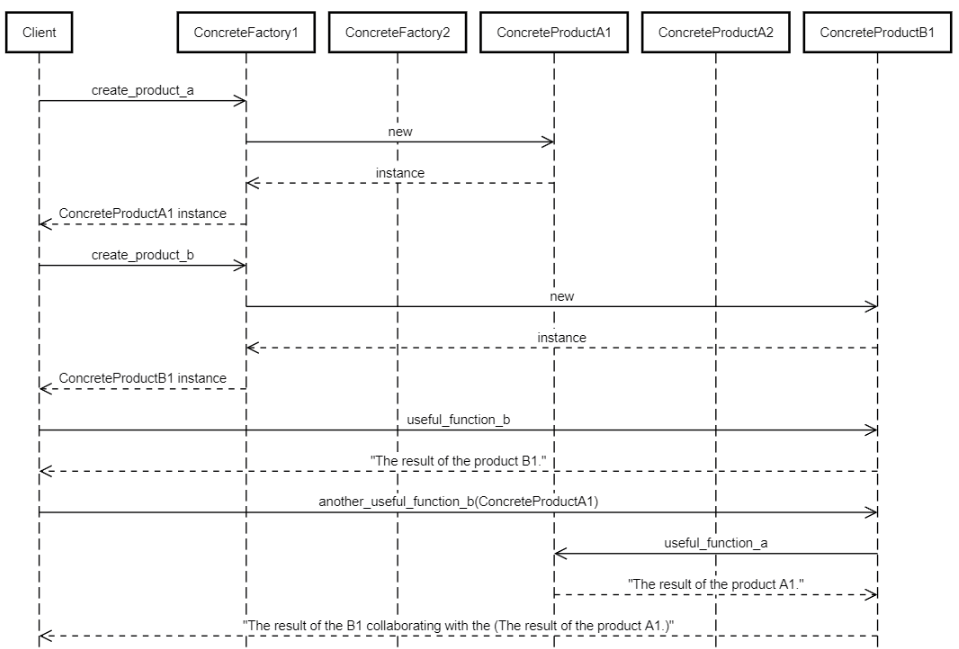
puts 'Client: Testing client code with the first factory type:'

client\_code(**ConcreteFactory1**.new)

puts "\n"

puts 'Client: Testing the same client code with the second factory type:'

client\_code(**ConcreteFactory2**.new)



**Singleton (одиночка)**

**Одиночка** — это порождающий паттерн, который гарантирует существование только одного объекта определённого класса, а также позволяет достучаться до этого объекта из любого места программы.

**Проблема**:  
В некоторых случаях требуется, чтобы у класса был только один экземпляр, например, для управления подключением к базе данных или конфигурацией приложения. Если создавать несколько экземпляров, это может привести к проблемам с согласованностью данных.

# Класс Одиночка предоставляет метод instance, который позволяет клиентам

# получить доступ к уникальному экземпляру одиночки.

**class** **Singleton**

@instance = new

private\_class\_method **:new**

# Статический метод, управляющий доступом к экземпляру одиночки.

#

# Эта реализация позволяет вам расширять класс Одиночки, сохраняя повсюду

# только один экземпляр каждого подкласса.

**def** **self**.instance

@instance

**end**

# Наконец, любой одиночка должен содержать некоторую бизнес-логику, которая

# может быть выполнена на его экземпляре.

**def** **some\_business\_logic**

# ...

**end**

**end**

# Клиентский код.

s1 = **Singleton**.instance

s2 = **Singleton**.instance

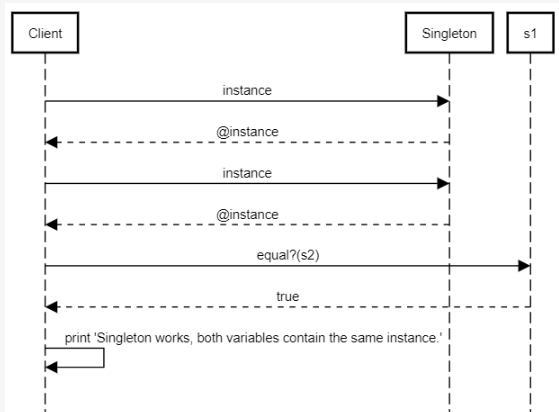
**if** s1.equal?(s2)

print 'Singleton works, both variables contain the same instance.'

**else**

print 'Singleton failed, variables contain different instances.'

**end**



1. **Строитель (Builder)**:
   * Проблема: Создание сложных объектов с множеством параметров.
   * Решение: Разделяет процесс создания объекта на отдельные шаги, позволяя создавать объекты с разными конфигурациями.
2. **Прототип (Prototype)**:
   * Проблема: Создание объектов, которые должны быть копиями существующих объектов.
   * Решение: Позволяет копировать существующие объекты без привязки к их классам.

SOLID

1. S (SRP) / Single Responsibility Principle - Принцип единственной ответственности
2. O (OCP) / Open–Closed Principle - Принцип открытости/закрытости
3. L (LSP) / Liskov Substitution Principle - Принцип подстановки Лисков
4. I (ISP) / Interface Segregation Principle - Принцип разделения интерфейса
5. D (DIP) / Dependency Inversion Principle - Принцип инверсии зависимостей
6. Принцип единственной ответственности - Модуль должен иметь одну и только одну причину для изменения.

# =============================================

# Single Responsibility Principle (Принцип единственной ответственности)

#

# A class should have only one reason to change

#

# Robert C. Martin

# =============================================

class LogSender

def initialize(data, user)

@data = data

@user = user

@message = ''

end

def generate\_log!

@message = @data.map { |row| "[#{Time.now}] User: #{row.user} action #{row.action}"}.join("\n")

end

def send\_log

Mailer.deliver(to: @user.email, message: @message)

end

end

# Что делает класс?

# Класс создает отчет и отправляет его.

#

# === USAGE ===================================

#

# log\_sender = LogSender.new(data, account)

# log\_sender.genetate\_log!

# log\_sender.send\_log

#

# =============================================

#

# === WARNING =================================

#

# Класс решает две задачи: создает и отправляет

# Это противоречит принципу SRP. (Single

# Responsibility Principle)

# Класс должен отвечать только за одну задачу,

# иначе это похоже на божественный класс,

# который делает все - это плохо.

#

# =============================================

#

# === SOLUTION ================================

#

# Сложный класс должен быть разбит на несколько

# простых составляющих, отвечающих за

# определенный аспект поведения. Это упрощает

# расширение системы, редактирование классов.

#

# =============================================

class LogGenerator

def initialize(data)

@data = data

end

def generate

@data.map { |row| "[#{Time.now}] User: #{row.user} action #{row.action}"}.join("\n")

end

end

class LogSender

def initialize(data, user)

@data = data

@user = user

end

def send

Mailer.deliver(to: @user.email, message: @message)

end

end

# === USAGE ===================================

#

# log = LogGenerator.new(data)

# logSender = LogSender.new(log, user)

# logSender.send

# =============================================

1. Принцип открытости/закрытости - Должна иметься возможность расширять поведение программных сущностей без их изменения (путем наследования). Открыт для расширения, закрыт для изменения

Было

  class BubbleSort

    def self.sort(array)

      puts "Sorting using Bubble Sort"

      array.sort

    end

  end

  class QuickSort

    def self.sort(array)

      puts "Sorting using Quick Sort"

      array.sort

    end

  end

  class MergeSort

    def self.sort(array)

      puts "Sorting using Merge Sort"

      array.sort

    end

  end

  # Класс Person

  class Person

    attr\_accessor :fullname

    def initialize(fullname)

      @fullname = fullname

    end

  end

  # Класс для списка людей

  class PersonList

    attr\_accessor :persons

    def initialize(persons)

      @persons = persons

    end

    def sort

        if @persons.size < 10

            BubbleSort.sort(array)

          elsif @persons.size < 1000

            MergeSort.sort(array)

          else

            QuickSort.sort(array)

          end

        end

    end

  end

Стало

# Базовый класс для сортировки

class Sort

    def self.sort(array)

      raise NotImplementedError, "#{self.class} has not implemented method '#{\_\_method\_\_}'"

    end

  end

  # Реализация пузырьковой сортировки

  class BubbleSort < Sort

    def self.sort(array)

      puts "Sorting using Bubble Sort"

      array.sort

    end

  end

  # Реализация быстрой сортировки

  class QuickSort < Sort

    def self.sort(array)

      puts "Sorting using Quick Sort"

      array.sort

    end

  end

  # Реализация сортировки слиянием

  class MergeSort < Sort

    def self.sort(array)

      puts "Sorting using Merge Sort"

      array.sort

    end

  end

  # Клиент для выбора стратегии сортировки

  class SortClient

    def self.sort(array)

      if array.size < 10

        BubbleSort.sort(array)

      elsif array.size < 1000

        MergeSort.sort(array)

      else

        QuickSort.sort(array)

      end

    end

  end

  # Класс Person

  class Person

    attr\_accessor :fullname

    def initialize(fullname)

      @fullname = fullname

    end

  end

  # Класс для списка людей

  class PersonList

    attr\_accessor :persons

    def initialize(persons)

      @persons = persons

    end

    def sort

      SortClient.sort(@persons)

    end

  end

  # Пример использования

  persons = [Person.new("Alice"), Person.new("Bob"), Person.new("Charlie")]

  person\_list = PersonList.new(persons)

  person\_list.sort

1. Принцип подстановки Лисков - Наследующий класс должен дополнять, а не изменять базовый (наследование).

Было:

class Database

    def connect

      raise NotImplementedError, "#{self.class} has not implemented method '#{\_\_method\_\_}'"

    end

    def read

      raise NotImplementedError, "#{self.class} has not implemented method '#{\_\_method\_\_}'"

    end

    def write

      raise NotImplementedError, "#{self.class} has not implemented method '#{\_\_method\_\_}'"

    end

    def join\_tables

         raise NotImplementedError, "#{self.class} has not implemented method '#{\_\_method\_\_}'"

      end

  end

  class MySQLDatabase < SQLDatabase

    def connect

      puts "Connecting to MySQL database..."

    end

    def read

      puts "Reading from MySQL database..."

    end

    def write

      puts "Writing to MySQL database..."

    end

    def join\_tables

      puts "Joining tables in MySQL database..."

    end

  end

  class MongoDatabase < NOSQLDatabase

    def connect

      puts "Connecting to MongoDB..."

    end

    def read

      puts "Reading from MongoDB..."

    end

    def write

      puts "Writing to MongoDB..."

    end

    def join\_tables

        return null

    end

end

Стало

class Database

    def connect

      raise NotImplementedError, "#{self.class} has not implemented method '#{\_\_method\_\_}'"

    end

    def read

      raise NotImplementedError, "#{self.class} has not implemented method '#{\_\_method\_\_}'"

    end

    def write

      raise NotImplementedError, "#{self.class} has not implemented method '#{\_\_method\_\_}'"

    end

  end

  class SQLDatabase < Database

    def connect

      puts "Connecting to SQL database..."

    end

    def read

      puts "Reading from SQL database..."

    end

    def write

      puts "Writing to SQL database..."

    end

    def join\_tables

      puts "Joining tables in SQL database..."

    end

  end

  class NOSQLDatabase < Database

    def connect

      puts "Connecting to NoSQL database..."

    end

    def read

      puts "Reading from NoSQL database..."

    end

    def write

      puts "Writing to NoSQL database..."

    end

    def create\_index

      puts "Creating index in NoSQL database..."

    end

  end

  class MySQLDatabase < SQLDatabase

    def connect

      puts "Connecting to MySQL database..."

    end

    def read

      puts "Reading from MySQL database..."

    end

    def write

      puts "Writing to MySQL database..."

    end

    def join\_tables

      puts "Joining tables in MySQL database..."

    end

  end

  class MongoDatabase < NOSQLDatabase

    def connect

      puts "Connecting to MongoDB..."

    end

    def read

      puts "Reading from MongoDB..."

    end

    def write

      puts "Writing to MongoDB..."

    end

    def create\_index

      puts "Creating index in MongoDB..."

    end

    def merge\_documents

      puts "Merging documents in MongoDB..."

    end

  end

  def start\_app(database)

    database.connect

  end

  # Использование

  start\_app(MongoDatabase.new)

  start\_app(MySQLDatabase.new)

1. Принцип разделения интерфейса - Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения (прочь избыточность).

Было

class Car

    def open\_door; end

    def start\_engine; end

    def repair\_engine; end

  end

  class Driver

    def drive\_car(car)

      car.open\_door

      car.start\_engine

    end

  end

  class Machanic

    def repair\_car(car)

      car.repair\_enginge

    end

  end

Стало

# Интерфейс для водителя

class Drivable

    def open\_door

      raise NotImplementedError, "#{self.class} has not implemented method '#{\_\_method\_\_}'"

    end

    def start\_engine

      raise NotImplementedError, "#{self.class} has not implemented method '#{\_\_method\_\_}'"

    end

  end

  # Интерфейс для механика

  class Repairable

    def repair\_engine

      raise NotImplementedError, "#{self.class} has not implemented method '#{\_\_method\_\_}'"

    end

  end

  # Класс Car реализует оба интерфейса

  class Car < Drivable

    def open\_door

      puts "Door opened"

    end

    def start\_engine

      puts "Engine started"

    end

  end

  # Класс CarSettings реализует интерфейс Repairable

  class CarSettings < Repairable

    def repair\_engine

      puts "Engine repaired"

    end

  end

  # Класс Driver зависит только от Drivable

  class Driver

    def drive\_car(car)

      car.open\_door

      car.start\_engine

    end

  end

  # Класс Mechanic зависит только от Repairable

  class Mechanic

    def repair\_car(car)

      car.repair\_engine

    end

  end

  # Пример использования

  car = Car.new

  car\_settings = CarSettings.new

  driver = Driver.new

  driver.drive\_car(car) # Door opened, Engine started

  mechanic = Mechanic.new

  mechanic.repair\_car(car\_settings) # Engine repaired

1. Принцип инверсии зависимостей - Зависимость на Абстракциях, а не на реализации.

class LogSender

    def initialize(data, user)

      @data = data

      @user = user

    end

    def send\_mail

      MailSender.new.send\_message(@data, @user)

    end

    def send\_vk

      VKSender.new.send\_message(@data, @user)

    end

  end

  class Sender

    def send\_message(data, user); end

  end

  class MailSender < Sender

    def send\_message(data, user); end

  end

  class VKSender < Sender

    def send\_message(data, user); end

  end

  # === WARNING =================================

  #

  # LogSender завuсит от двух классов реализующих логику отправки сообщения

  # MailSender и VKSender, те логика построенна относительно деталей реализации,

  # а не от асбтракции

  #

  # =============================================

  #

  # === SOLUTION ================================

  #

  # =============================================

  class LogSender

    def initialize(data, user)

      @data = data

      @user = user

    end

    def send\_report(sender = MailSender.new)

      sender.send\_message(@data, @user)

    end

  end

  class Sender

    def send\_message(data, user); end

  end

  class MailSender < Sender

    def send\_message(data, user)

      # implementation

    end

  end

  class VKSender < Sender

    def send\_message(data, user)

      # implementation

    end

  end

  # Что изменилось?

  # Мы подрефачили код таким образом, чтобы

  # появилась зависимость от интерфейса другого

  # модуля, а именно наличие метода send, а не

  # от зависимости способа/механизма отправления

  # отчета.

12. Расскажите про подходы в сравнении объектов на равенство и на больше-меньше в руби. Расскажите, какие принципы ООП и паттерны проектирования используются в реализации данных подходов.

В Ruby сравнение объектов на равенство и на больше-меньше реализуется с помощью методов, которые можно переопределять в классах. Эти методы позволяют задать логику сравнения объектов в зависимости от их внутреннего состояния.

### Основные методы для сравнения

1. **Методы для сравнения на равенство**:
   * ==: Проверяет, равны ли два объекта.
   * eql?: Проверяет, равны ли два объекта по значению и типу.
   * equal?: Проверяет, являются ли два объекта одним и тем же объектом в памяти.
2. **Методы для сравнения на больше-меньше**:
   * <=>: Возвращает -1, 0 или 1 в зависимости от того, меньше, равно или больше текущий объект другого объекта.
   * <, >, <=, >=: Используют результат метода <=> для сравнения.

### Примеры сравнения объектов

#### 1. ****Сравнение на равенство****

**Класс Person**:

ruby

Copy

class Person

attr\_accessor :name, :age

def initialize(name, age)

@name = name

@age = age

end

def ==(other)

other.is\_a?(Person) && @name == other.name && @age == other.age

end

def eql?(other)

self == other

end

def hash

[@name, @age].hash

end

end

person1 = Person.new("Alice", 30)

person2 = Person.new("Alice", 30)

person3 = Person.new("Bob", 25)

puts person1 == person2 # true

puts person1 == person3 # false

**Объяснение**:

* Метод == переопределен для сравнения объектов по их атрибутам.
* Метод eql? используется для сравнения объектов в контексте хэшей и множеств.
* Метод hash должен быть переопределен, если переопределен eql?, чтобы объекты с одинаковыми атрибутами имели одинаковый хэш.

#### 2. ****Сравнение на больше-меньше****

**Класс Product**:

ruby

Copy

class Product

include Comparable

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

def <=>(other)

@price <=> other.price

end

end

product1 = Product.new("Laptop", 1000)

product2 = Product.new("Phone", 500)

product3 = Product.new("Tablet", 750)

puts product1 > product2 # true

puts product2 < product3 # true

puts product1.between?(product2, product3) # false

**Объяснение**:

* Метод <=> переопределен для сравнения объектов по цене.
* Модуль Comparable добавляет методы <, >, <=, >=, ==, between? на основе метода <=>.

### Принципы ООП и паттерны проектирования

#### 1. ****Принцип полиморфизма****

* Полиморфизм позволяет объектам разных классов использовать одинаковые методы для сравнения.
* Пример: Метод <=> может быть реализован в разных классах для сравнения объектов по разным критериям.

#### 2. ****Принцип инкапсуляции****

* Логика сравнения инкапсулирована внутри класса. Внешний код не знает, как именно сравниваются объекты.
* Пример: Метод == скрывает детали сравнения атрибутов объекта.

#### 3. ****Принцип открытости/закрытости****

* Класс открыт для расширения (можно добавлять новые атрибуты для сравнения), но закрыт для модификации (не нужно изменять существующий код сравнения).
* Пример: Добавление нового атрибута в класс Person не требует изменения метода ==.

#### 4. ****Паттерн "Стратегия"****

* Метод <=> можно рассматривать как стратегию сравнения объектов. В зависимости от реализации, объекты могут сравниваться по разным критериям.
* Пример: В классе Product объекты сравниваются по цене, но можно добавить другую стратегию (например, по имени).

### Пример использования паттерна "Стратегия"

**Класс Comparator**:

ruby

Copy

class Comparator

def compare(a, b)

raise NotImplementedError, "#{self.class} has not implemented method '#{\_\_method\_\_}'"

end

end

class PriceComparator < Comparator

def compare(a, b)

a.price <=> b.price

end

end

class NameComparator < Comparator

def compare(a, b)

a.name <=> b.name

end

end

class Product

attr\_accessor :name, :price

def initialize(name, price)

@name = name

@price = price

end

def <=>(other)

@price <=> other.price

end

end

# Использование

products = [

Product.new("Laptop", 1000),

Product.new("Phone", 500),

Product.new("Tablet", 750)

]

price\_comparator = PriceComparator.new

name\_comparator = NameComparator.new

# Сортировка по цене

sorted\_by\_price = products.sort { |a, b| price\_comparator.compare(a, b) }

puts sorted\_by\_price.map(&:name) # ["Phone", "Tablet", "Laptop"]

# Сортировка по имени

sorted\_by\_name = products.sort { |a, b| name\_comparator.compare(a, b) }

puts sorted\_by\_name.map(&:name) # ["Laptop", "Phone", "Tablet"]

**Объяснение**:

* Класс Comparator определяет интерфейс для сравнения объектов.
* Классы PriceComparator и NameComparator реализуют разные стратегии сравнения.
* Класс Product использует стратегию сравнения по цене по умолчанию.

13. Расскажите, как иллюстрируется взаимодействие объектов на диаграмме последовательности. Покажите тривиальный пример, далее покажите пример диаграммы последовательностей, иллюстрирующий взаимодействие объектов в рамках любого архитектурного паттерна.

**Основные элементы диаграммы последовательности**

1. **Объекты (участники)**:
   * Представляются прямоугольниками с именем объекта и классом (например, :User).
   * Жизненная линия объекта (вертикальная пунктирная линия) показывает время жизни объекта.
2. **Сообщения**:
   * Стрелки между объектами, показывающие вызовы методов или передачу данных.
   * Типы сообщений:
     + **Синхронные**: Вызов метода с ожиданием ответа (сплошная стрелка с треугольником).
     + **Асинхронные**: Вызов без ожидания ответа (стрелка с открытым наконечником).
     + **Возврат**: Ответ на синхронный вызов (пунктирная стрелка).
3. **Активация**:
   * Прямоугольник на жизненной линии объекта, показывающий период активности объекта (например, выполнение метода).
4. **Условия и циклы**:
   * Показывают ветвление и повторение в потоке управления.

class User

    def save

      puts "User saved to database"

    end

  end

  # Контроллер

  class UsersController

    def create

      user = User.new

      user.save

      render\_response("User created")

    end

    private

    def render\_response(message)

      puts message

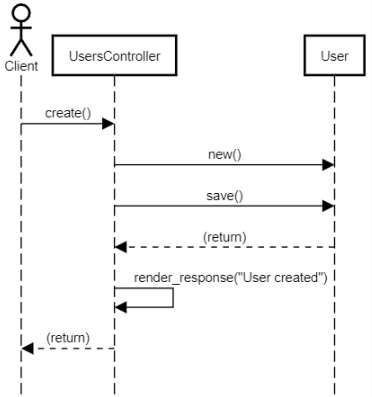
    end

  end

  # Пример использования

  controller = UsersController.new

  controller.create



### Архитектурный паттерн MVC (Model-View-Controller)

**MVC** — это архитектурный паттерн, который разделяет приложение на три компонента:

1. **Model (Модель)**: Отвечает за данные и бизнес-логику.
2. **View (Представление)**: Отвечает за отображение данных пользователю.
3. **Controller (Контроллер)**: Управляет взаимодействием между Model и View.

### Базовый подход

1. **Model**:
   * Содержит данные и логику их обработки.
   * Не зависит от View и Controller.
   * Пример: Класс User, который представляет данные пользователя и методы для работы с ними.
2. **View**:
   * Отображает данные пользователю.
   * Получает данные от Model, но не изменяет их.
   * Пример: HTML-шаблон, который отображает список пользователей.
3. **Controller**:
   * Обрабатывает пользовательские запросы.
   * Управляет потоком данных между Model и View.
   * Пример: Класс UsersController, который обрабатывает запросы на создание, чтение, обновление и удаление пользователей.

### Паттерны проектирования в MVC

1. **Observer (Наблюдатель)**:
   * View наблюдает за изменениями в Model и обновляется автоматически.
   * Пример: В Rails View обновляется при изменении данных в Model.
2. **Strategy (Стратегия)**:
   * Контроллер может использовать разные стратегии для обработки запросов.
   * Пример: В Rails контроллеры могут использовать разные форматы ответа (HTML, JSON).
3. **Composite (Компоновщик)**:
   * View может состоять из нескольких компонентов (например, шаблоны и частичные шаблоны).
   * Пример: В Rails используются частичные шаблоны (partials).
4. **Front Controller (Фронтальный контроллер)**:
   * Все запросы обрабатываются через один центральный контроллер.
   * Пример: В Rails ApplicationController является базовым контроллером для всех запросов.

### Взаимодействие объектов в рамках операций CRUD

Рассмотрим пример реализации операций CRUD (Create, Read, Update, Delete) в MVC.

#### Сценарий: Создание пользователя

1. Пользователь отправляет форму на создание нового пользователя.
2. Контроллер (UsersController) обрабатывает запрос.
3. Контроллер создает модель (User) и сохраняет ее в базе данных.
4. Контроллер возвращает ответ пользователю.

**Код**:

ruby

Copy

# Модель

class User < ApplicationRecord

end

# Контроллер

class UsersController < ApplicationController

def create

@user = User.new(user\_params)

if @user.save

redirect\_to @user, notice: 'User was successfully created.'

else

render :new

end

end

private

def user\_params

params.require(:user).permit(:name, :email)

end

end

# Представление (app/views/users/new.html.erb)

<%= form\_with(model: @user) do |form| %>

<%= form.label :name %>

<%= form.text\_field :name %>

<%= form.label :email %>

<%= form.text\_field :email %>

<%= form.submit %>

<% end %>

**Диаграмма последовательности**:

plaintext

Copy

Client -> UsersController: create()

UsersController -> User: new(user\_params)

UsersController -> User: save()

User --> UsersController: (return)

UsersController -> UsersController: redirect\_to(@user)

UsersController --> Client: (return)

#### Сценарий: Чтение пользователя

1. Пользователь запрашивает информацию о пользователе.
2. Контроллер (UsersController) обрабатывает запрос.
3. Контроллер получает данные из модели (User).
4. Контроллер передает данные в представление.
5. Представление отображает данные пользователю.

**Код**:

ruby

Copy

# Контроллер

class UsersController < ApplicationController

def show

@user = User.find(params[:id])

end

end

# Представление (app/views/users/show.html.erb)

<h1><%= @user.name %></h1>

<p>Email: <%= @user.email %></p>

**Диаграмма последовательности**:

plaintext

Copy

Client -> UsersController: show()

UsersController -> User: find(params[:id])

User --> UsersController: (return)

UsersController -> UsersController: render(:show)

UsersController --> Client: (return)

### Фреймворки, использующие MVC

1. **Ruby on Rails**:
   * **Model**: ActiveRecord.
   * **View**: ERB, Haml, Slim.
   * **Controller**: ActionController.
2. **Django (Python)**:
   * **Model**: Django ORM.
   * **View**: Django Templates.
   * **Controller**: Django Views.
3. **Spring MVC (Java)**:
   * **Model**: POJOs (Plain Old Java Objects).
   * **View**: JSP, Thymeleaf.
   * **Controller**: Spring Controllers.
4. **ASP.NET MVC (C#)**:
   * **Model**: Entity Framework.
   * **View**: Razor.
   * **Controller**: ASP.NET Controllers.

### MVP Архитектурный паттерн MVP (Model-View-Presenter)

**MVP** — это архитектурный паттерн, который является эволюцией MVC. Он был разработан для решения некоторых проблем, возникающих при использовании MVC, особенно в контексте разработки пользовательских интерфейсов.

### Проблематика MVC, которая привела к появлению MVP

1. **Тесная связь между View и Controller**:
   * В MVC View и Controller часто оказываются слишком тесно связанными, что затрудняет тестирование и повторное использование кода.
   * View может напрямую обращаться к Model, что нарушает принцип единственной ответственности.
2. **Сложность тестирования**:
   * В MVC Controller часто содержит логику, которая зависит от View, что делает его трудным для тестирования.
   * View может быть сложно тестировать из-за ее зависимости от Controller и Model.

### Базовый подход MVP

**MVP** разделяет приложение на три компонента:

1. **Model (Модель)**: Отвечает за данные и бизнес-логику.
2. **View (Представление)**: Отвечает за отображение данных пользователю.
3. **Presenter (Презентер)**: Посредник между Model и View. Он обрабатывает пользовательские действия, обновляет Model и обновляет View.

### Взаимодействие объектов в рамках операций CRUD

Рассмотрим пример реализации операций CRUD (Create, Read, Update, Delete) в MVP.

#### Сценарий: Создание пользователя

1. Пользователь отправляет форму на создание нового пользователя.
2. View передает данные Presenter.
3. Presenter создает модель (User) и сохраняет ее в базе данных.
4. Presenter обновляет View.

**Код**:

ruby

Copy

# Модель

class User

attr\_accessor :name, :email

def save

puts "User saved to database"

end

end

# Представление

class UserView

def get\_user\_data

{ name: "Alice", email: "alice@example.com" }

end

def show\_message(message)

puts message

end

end

# Презентер

class UserPresenter

def initialize(view)

@view = view

end

def create\_user

user\_data = @view.get\_user\_data

user = User.new

user.name = user\_data[:name]

user.email = user\_data[:email]

user.save

@view.show\_message("User created successfully")

end

end

# Пример использования

view = UserView.new

presenter = UserPresenter.new(view)

presenter.create\_user

**Диаграмма последовательности**:

plaintext

Copy

Client -> UserView: get\_user\_data()

UserView --> Client: (return)

Client -> UserPresenter: create\_user()

UserPresenter -> User: new()

UserPresenter -> User: save()

User --> UserPresenter: (return)

UserPresenter -> UserView: show\_message("User created successfully")

UserView --> Client: (return)

#### Сценарий: Чтение пользователя

1. Пользователь запрашивает информацию о пользователе.
2. View передает запрос Presenter.
3. Presenter получает данные из модели (User).
4. Presenter обновляет View.

**Код**:

ruby

Copy

# Модель

class User

attr\_accessor :name, :email

def self.find(id)

# Заглушка для поиска пользователя

new.tap do |user|

user.name = "Alice"

user.email = "alice@example.com"

end

end

end

# Представление

class UserView

def show\_user\_details(user)

puts "Name: #{user.name}, Email: #{user.email}"

end

end

# Презентер

class UserPresenter

def initialize(view)

@view = view

end

def show\_user(id)

user = User.find(id)

@view.show\_user\_details(user)

end

end

# Пример использования

view = UserView.new

presenter = UserPresenter.new(view)

presenter.show\_user(1)

**Диаграмма последовательности**:

plaintext

Copy

Client -> UserPresenter: show\_user(1)

UserPresenter -> User: find(1)

User --> UserPresenter: (return)

UserPresenter -> UserView: show\_user\_details(user)

UserView --> Client: (return)

### Фреймворки, использующие MVP

1. **Android**:
   * MVP широко используется в разработке Android-приложений для разделения логики и интерфейса.
   * Пример: Библиотеки Mosby, Moxy.
2. **Windows Forms (C#)**:
   * MVP используется для разработки desktop-приложений на платформе .NET.
   * Пример: Библиотека MVP Light.
3. **GWT (Google Web Toolkit)**:
   * MVP используется для разработки веб-приложений на Java.
   * Пример: Библиотека GWTP.

MVVM

### Базовый подход MVVM

**MVVM** разделяет приложение на три компонента:

1. **Model (Модель)**: Отвечает за данные и бизнес-логику.
2. **View (Представление)**: Отвечает за отображение данных пользователю.
3. **ViewModel (Модель представления)**: Посредник между Model и View. Он содержит логику представления и состояние View, а также обеспечивает привязку данных между View и Model.

### Взаимодействие объектов в рамках операций CRUD

Рассмотрим пример реализации операций CRUD (Create, Read, Update, Delete) в MVVM.

#### Сценарий: Создание пользователя

1. Пользователь отправляет форму на создание нового пользователя.
2. View передает данные ViewModel.
3. ViewModel создает модель (User) и сохраняет ее в базе данных.
4. ViewModel обновляет View через привязку данных.

**Код**:

ruby

Copy

# Модель

class User

attr\_accessor :name, :email

def save

puts "User saved to database"

end

end

# ViewModel

class UserViewModel

attr\_accessor :name, :email

def initialize

@user = User.new

end

def create\_user

@user.name = @name

@user.email = @email

@user.save

puts "User created successfully"

end

end

# Представление

class UserView

def initialize(view\_model)

@view\_model = view\_model

end

def submit\_form(name, email)

@view\_model.name = name

@view\_model.email = email

@view\_model.create\_user

end

end

# Пример использования

view\_model = UserViewModel.new

view = UserView.new(view\_model)

view.submit\_form("Alice", "alice@example.com")

**Диаграмма последовательности**:

plaintext

Copy

Client -> UserView: submit\_form("Alice", "alice@example.com")

UserView -> UserViewModel: name = "Alice"

UserView -> UserViewModel: email = "alice@example.com"

UserView -> UserViewModel: create\_user()

UserViewModel -> User: new()

UserViewModel -> User: save()

User --> UserViewModel: (return)

UserViewModel --> UserView: (return)

UserView --> Client: (return)

#### Сценарий: Чтение пользователя

1. Пользователь запрашивает информацию о пользователе.
2. View передает запрос ViewModel.
3. ViewModel получает данные из модели (User).
4. ViewModel обновляет View через привязку данных.

**Код**:

ruby

Copy

# Модель

class User

attr\_accessor :name, :email

def self.find(id)

# Заглушка для поиска пользователя

new.tap do |user|

user.name = "Alice"

user.email = "alice@example.com"

end

end

end

# ViewModel

class UserViewModel

attr\_accessor :name, :email

def show\_user(id)

user = User.find(id)

@name = user.name

@email = user.email

end

end

# Представление

class UserView

def initialize(view\_model)

@view\_model = view\_model

end

def display\_user(id)

@view\_model.show\_user(id)

puts "Name: #{@view\_model.name}, Email: #{@view\_model.email}"

end

end

# Пример использования

view\_model = UserViewModel.new

view = UserView.new(view\_model)

view.display\_user(1)

**Диаграмма последовательности**:

plaintext

Copy

Client -> UserView: display\_user(1)

UserView -> UserViewModel: show\_user(1)

UserViewModel -> User: find(1)

User --> UserViewModel: (return)

UserViewModel --> UserView: (return)

UserView --> Client: (return)

### Фреймворки, использующие MVVM

1. **WPF (Windows Presentation Foundation)**:
   * MVVM широко используется в разработке desktop-приложений на платформе .NET.
   * Пример: Библиотека Prism.
2. **Xamarin.Forms**:
   * MVVM используется для разработки мобильных приложений на платформе Xamarin.
   * Пример: Библиотека MVVM Light.
3. **Angular**:
   * MVVM используется для разработки веб-приложений на платформе Angular.
   * Пример: Компоненты и сервисы Angular.
4. **Vue.js**:
   * MVVM используется для разработки веб-приложений на платформе Vue.js.
   * Пример: Реактивные свойства и методы Vue.js.

**1. Неуместная близость (Inappropriate Intimacy)**

**Описание**: Классы слишком тесно взаимодействуют друг с другом, нарушая инкапсуляцию.

**Пример**:

ruby

Copy

class User

attr\_accessor :name, :wallet

def initialize(name)

@name = name

@wallet = Wallet.new

end

end

class Wallet

attr\_accessor :balance

def initialize

@balance = 0

end

def add\_money(amount)

@balance += amount

end

end

class PaymentService

def pay(user, amount)

user.wallet.add\_money(amount) # Нарушение инкапсуляции

end

end

**Рефакторинг**:

* Используем принцип "Tell, Don't Ask". Класс User должен управлять своим кошельком.

ruby

Copy

class User

attr\_accessor :name

def initialize(name)

@name = name

@wallet = Wallet.new

end

def add\_money\_to\_wallet(amount)

@wallet.add\_money(amount)

end

end

class PaymentService

def pay(user, amount)

user.add\_money\_to\_wallet(amount) # Делегирование ответственности

end

end

**2. Завистливые функции (Feature Envy)**

**Описание**: Метод одного класса чрезмерно использует данные или методы другого класса.

**Пример**:

ruby

Copy

class Customer

attr\_accessor :name, :address

def initialize(name, address)

@name = name

@address = address

end

end

class Order

def get\_shipping\_label(customer)

"#{customer.address.street}, #{customer.address.city}" # Зависть к данным Customer

end

end

**Рефакторинг**:

* Переместите метод в класс, которому принадлежат данные.

ruby

Copy

class Customer

attr\_accessor :name, :address

def initialize(name, address)

@name = name

@address = address

end

def shipping\_label

"#{address.street}, #{address.city}"

end

end

class Order

def get\_shipping\_label(customer)

customer.shipping\_label # Делегирование

end

end

**Паттерны для предотвращения**:

* **Принцип единственной ответственности (SRP)**: Класс должен отвечать только за одну задачу.
* **Делегирование**: Передача ответственности за выполнение задачи соответствующему классу.

**3. Классы данных (Data Class)**

**Описание**: Класс, который содержит только данные и не имеет поведения.

**Пример**:

ruby

Copy

class Person

attr\_accessor :name, :age

end

**Рефакторинг**:

* Добавьте методы для работы с данными.

ruby

Copy

class Person

attr\_accessor :name, :age

def adult?

age >= 18

end

end

**Паттерны для предотвращения**:

* **Инкапсуляция**: Скрывайте данные и предоставляйте методы для работы с ними.
* **Объектно-ориентированное проектирование**: Классы должны содержать как данные, так и поведение.

**4. Одержимость элементарными типами (Primitive Obsession)**

**Описание**: Использование примитивных типов вместо создания специализированных классов.

**Пример**:

ruby

Copy

class Order

attr\_accessor :customer\_name, :customer\_address, :product\_name, :price

end

**Рефакторинг**:

* Создайте классы для представления сложных данных.

ruby

Copy

class Customer

attr\_accessor :name, :address

end

class Product

attr\_accessor :name, :price

end

class Order

attr\_accessor :customer, :product

end

**Паттерны для предотвращения**:

* **Объектно-ориентированное проектирование**: Используйте классы для представления сложных данных.
* **Value Object**: Паттерн для представления неизменяемых объектов данных.

**5. Ленивый класс (Lazy Class)**

**Описание**: Класс, который не выполняет достаточно функций, чтобы оправдать свое существование.

**Пример**:

ruby

Copy

class Calculator

def add(a, b)

a + b

end

end

**Рефакторинг**:

* Удалите класс и переместите его функциональность в другой класс.

ruby

Copy

module MathUtils

def self.add(a, b)

a + b

end

end

**Паттерны для предотвращения**:

* **Принцип KISS (Keep It Simple, Stupid)**: Не создавайте избыточных сущностей.
* **Рефакторинг "Inline Class"**: Объедините класс с другим, если он не выполняет достаточно функций.

**1. Дублирование кода (Duplicate Code)**

**Описание**: Один и тот же код повторяется в нескольких местах.

**Почему это антипаттерн**: Дублирование усложняет поддержку, так как изменения нужно вносить в нескольких местах, что увеличивает вероятность ошибок.

**Пример**:

ruby

Copy

def calculate\_area\_of\_circle(radius)

Math::PI \* radius \*\* 2

end

def calculate\_area\_of\_sphere(radius)

4 \* Math::PI \* radius \*\* 2

end

**Рефакторинг**:

* Вынесите общую логику в отдельный метод.

ruby

Copy

def calculate\_area(radius, multiplier)

multiplier \* Math::PI \* radius \*\* 2

end

def calculate\_area\_of\_circle(radius)

calculate\_area(radius, 1)

end

def calculate\_area\_of\_sphere(radius)

calculate\_area(radius, 4)

end

**Паттерны для предотвращения**:

* **DRY (Don't Repeat Yourself)**: Избегайте дублирования кода.
* **Методы и модули**: Выносите общую логику в методы или модули.

**2. Длинный метод (Long Method)**

**Описание**: Метод содержит слишком много строк кода и выполняет несколько задач.

**Почему это антипаттерн**: Длинные методы сложно читать, тестировать и поддерживать.

**Пример**:

ruby

Copy

def process\_order(order)

# Проверка наличия товара

if order.items.any? { |item| item.stock == 0 }

raise "Товара нет в наличии"

end

# Расчет стоимости

total = order.items.sum(&:price)

# Применение скидки

if order.customer.loyalty\_level == :gold

total \*= 0.9

end

# Отправка уведомления

send\_email(order.customer.email, "Ваш заказ на сумму #{total} оформлен")

end

**Рефакторинг**:

* Разделите метод на несколько более мелких методов.

ruby

Copy

def process\_order(order)

check\_stock(order)

total = calculate\_total(order)

send\_notification(order, total)

end

def check\_stock(order)

raise "Товара нет в наличии" if order.items.any? { |item| item.stock == 0 }

end

def calculate\_total(order)

total = order.items.sum(&:price)

total \*= 0.9 if order.customer.loyalty\_level == :gold

total

end

def send\_notification(order, total)

send\_email(order.customer.email, "Ваш заказ на сумму #{total} оформлен")

end

**3. Длинный класс (Large Class)**

**Описание**: Класс содержит слишком много методов и полей, выполняя множество задач.

**Почему это антипаттерн**: Класс становится сложным для понимания и поддержки.

**Пример**:

ruby

Copy

class OrderProcessor

def initialize(order)

@order = order

end

def check\_stock

# Логика проверки наличия товара

end

def calculate\_total

# Логика расчета стоимости

end

def apply\_discount

# Логика применения скидки

end

def send\_notification

# Логика отправки уведомления

end

def process\_payment

# Логика обработки платежа

end

end

**Рефакторинг**:

* Разделите класс на несколько более мелких классов.

ruby

Copy

class StockChecker

def check(order)

# Логика проверки наличия товара

end

end

class TotalCalculator

def calculate(order)

# Логика расчета стоимости

end

end

class NotificationSender

def send(order)

# Логика отправки уведомления

end

end

**Паттерны для предотвращения**:

* **Принцип единственной ответственности (SRP)**: Класс должен выполнять только одну задачу.
* **Разделение на классы**: Разделяйте код на небольшие классы с понятными обязанностями.

**4. Длинный список параметров (Long Parameter List)**

**Описание**: Метод принимает слишком много параметров.

**Почему это антипаттерн**: Длинные списки параметров сложно читать и использовать.

**Пример**:

ruby

Copy

def create\_user(name, email, password, address, phone, date\_of\_birth)

# Логика создания пользователя

end

**Рефакторинг**:

* Замените параметры объектом.

ruby

Copy

class UserParams

attr\_accessor :name, :email, :password, :address, :phone, :date\_of\_birth

end

def create\_user(params)

# Логика создания пользователя

end

**Паттерны для предотвращения**:

* **Объекты параметров**: Используйте объекты для группировки связанных параметров.
* **Builder**: Паттерн для пошагового создания сложных объектов.

**5. Повторы условных операторов (Conditional Complexity)**

**Описание**: В коде много повторяющихся или сложных условных операторов.

**Почему это антипаттерн**: Усложняет чтение и поддержку кода.

**Пример**:

ruby

Copy

def calculate\_discount(customer)

if customer.loyalty\_level == :gold

0.2

elsif customer.loyalty\_level == :silver

0.1

else

0

end

end

**Рефакторинг**:

* Используйте полиморфизм или хэши для упрощения логики.

ruby

Copy

DISCOUNTS = { gold: 0.2, silver: 0.1, default: 0 }

def calculate\_discount(customer)

DISCOUNTS[customer.loyalty\_level] || DISCOUNTS[:default]

end

**Паттерны для предотвращения**:

* **Полиморфизм**: Используйте наследование или модули для обработки различных случаев.
* **Хэши или конфигурации**: Храните данные в структурированном виде.

**6. Альтернативные классы с разными интерфейсами (Alternative Classes with Different Interfaces)**

**Описание**: Классы выполняют схожие задачи, но имеют разные интерфейсы.

**Почему это антипаттерн**: Усложняет использование и расширение системы.

**Пример**:

ruby

Copy

class Circle

def area(radius)

Math::PI \* radius \*\* 2

end

end

class Square

def calculate\_area(side)

side \*\* 2

end

end

**Рефакторинг**:

* Унифицируйте интерфейсы классов.

ruby

Copy

class Circle

def area(radius)

Math::PI \* radius \*\* 2

end

end

class Square

def area(side)

side \*\* 2

end

end

**Паттерны для предотвращения**:

* **Интерфейсы или модули**: Используйте общие интерфейсы для схожих классов.
* **Адаптер**: Паттерн для унификации интерфейсов.

**1. Стрела дробью (Arrow Anti-Pattern)**

**Описание**: Код с глубокой вложенностью условных операторов или циклов, напоминающий стрелу (if внутри if внутри if и т.д.).

**Почему это антипаттерн**:

* Сложно читать и понимать.
* Высокая цикломатическая сложность.
* Повышает вероятность ошибок.

**Пример**:

ruby

Copy

def process\_order(order)

if order.valid?

if order.payment\_received?

if order.items\_in\_stock?

order.ship

else

raise "Товара нет в наличии"

end

else

raise "Оплата не получена"

end

else

raise "Неверный заказ"

end

end

**Рефакторинг**:

* Используйте "ранний возврат" (early return) или guard clauses.

ruby

Copy

def process\_order(order)

raise "Неверный заказ" unless order.valid?

raise "Оплата не получена" unless order.payment\_received?

raise "Товара нет в наличии" unless order.items\_in\_stock?

order.ship

end

**Паттерны для предотвращения**:

* **Guard Clauses**: Упрощают логику, уменьшая вложенность.
* **Принцип KISS (Keep It Simple, Stupid)**: Избегайте сложных конструкций.

**2. Расходящиеся модификации (Divergent Change)**

**Описание**: Класс изменяется по разным причинам в разных местах, что указывает на нарушение принципа единственной ответственности (SRP).

**Почему это антипаттерн**:

* Класс становится сложным для поддержки.
* Изменения в одной части кода могут повлиять на другие, не связанные части.

**Пример**:

ruby

Copy

class Order

def calculate\_total

# Логика расчета стоимости

end

def send\_notification

# Логика отправки уведомления

end

def save\_to\_database

# Логика сохранения в базу данных

end

end

**Рефакторинг**:

* Разделите класс на несколько классов, каждый с одной ответственностью.

ruby

Copy

class OrderCalculator

def calculate\_total(order)

# Логика расчета стоимости

end

end

class OrderNotifier

def send\_notification(order)

# Логика отправки уведомления

end

end

class OrderRepository

def save(order)

# Логика сохранения в базу данных

end

end

**Паттерны для предотвращения**:

* **Принцип единственной ответственности (SRP)**: Класс должен выполнять только одну задачу.
* **Разделение на классы**: Разделяйте код на небольшие классы с понятными обязанностями.

**3. Посредник (Middle Man)**

**Описание**: Класс, который делегирует почти все свои методы другому классу, не добавляя собственной логики.

**Почему это антипаттерн**:

* Лишний уровень абстракции, который усложняет код.
* Нарушает принцип "Tell, Don't Ask".

**Пример**:

ruby

Copy

class OrderProcessor

def initialize(order)

@order = order

end

def calculate\_total

@order.calculate\_total

end

def ship

@order.ship

end

end

**Рефакторинг**:

* Удалите посредника и работайте напрямую с целевым классом.

ruby

Copy

order = Order.new

order.calculate\_total

order.ship

**Паттерны для предотвращения**:

* **Принцип KISS**: Избегайте избыточных абстракций.
* **Делегирование**: Используйте только там, где это действительно необходимо.

**4. Теоретическая общность (Speculative Generality)**

**Описание**: Добавление избыточной абстракции или функциональности "на будущее", которая никогда не используется.

**Почему это антипаттерн**:

* Усложняет код без необходимости.
* Затрудняет понимание и поддержку.

**Пример**:

ruby

Copy

class Animal

def eat

# Логика для еды

end

def sleep

# Логика для сна

end

def fly

# Логика для полета (но не все животные летают)

end

end

**Рефакторинг**:

* Удалите неиспользуемую функциональность.

ruby

Copy

class Animal

def eat

# Логика для еды

end

def sleep

# Логика для сна

end

end

class Bird < Animal

def fly

# Логика для полета

end

end

**Паттерны для предотвращения**:

* **YAGNI (You Aren't Gonna Need It)**: Не добавляйте функциональность, пока она не понадобится.
* **Рефакторинг "Удаление лишнего"**: Удаляйте неиспользуемый код.

**5. Параллельные иерархии наследования (Parallel Inheritance Hierarchies)**

**Описание**: Две иерархии классов тесно связаны, и изменение в одной требует изменения в другой.

**Почему это антипаттерн**:

* Высокая связанность между иерархиями.
* Усложняет поддержку и расширение.

**Пример**:

ruby

Copy

class Vehicle

# Общая логика для транспортных средств

end

class Car < Vehicle

# Логика для автомобилей

end

class Truck < Vehicle

# Логика для грузовиков

end

class Driver

# Общая логика для водителей

end

class CarDriver < Driver

# Логика для водителей автомобилей

end

class TruckDriver < Driver

# Логика для водителей грузовиков

end

**Рефакторинг**:

* Используйте композицию вместо наследования.

ruby

Copy

class Vehicle

# Общая логика для транспортных средств

end

class Car < Vehicle

# Логика для автомобилей

end

class Truck < Vehicle

# Логика для грузовиков

end

class Driver

attr\_accessor :vehicle

def initialize(vehicle)

@vehicle = vehicle

end

end

**Паттерны для предотвращения**:

* **Композиция вместо наследования**: Используйте композицию для уменьшения связанности.
* **Принцип открытости/закрытости (OCP)**: Классы должны быть открыты для расширения, но закрыты для модификации.

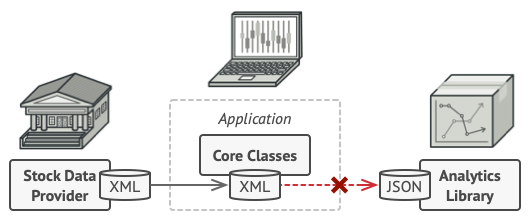
**Взаимосвязь антипаттернов**

1. **Стрела дробью** и **Расходящиеся модификации**:
   * Оба связаны с нарушением SRP. Стрела дробью возникает из-за сложной логики, а расходящиеся модификации — из-за множества причин для изменения класса.
2. **Посредник** и **Теоретическая общность**:
   * Оба связаны с избыточной абстракцией. Посредник добавляет ненужный уровень делегирования, а теоретическая общность — ненужную функциональность.
3. **Параллельные иерархии наследования**:
   * Возникает из-за тесной связанности между классами, что может быть следствием неправильного использования наследования вместо композиции.

ЕЩЕ МОИ ЛЮБИМЫЕ ПАТТЕРНЫ)  
**Адаптер** — это структурный паттерн, который позволяет подружить несовместимые объекты.

Представьте, что вы делаете приложение для торговли на бирже. Ваше приложение скачивает биржевые котировки из нескольких источников в XML, а затем рисует красивые графики.

В какой-то момент вы решаете улучшить приложение, применив стороннюю библиотеку аналитики. Но вот беда — библиотека поддерживает только формат данных JSON, несовместимый с вашим приложением.



Подключить стороннюю библиотеку не выйдет из-за несовместимых форматов данных.

Вы смогли бы переписать библиотеку, чтобы та поддерживала формат XML. Но, во-первых, это может нарушить работу существующего кода, который уже зависит от библиотеки. А во-вторых, у вас может просто не быть доступа к её исходному коду.

## Решение

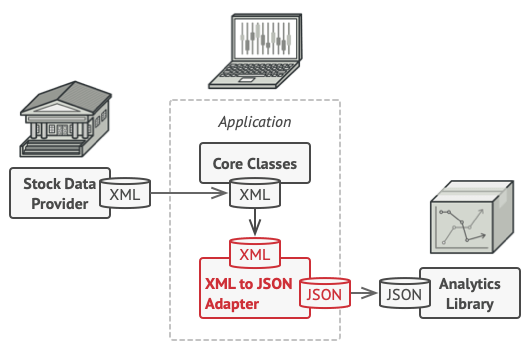
Вы можете создать адаптер. Это объект-переводчик, который трансформирует интерфейс или данные одного объекта в такой вид, чтобы он стал понятен другому объекту.

При этом адаптер оборачивает один из объектов, так что другой объект даже не знает о наличии первого. Например, вы можете обернуть объект, работающий в метрах, адаптером, который бы конвертировал данные в футы.

Адаптеры могут не только переводить данные из одного формата в другой, но и помогать объектам с разными интерфейсами работать сообща. Это работает так:

1. Адаптер имеет интерфейс, который совместим с одним из объектов.
2. Поэтому этот объект может свободно вызывать методы адаптера.
3. Адаптер получает эти вызовы и перенаправляет их второму объекту, но уже в том формате и последовательности, которые понятны второму объекту.

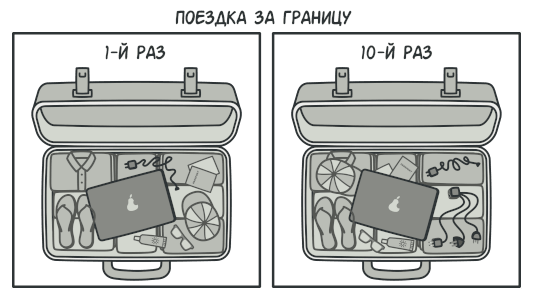
Иногда возможно создать даже двухсторонний адаптер, который работал бы в обе стороны.



Программа может работать со сторонней библиотекой через адаптер.

Таким образом, в приложении биржевых котировок вы могли бы создать класс XML\_To\_JSON\_Adapter, который бы оборачивал объект того или иного класса библиотеки аналитики. Ваш код посылал бы адаптеру запросы в формате XML, а адаптер сначала транслировал входящие данные в формат JSON, а затем передавал бы их методам обёрнутого объекта аналитики.

## Аналогия из жизни



Содержимое чемоданов до и после поездки за границу.

Когда вы в первый раз летите за границу, вас может ждать сюрприз при попытке зарядить ноутбук. Стандарты розеток в разных странах отличаются. Ваша европейская зарядка будет бесполезна в США без специального адаптера, позволяющего подключиться к розетке другого типа.

Этот пример показывает структуру паттерна **Адаптер**, а именно — из каких классов он состоит, какие роли эти классы выполняют и как они взаимодействуют друг с другом.

#### ****main.rb:**** Пример структуры паттерна

# Целевой класс объявляет интерфейс, с которым может работать клиентский код.

**class** **Target**

# @return [String]

**def** **request**

'Target: The default target\'s behavior.'

**end**

**end**

# Адаптируемый класс содержит некоторое полезное поведение, но его интерфейс

# несовместим с существующим клиентским кодом. Адаптируемый класс нуждается в

# некоторой доработке, прежде чем клиентский код сможет его использовать.

**class** **Adaptee**

# @return [String]

**def** **specific\_request**

'.eetpadA eht fo roivaheb laicepS'

**end**

**end**

# Адаптер делает интерфейс Адаптируемого класса совместимым с целевым

# интерфейсом.

**class** **Adapter** < **Target**

# @param [Adaptee] adaptee

**def** **initialize**(adaptee)

@adaptee = adaptee

**end**

**def** **request**

"Adapter: (TRANSLATED) #{@adaptee.specific\_request.reverse!}"

**end**

**end**

# Клиентский код поддерживает все классы, использующие интерфейс Target.

#

# @param [Target] target

**def** **client\_code**(target)

print target.request

**end**

puts 'Client: I can work just fine with the Target objects:'

target = **Target**.new

client\_code(target)

puts "\n\n"

adaptee = **Adaptee**.new

puts 'Client: The Adaptee class has a weird interface. See, I don\'t understand it:'

puts "Adaptee: #{adaptee.specific\_request}"

puts "\n"

puts 'Client: But I can work with it via the Adapter:'

adapter = **Adapter**.new(adaptee)

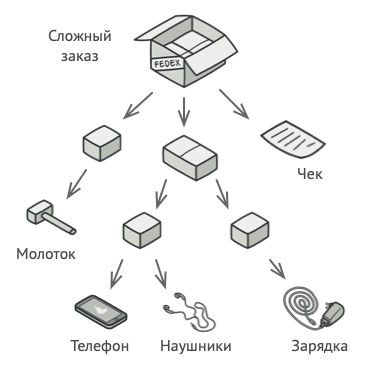
client\_code(adapter)

**Компоновщик** — это структурный паттерн, который позволяет создавать дерево объектов и работать с ним так же, как и с единичным объектом.

Паттерн Компоновщик имеет смысл только тогда, когда основная модель вашей программы может быть структурирована в виде дерева.

Например, есть два объекта: Продукт и Коробка. Коробка может содержать несколько Продуктов и других Коробок поменьше. Те, в свою очередь, тоже содержат либо Продукты, либо Коробки и так далее.

Теперь предположим, ваши Продукты и Коробки могут быть частью заказов. Каждый заказ может содержать как простые Продукты без упаковки, так и составные Коробки. Ваша задача состоит в том, чтобы узнать цену всего заказа.



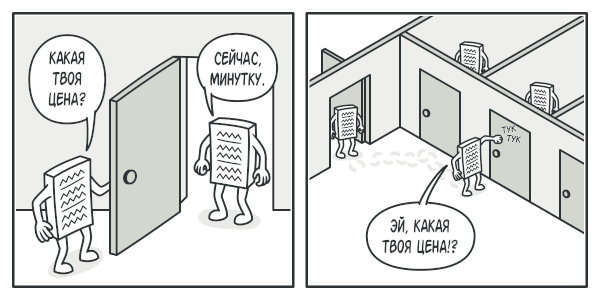
Заказ может состоять из различных продуктов, упакованных в собственные коробки.

Если решать задачу в лоб, то вам потребуется открыть все коробки заказа, перебрать все продукты и посчитать их суммарную стоимость. Но это слишком хлопотно, так как типы коробок и их содержимое могут быть вам неизвестны. Кроме того, наперёд неизвестно и количество уровней вложенности коробок, поэтому перебрать коробки простым циклом не выйдет.

## Решение

Компоновщик предлагает рассматривать Продукт и Коробку через единый интерфейс с общим методом получения стоимости.

Продукт просто вернёт свою цену. Коробка спросит цену каждого предмета внутри себя и вернёт сумму результатов. Если одним из внутренних предметов окажется коробка поменьше, она тоже будет перебирать своё содержимое, и так далее, пока не будут посчитаны все составные части.



Компоновщик рекурсивно запускает действие по всем элементам дерева — от корня к листьям.

Для вас, клиента, главное, что теперь не нужно ничего знать о структуре заказов. Вы вызываете метод получения цены, он возвращает цифру, а вы не тонете в горах картона и скотча.

# Базовый класс Компонент объявляет общие операции как для простых, так и для

# сложных объектов структуры.

#

# @abstract

**class** **Component**

# @return [Component]

**def** **parent**

@parent

**end**

# При необходимости базовый Компонент может объявить интерфейс для установки и

# получения родителя компонента в древовидной структуре. Он также может

# предоставить некоторую реализацию по умолчанию для этих методов.

#

# @param [Component] parent

**def** **parent**=(parent)

@parent = parent

**end**

# В некоторых случаях целесообразно определить операции управления потомками

# прямо в базовом классе Компонент. Таким образом, вам не нужно будет

# предоставлять конкретные классы компонентов клиентскому коду, даже во время

# сборки дерева объектов. Недостаток такого подхода в том, что эти методы

# будут пустыми для компонентов уровня листа.

#

# @abstract

#

# @param [Component] component

**def** **add**(component)

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

# @abstract

#

# @param [Component] component

**def** **remove**(component)

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

# Вы можете предоставить метод, который позволит клиентскому коду понять,

# может ли компонент иметь вложенные объекты.

#

# @return [Boolean]

**def** **composite?**

**false**

**end**

# Базовый Компонент может сам реализовать некоторое поведение по умолчанию или

# поручить это конкретным классам, объявив метод, содержащий поведение

# абстрактным.

#

# @abstract

#

# @return [String]

**def** **operation**

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

**end**

# Класс Лист представляет собой конечные объекты структуры. Лист не может иметь

# вложенных компонентов.

#

# Обычно объекты Листьев выполняют фактическую работу, тогда как объекты

# Контейнера лишь делегируют работу своим подкомпонентам.

**class** **Leaf** < **Component**

# return [String]

**def** **operation**

'Leaf'

**end**

**end**

# Класс Контейнер содержит сложные компоненты, которые могут иметь вложенные

# компоненты. Обычно объекты Контейнеры делегируют фактическую работу своим

# детям, а затем «суммируют» результат.

**class** **Composite** < **Component**

**def** **initialize**

@children = []

**end**

# Объект контейнера может как добавлять компоненты в свой список вложенных

# компонентов, так и удалять их, как простые, так и сложные.

# @param [Component] component

**def** **add**(component)

@children.append(component)

component.parent = **self**

**end**

# @param [Component] component

**def** **remove**(component)

@children.remove(component)

component.parent = **nil**

**end**

# @return [Boolean]

**def** **composite?**

**true**

**end**

# Контейнер выполняет свою основную логику особым образом. Он проходит

# рекурсивно через всех своих детей, собирая и суммируя их результаты.

# Поскольку потомки контейнера передают эти вызовы своим потомкам и так далее,

# в результате обходится всё дерево объектов.

#

# @return [String]

**def** **operation**

results = []

@children.each { |**child**| results.append(child.operation) }

"Branch(#{results.join('+')})"

**end**

**end**

# Клиентский код работает со всеми компонентами через базовый интерфейс.

#

# @param [Component] component

**def** **client\_code**(component)

puts "RESULT: #{component.operation}"

**end**

# Благодаря тому, что операции управления потомками объявлены в базовом классе

# Компонента, клиентский код может работать как с простыми, так и со сложными

# компонентами, вне зависимости от их конкретных классов.

#

# @param [Component] component

# @param [Component] component2

**def** **client\_code2**(component1, component2)

component1.add(component2) **if** component1.composite?

print "RESULT: #{component1.operation}"

**end**

# Таким образом, клиентский код может поддерживать простые компоненты-листья...

simple = **Leaf**.new

puts 'Client: I\'ve got a simple component:'

client\_code(simple)

puts "\n"

# ...а также сложные контейнеры.

tree = **Composite**.new

branch1 = **Composite**.new

branch1.add(**Leaf**.new)

branch1.add(**Leaf**.new)

branch2 = **Composite**.new

branch2.add(**Leaf**.new)

tree.add(branch1)

tree.add(branch2)

puts 'Client: Now I\'ve got a composite tree:'

client\_code(tree)

puts "\n"

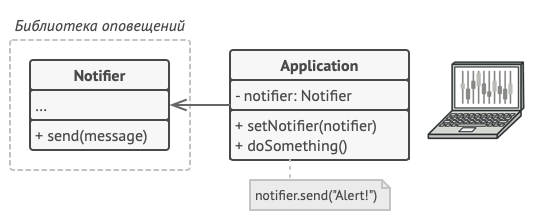
puts 'Client: I don\'t need to check the components classes even when managing the tree:'

client\_code2(tree, simple)

**Декоратор** — это структурный паттерн, который позволяет добавлять объектам новые поведения на лету, помещая их в объекты-обёртки.

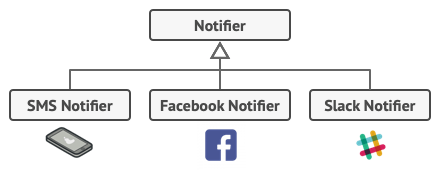
Вы работаете над библиотекой оповещений, которую можно подключать к разнообразным программам, чтобы получать уведомления о важных событиях.

Основой библиотеки является класс Notifier с методом send, который принимает на вход строку-сообщение и высылает её всем администраторам по электронной почте. Сторонняя программа должна создать и настроить этот объект, указав кому отправлять оповещения, а затем использовать его каждый раз, когда что-то случается.



Сторонние программы используют главный класс оповещений.

В какой-то момент стало понятно, что одних email-оповещений пользователям мало. Некоторые из них хотели бы получать извещения о критических проблемах через SMS. Другие хотели бы получать их в виде сообщений Facebook. Корпоративные пользователи хотели бы видеть сообщения в Slack.

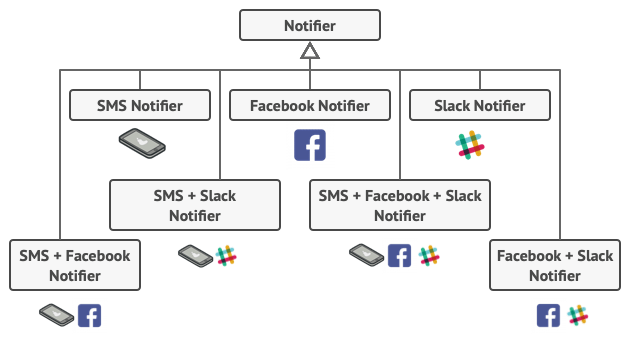


Каждый тип оповещения живёт в собственном подклассе.

Сначала вы добавили каждый из этих типов оповещений в программу, унаследовав их от базового класса Notifier. Теперь пользователь выбирал один из типов оповещений, который и использовался в дальнейшем.

Но затем кто-то резонно спросил, почему нельзя выбрать несколько типов оповещений сразу? Ведь если вдруг в вашем доме начался пожар, вы бы хотели получить оповещения по всем каналам, не так ли?

Вы попытались реализовать все возможные комбинации подклассов оповещений. Но после того как вы добавили первый десяток классов, стало ясно, что такой подход невероятно раздувает код программы.



Комбинаторный взрыв подклассов при совмещении типов оповещений.

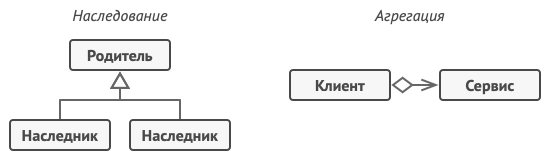
Итак, нужен какой-то другой способ комбинирования поведения объектов, который не приводит к взрыву количества подклассов.

## Решение

Наследование — это первое, что приходит в голову многим программистам, когда нужно расширить какое-то существующее поведение. Но механизм наследования имеет несколько досадных проблем.

* Он **статичен**. Вы не можете изменить поведение существующего объекта. Для этого вам надо создать новый объект, выбрав другой подкласс.
* Он **не разрешает наследовать поведение нескольких классов одновременно**. Из-за этого вам приходится создавать множество подклассов-комбинаций для получения совмещённого поведения.

Одним из способов обойти эти проблемы является замена наследования агрегацией либо композицией . Это когда один объект содержит ссылку на другой и делегирует ему работу, вместо того чтобы самому наследовать его поведение. Как раз на этом принципе построен паттерн Декоратор.

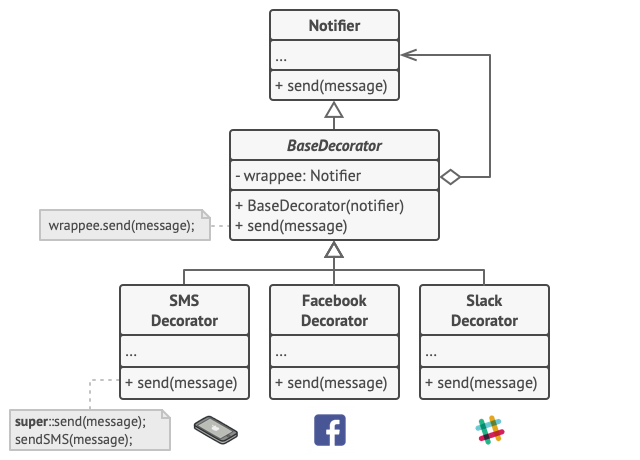


Наследование против Агрегации.

Декоратор имеет альтернативное название — обёртка. Оно более точно описывает суть паттерна: вы помещаете целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё.

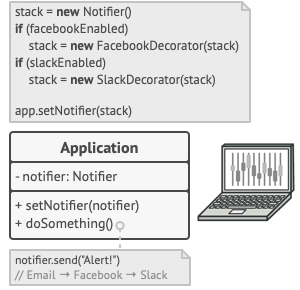
Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать — чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно — результат будет иметь объединённое поведение всех обёрток сразу.

В примере с оповещениями мы оставим в базовом классе простую отправку по электронной почте, а расширенные способы отправки сделаем декораторами.



Расширенные способы оповещения становятся декораторами.

Сторонняя программа, выступающая клиентом, во время первичной настройки будет заворачивать объект оповещений в те обёртки, которые соответствуют желаемому способу оповещения.



Программа может составлять составные объекты из декораторов.

Последняя обёртка в списке и будет тем объектом, с которым клиент будет работать в остальное время. Для остального клиентского кода, по сути, ничего не изменится, ведь все обёртки имеют точно такой же интерфейс, что и базовый класс оповещений.

Таким же образом можно изменять не только способ доставки оповещений, но и форматирование, список адресатов и так далее. К тому же клиент может «дообернуть» объект любыми другими обёртками, когда ему захочется.

# Базовый интерфейс Компонента определяет поведение, которое изменяется

# декораторами.

**class** **Component**

# @return [String]

**def** **operation**

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

**end**

# Конкретные Компоненты предоставляют реализации поведения по умолчанию. Может

# быть несколько вариаций этих классов.

**class** **ConcreteComponent** < **Component**

# @return [String]

**def** **operation**

'ConcreteComponent'

**end**

**end**

# Базовый класс Декоратора следует тому же интерфейсу, что и другие компоненты.

# Основная цель этого класса - определить интерфейс обёртки для всех конкретных

# декораторов. Реализация кода обёртки по умолчанию может включать в себя поле

# для хранения завёрнутого компонента и средства его инициализации.

**class** **Decorator** < **Component**

attr\_accessor **:component**

# @param [Component] component

**def** **initialize**(component)

@component = component

**end**

# Декоратор делегирует всю работу обёрнутому компоненту.

#

# @return [String]

**def** **operation**

@component.operation

**end**

**end**

# Конкретные Декораторы вызывают обёрнутый объект и изменяют его результат

# некоторым образом.

**class** **ConcreteDecoratorA** < **Decorator**

# Декораторы могут вызывать родительскую реализацию операции, вместо того,

# чтобы вызвать обёрнутый объект напрямую. Такой подход упрощает расширение

# классов декораторов.

#

# @return [String]

**def** **operation**

"ConcreteDecoratorA(#{@component.operation})"

**end**

**end**

# Декораторы могут выполнять своё поведение до или после вызова обёрнутого

# объекта.

**class** **ConcreteDecoratorB** < **Decorator**

# @return [String]

**def** **operation**

"ConcreteDecoratorB(#{@component.operation})"

**end**

**end**

# Клиентский код работает со всеми объектами, используя интерфейс Компонента.

# Таким образом, он остаётся независимым от конкретных классов компонентов, с

# которыми работает.

#

# @param [Component] component

**def** **client\_code**(component)

# ...

print "RESULT: #{component.operation}"

# ...

**end**

# Таким образом, клиентский код может поддерживать как простые компоненты...

simple = **ConcreteComponent**.new

puts 'Client: I\'ve got a simple component:'

client\_code(simple)

puts "\n\n"

# ...так и декорированные.

#

# Обратите внимание, что декораторы могут обёртывать не только простые

# компоненты, но и другие декораторы.

decorator1 = **ConcreteDecoratorA**.new(simple)

decorator2 = **ConcreteDecoratorB**.new(decorator1)

puts 'Client: Now I\'ve got a decorated component:'

client\_code(decorator2)

## **Фасад** — это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.  Проблема

Вашему коду приходится работать с большим количеством объектов некой сложной библиотеки или фреймворка. Вы должны самостоятельно инициализировать эти объекты, следить за правильным порядком зависимостей и так далее.

В результате бизнес-логика ваших классов тесно переплетается с деталями реализации сторонних классов. Такой код довольно сложно понимать и поддерживать.

## Решение

Фасад — это простой интерфейс для работы со сложной подсистемой, содержащей множество классов. Фасад может иметь урезанный интерфейс, не имеющий 100% функциональности, которой можно достичь, используя сложную подсистему напрямую. Но он предоставляет именно те фичи, которые нужны клиенту, и скрывает все остальные.

Фасад полезен, если вы используете какую-то сложную библиотеку со множеством подвижных частей, но вам нужна только часть её возможностей.

К примеру, программа, заливающая видео котиков в социальные сети, может использовать профессиональную библиотеку сжатия видео. Но все, что нужно клиентскому коду этой программы — простой метод encode(filename, format). Создав класс с таким методом, вы реализуете свой первый фасад.

# Класс Фасада предоставляет простой интерфейс для сложной логики одной или

# нескольких подсистем. Фасад делегирует запросы клиентов соответствующим

# объектам внутри подсистемы. Фасад также отвечает за управление их жизненным

# циклом. Все это защищает клиента от нежелательной сложности подсистемы.

**class** **Facade**

# В зависимости от потребностей вашего приложения вы можете предоставить

# Фасаду существующие объекты подсистемы или заставить Фасад создать их

# самостоятельно.

#

# @param [Subsystem1] subsystem1

# @param [Subsystem2] subsystem2

**def** **initialize**(subsystem1, subsystem2)

@subsystem1 = subsystem1 || **Subsystem1**.new

@subsystem2 = subsystem2 || **Subsystem2**.new

**end**

# Методы Фасада удобны для быстрого доступа к сложной функциональности

# подсистем. Однако клиенты получают только часть возможностей подсистемы.

#

# @return [String]

**def** **operation**

results = []

results.append('Facade initializes subsystems:')

results.append(@subsystem1.operation1)

results.append(@subsystem2.operation1)

results.append('Facade orders subsystems to perform the action:')

results.append(@subsystem1.operation\_n)

results.append(@subsystem2.operation\_z)

results.join("\n")

**end**

**end**

# Подсистема может принимать запросы либо от фасада, либо от клиента напрямую. В

# любом случае, для Подсистемы Фасад – это ещё один клиент, и он не является

# частью Подсистемы.

**class** **Subsystem1**

# @return [String]

**def** **operation1**

'Subsystem1: Ready!'

**end**

# ...

# @return [String]

**def** **operation\_n**

'Subsystem1: Go!'

**end**

**end**

# Некоторые фасады могут работать с разными подсистемами одновременно.

**class** **Subsystem2**

# @return [String]

**def** **operation1**

'Subsystem2: Get ready!'

**end**

# ...

# @return [String]

**def** **operation\_z**

'Subsystem2: Fire!'

**end**

**end**

# Клиентский код работает со сложными подсистемами через простой интерфейс,

# предоставляемый Фасадом. Когда фасад управляет жизненным циклом подсистемы,

# клиент может даже не знать о существовании подсистемы. Такой подход позволяет

# держать сложность под контролем.

#

# @param [Facade] facade

**def** **client\_code**(facade)

print facade.operation

**end**

# В клиентском коде могут быть уже созданы некоторые объекты подсистемы. В этом

# случае может оказаться целесообразным инициализировать Фасад с этими объектами

# вместо того, чтобы позволить Фасаду создавать новые экземпляры.

subsystem1 = **Subsystem1**.new

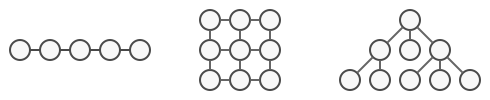
subsystem2 = **Subsystem2**.new

facade = **Facade**.new(subsystem1, subsystem2)

client\_code(facade)

**Итератор** — это поведенческий паттерн, позволяющий последовательно обходить сложную коллекцию, без раскрытия деталей её реализации.

Коллекции — самая распространённая структура данных, которую вы можете встретить в программировании. Это набор объектов, собранный в одну кучу по каким-то критериям.

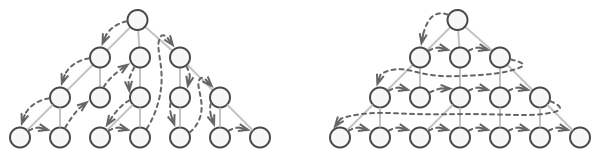


Разные типы коллекций.

Большинство коллекций выглядят как обычный список элементов. Но есть и экзотические коллекции, построенные на основе деревьев, графов и других сложных структур данных.

Но как бы ни была структурирована коллекция, пользователь должен иметь возможность последовательно обходить её элементы, чтобы проделывать с ними какие-то действия.

Но каким способом следует перемещаться по сложной структуре данных? Например, сегодня может быть достаточным обход дерева в глубину, но завтра потребуется возможность перемещаться по дереву в ширину. А на следующей неделе и того хуже — понадобится обход коллекции в случайном порядке.

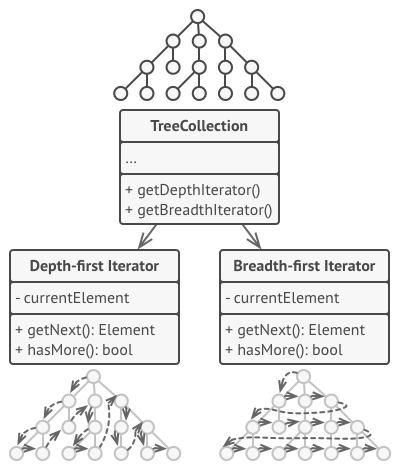


Одну и ту же коллекцию можно обходить разными способами.

Добавляя всё новые алгоритмы в код коллекции, вы понемногу размываете её основную задачу, которая заключается в эффективном хранении данных. Некоторые алгоритмы могут быть и вовсе слишком «заточены» под определённое приложение и смотреться дико в общем классе коллекции.

## Решение

Идея паттерна Итератор состоит в том, чтобы вынести поведение обхода коллекции из самой коллекции в отдельный класс.



Итераторы содержат код обхода коллекции. Одну коллекцию могут обходить сразу несколько итераторов.

Объект-итератор будет отслеживать состояние обхода, текущую позицию в коллекции и сколько элементов ещё осталось обойти. Одну и ту же коллекцию смогут одновременно обходить различные итераторы, а сама коллекция не будет даже знать об этом.

К тому же, если вам понадобится добавить новый способ обхода, вы сможете создать отдельный класс итератора, не изменяя существующий код коллекции.

**class** **AlphabeticalOrderIterator**

# Примесь Enumerable в Ruby предоставляет классы методами обхода, поиска и

# сортировки значений. Класс, реализующий Enumerable должен определить метод

# `each`, который возвращает (в yield) последовательно элементы коллекции.

include **Enumerable**

# Этот атрибут указывает направление обхода.

# @return [Boolean]

attr\_accessor **:reverse**

**private** **:reverse**

# @return [Array]

attr\_accessor **:collection**

**private** **:collection**

# @param [Array] collection

# @param [Boolean] reverse

**def** **initialize**(collection, **reverse:** **false**)

@collection = collection

@reverse = reverse

**end**

**def** **each**(&block)

**return** @collection.reverse.each(&block) **if** reverse

@collection.each(&block)

**end**

**end**

**class** **WordsCollection**

# @return [Array]

attr\_accessor **:collection**

**private** **:collection**

**def** **initialize**(collection = [])

@collection = collection

**end**

# Метод iterator возвращает объект итератора, по умолчанию мы возвращаем

# итератор с сортировкой по возрастанию.

#

# @return [AlphabeticalOrderIterator]

**def** **iterator**

**AlphabeticalOrderIterator**.new(@collection)

**end**

# @return [AlphabeticalOrderIterator]

**def** **reverse\_iterator**

**AlphabeticalOrderIterator**.new(@collection, **reverse:** **true**)

**end**

# @param [String] item

**def** **add\_item**(item)

@collection << item

**end**

**end**

# Клиентский код может знать или не знать о Конкретном Итераторе или классах

# Коллекций, в зависимости от уровня косвенности, который вы хотите сохранить в

# своей программе.

collection = **WordsCollection**.new

collection.add\_item('First')

collection.add\_item('Second')

collection.add\_item('Third')

puts 'Straight traversal:'

collection.iterator.each { |**item**| puts item }

puts "\n"

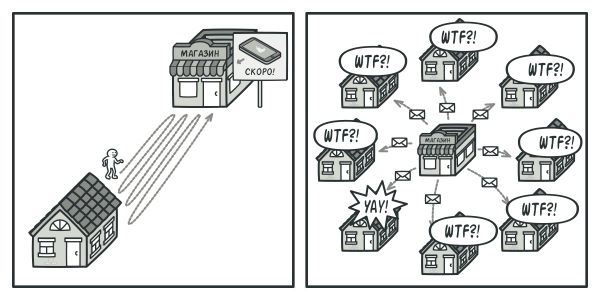
puts 'Reverse traversal:'

collection.reverse\_iterator.each { |**item**| puts item }

**Наблюдатель** — это поведенческий паттерн, который позволяет объектам оповещать другие объекты об изменениях своего состояния.

Представьте, что вы имеете два объекта: Покупатель и Магазин. В магазин вот-вот должны завезти новый товар, который интересен покупателю.

Покупатель может каждый день ходить в магазин, чтобы проверить наличие товара. Но при этом он будет злиться, без толку тратя своё драгоценное время.



Постоянное посещение магазина или спам?

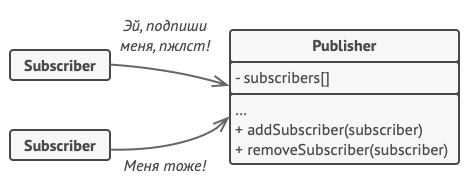
С другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический, и не всем он нужен.

Получается конфликт: либо покупатель тратит время на периодические проверки, либо магазин тратит ресурсы на бесполезные оповещения.

## Решение

Давайте называть Издателями те объекты, которые содержат важное или интересное для других состояние. Остальные объекты, которые хотят отслеживать изменения этого состояния, назовём Подписчиками.

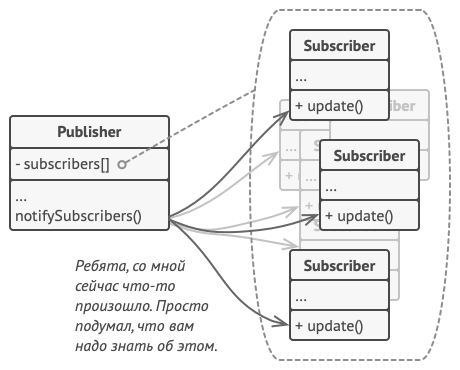
Паттерн Наблюдатель предлагает хранить внутри объекта издателя список ссылок на объекты подписчиков, причём издатель не должен вести список подписки самостоятельно. Он предоставит методы, с помощью которых подписчики могли бы добавлять или убирать себя из списка.



Подписка на события.

Теперь самое интересное. Когда в издателе будет происходить важное событие, он будет проходиться по списку подписчиков и оповещать их об этом, вызывая определённый метод объектов-подписчиков.

Издателю безразлично, какой класс будет иметь тот или иной подписчик, так как все они должны следовать общему интерфейсу и иметь единый метод оповещения.



Оповещения о событиях.

Увидев, как складно всё работает, вы можете выделить общий интерфейс, описывающий методы подписки и отписки, и для всех издателей. После этого подписчики смогут работать с разными типами издателей, а также получать оповещения от них через один и тот же метод.

# Интерфейс издателя объявляет набор методов для управлениями подписчиками.

#

# @abstract

**class** **Subject**

# Присоединяет наблюдателя к издателю.

#

# @abstract

#

# @param [Observer] observer

**def** **attach**(observer)

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

# Отсоединяет наблюдателя от издателя.

#

# @abstract

#

# @param [Observer] observer

**def** **detach**(observer)

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

# Уведомляет всех наблюдателей о событии.

#

# @abstract

**def** **notify**

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

**end**

# Издатель владеет некоторым важным состоянием и оповещает наблюдателей о его

# изменениях.

**class** **ConcreteSubject** < **Subject**

# Для удобства в этой переменной хранится состояние Издателя, необходимое всем

# подписчикам.

attr\_accessor **:state**

# @!attribute observers

# @return [Array<Observer>] attr\_accessor :observers private :observers

**def** **initialize**

@observers = []

**end**

# Список подписчиков. В реальной жизни список подписчиков может храниться в

# более подробном виде (классифицируется по типу события и т.д.)

# @param [Obserser] observer

**def** **attach**(observer)

puts 'Subject: Attached an observer.'

@observers << observer

**end**

# @param [Obserser] observer

**def** **detach**(observer)

@observers.delete(observer)

**end**

# Методы управления подпиской.

# Запуск обновления в каждом подписчике.

**def** **notify**

puts 'Subject: Notifying observers...'

@observers.each { |**observer**| observer.update(**self**) }

**end**

# Обычно логика подписки – только часть того, что делает Издатель. Издатели

# часто содержат некоторую важную бизнес-логику, которая запускает метод

# уведомления всякий раз, когда должно произойти что-то важное (или после

# этого).

**def** **some\_business\_logic**

puts "\nSubject: I'm doing something important."

@state = rand(0..10)

puts "Subject: My state has just changed to: #{@state}"

notify

**end**

**end**

# Интерфейс Наблюдателя объявляет метод уведомления, который издатели используют

# для оповещения своих подписчиков.

#

# @abstract

**class** **Observer**

# Получить обновление от субъекта.

#

# @abstract

#

# @param [Subject] subject

**def** **update**(\_subject)

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

**end**

# Конкретные Наблюдатели реагируют на обновления, выпущенные Издателем, к

# которому они прикреплены.

**class** **ConcreteObserverA** < **Observer**

# @param [Subject] subject

**def** **update**(subject)

puts 'ConcreteObserverA: Reacted to the event' **if** subject.state < 3

**end**

**end**

**class** **ConcreteObserverB** < **Observer**

# @param [Subject] subject

**def** **update**(subject)

**return** **unless** subject.state.zero? || subject.state >= 2

puts 'ConcreteObserverB: Reacted to the event'

**end**

**end**

# Клиентский код.

subject = **ConcreteSubject**.new

observer\_a = **ConcreteObserverA**.new

subject.attach(observer\_a)

observer\_b = **ConcreteObserverB**.new

subject.attach(observer\_b)

subject.some\_business\_logic

subject.some\_business\_logic

subject.detach(observer\_a)

subject.some\_business\_logic

**Стратегия** — это поведенческий паттерн, выносит набор алгоритмов в собственные классы и делает их взаимозаменимыми.

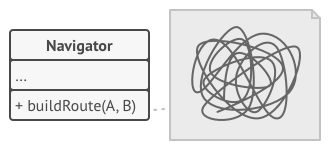
Вы решили написать приложение-навигатор для путешественников. Оно должно показывать красивую и удобную карту, позволяющую с лёгкостью ориентироваться в незнакомом городе.

Одной из самых востребованных функций являлся поиск и прокладывание маршрутов. Пребывая в неизвестном ему городе, пользователь должен иметь возможность указать начальную точку и пункт назначения, а навигатор — проложит оптимальный путь.

Первая версия вашего навигатора могла прокладывать маршрут лишь по дорогам, поэтому отлично подходила для путешествий на автомобиле. Но, очевидно, не все ездят в отпуск на машине. Поэтому следующим шагом вы добавили в навигатор прокладывание пеших маршрутов.

Через некоторое время выяснилось, что некоторые люди предпочитают ездить по городу на общественном транспорте. Поэтому вы добавили и такую опцию прокладывания пути.

Но и это ещё не всё. В ближайшей перспективе вы хотели бы добавить прокладывание маршрутов по велодорожкам. А в отдалённом будущем — интересные маршруты посещения достопримечательностей.



Код навигатора становится слишком раздутым.

Если с популярностью навигатора не было никаких проблем, то техническая часть вызывала вопросы и периодическую головную боль. С каждым новым алгоритмом код основного класса навигатора увеличивался вдвое. В таком большом классе стало довольно трудно ориентироваться.

Любое изменение алгоритмов поиска, будь то исправление багов или добавление нового алгоритма, затрагивало основной класс. Это повышало риск сделать ошибку, случайно задев остальной работающий код.

Кроме того, осложнялась командная работа с другими программистами, которых вы наняли после успешного релиза навигатора. Ваши изменения нередко затрагивали один и тот же код, создавая конфликты, которые требовали дополнительного времени на их разрешение.

## Решение

Паттерн Стратегия предлагает определить семейство схожих алгоритмов, которые часто изменяются или расширяются, и вынести их в собственные классы, называемые стратегиями.

Вместо того, чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы. Чтобы сменить алгоритм, вам будет достаточно подставить в контекст другой объект-стратегию.

Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста.



Стратегии построения пути.

В нашем примере каждый алгоритм поиска пути переедет в свой собственный класс. В этих классах будет определён лишь один метод, принимающий в параметрах координаты начала и конца пути, а возвращающий массив точек маршрута.

Хотя каждый класс будет прокладывать маршрут по-своему, для навигатора это не будет иметь никакого значения, так как его работа заключается только в отрисовке маршрута. Навигатору достаточно подать в стратегию данные о начале и конце маршрута, чтобы получить массив точек маршрута в оговорённом формате.

Класс навигатора будет иметь метод для установки стратегии, позволяя изменять стратегию поиска пути на лету. Такой метод пригодится клиентскому коду навигатора, например, переключателям типов маршрутов в пользовательском интерфейсе.

# Контекст определяет интерфейс, представляющий интерес для клиентов.

**class** **Context**

# Контекст хранит ссылку на один из объектов Стратегии. Контекст не знает

# конкретного класса стратегии. Он должен работать со всеми стратегиями через

# интерфейс Стратегии.

# @return [Strategy]

attr\_writer **:strategy**

# Обычно Контекст принимает стратегию через конструктор, а также предоставляет

# сеттер для её изменения во время выполнения.

#

# @param [Strategy] strategy

**def** **initialize**(strategy)

@strategy = strategy

**end**

# Обычно Контекст позволяет заменить объект Стратегии во время выполнения.

#

# @param [Strategy] strategy

**def** **strategy**=(strategy)

@strategy = strategy

**end**

# Вместо того, чтобы самостоятельно реализовывать множественные версии

# алгоритма, Контекст делегирует некоторую работу объекту Стратегии.

**def** **do\_some\_business\_logic**

# ...

puts 'Context: Sorting data using the strategy (not sure how it\'ll do it)'

result = @strategy.do\_algorithm(%w[a b c d e])

print result.join(',')

# ...

**end**

**end**

# Интерфейс Стратегии объявляет операции, общие для всех поддерживаемых версий

# некоторого алгоритма.

#

# Контекст использует этот интерфейс для вызова алгоритма, определённого

# Конкретными Стратегиями.

#

# @abstract

**class** **Strategy**

# @abstract

#

# @param [Array] data

**def** **do\_algorithm**(\_data)

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

**end**

# Конкретные Стратегии реализуют алгоритм, следуя базовому интерфейсу Стратегии.

# Этот интерфейс делает их взаимозаменяемыми в Контексте.

**class** **ConcreteStrategyA** < **Strategy**

# @param [Array] data

#

# @return [Array]

**def** **do\_algorithm**(data)

data.sort

**end**

**end**

**class** **ConcreteStrategyB** < **Strategy**

# @param [Array] data

#

# @return [Array]

**def** **do\_algorithm**(data)

data.sort.reverse

**end**

**end**

# Клиентский код выбирает конкретную стратегию и передаёт её в контекст. Клиент

# должен знать о различиях между стратегиями, чтобы сделать правильный выбор.

context = **Context**.new(**ConcreteStrategyA**.new)

puts 'Client: Strategy is set to normal sorting.'

context.do\_some\_business\_logic

puts "\n\n"

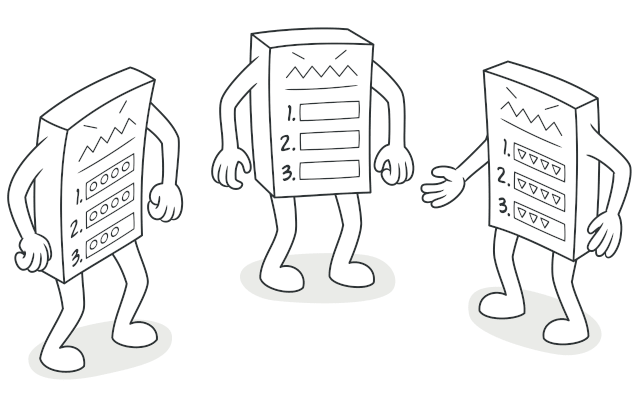
puts 'Client: Strategy is set to reverse sorting.'

context.strategy = **ConcreteStrategyB**.new

context.do\_some\_business\_logic

**Шаблонный метод** — это поведенческий паттерн, задающий скелет алгоритма в суперклассе и заставляющий подклассы реализовать конкретные шаги этого алгоритма.

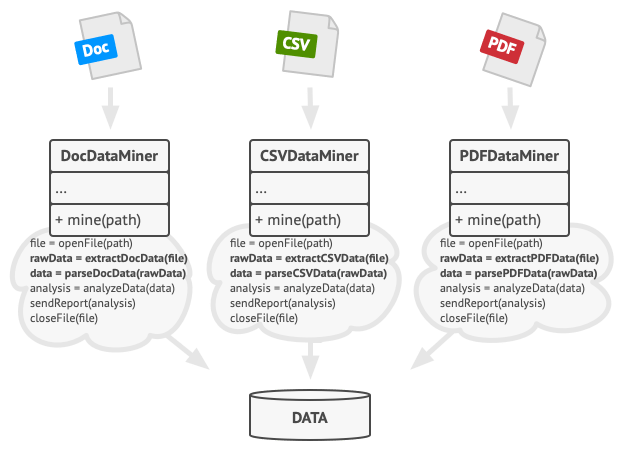
**Шаблонный метод** — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.



## Проблема

Вы пишете программу для дата-майнинга в офисных документах. Пользователи будут загружать в неё документы в разных форматах (PDF, DOC, CSV), а программа должна извлекать из них полезную информацию.

В первой версии вы ограничились только обработкой DOC-файлов. В следующей версии добавили поддержку CSV. А через месяц прикрутили работу с PDF-документами.



Классы дата-майнинга содержат много дублирования.

В какой-то момент вы заметили, что код всех трёх классов обработки документов хоть и отличается в части работы с файлами, но содержат довольно много общего в части самого извлечения данных. Было бы здорово избавится от повторной реализации алгоритма извлечения данных в каждом из классов.

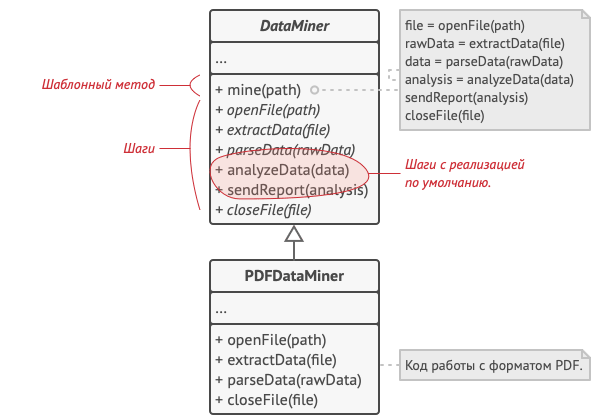
К тому же остальной код, работающий с объектами этих классов, наполнен условиями, проверяющими тип обработчика перед началом работы. Весь этот код можно упростить, если слить все три класса воедино либо свести их к общему интерфейсу.

## Решение

Паттерн Шаблонный метод предлагает разбить алгоритм на последовательность шагов, описать эти шаги в отдельных методах и вызывать их в одном шаблонном методе друг за другом.

Это позволит подклассам переопределять некоторые шаги алгоритма, оставляя без изменений его структуру и остальные шаги, которые для этого подкласса не так важны.

В нашем примере с дата-майнингом мы можем создать общий базовый класс для всех трёх алгоритмов. Этот класс будет состоять из шаблонного метода, который последовательно вызывает шаги разбора документов.



Шаблонный метод разбивает алгоритм на шаги, позволяя подклассам переопределить некоторые из них.

Для начала шаги шаблонного метода можно сделать абстрактными. Из-за этого все подклассы должны будут реализовать каждый из шагов по-своему. В нашем случае все подклассы и так содержат реализацию каждого из шагов, поэтому ничего дополнительно делать не нужно.

По-настоящему важным является следующий этап. Теперь мы можем определить общее для всех классов поведение и вынести его в суперкласс. В нашем примере шаги открытия, считывания и закрытия могут отличаться для разных типов документов, поэтому останутся абстрактными. А вот одинаковый для всех типов документов код обработки данных переедет в базовый класс.

Как видите, у нас получилось два вида шагов: абстрактные, которые каждый подкласс обязательно должен реализовать, а также шаги с реализацией по умолчанию, которые можно переопределять в подклассах, но не обязательно.

Но есть и третий тип шагов — хуки: их не обязательно переопределять, но они не содержат никакого кода, выглядя как обычные методы. Шаблонный метод останется рабочим, даже если ни один подкласс не переопределит такой хук. Однако, хук даёт подклассам дополнительные точки «вклинивания» в шаблонный метод.

# Абстрактный Класс определяет шаблонный метод, содержащий скелет некоторого

# алгоритма, состоящего из вызовов (обычно) абстрактных примитивных операций.

#

# Конкретные подклассы должны реализовать эти операции, но оставить сам

# шаблонный метод без изменений.

#

# @abstract

**class** **AbstractClass**

# Шаблонный метод определяет скелет алгоритма.

**def** **template\_method**

base\_operation1

required\_operations1

base\_operation2

hook1

required\_operations2

base\_operation3

hook2

**end**

# Эти операции уже имеют реализации.

**def** **base\_operation1**

puts 'AbstractClass says: I am doing the bulk of the work'

**end**

**def** **base\_operation2**

puts 'AbstractClass says: But I let subclasses override some operations'

**end**

**def** **base\_operation3**

puts 'AbstractClass says: But I am doing the bulk of the work anyway'

**end**

# А эти операции должны быть реализованы в подклассах.

#

# @abstract

**def** **required\_operations1**

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

# @abstract

**def** **required\_operations2**

**raise** **NotImplementedError**, "#{**self**.class} has not implemented method '#{\_\_method\_\_}'"

**end**

# Это «хуки». Подклассы могут переопределять их, но это не обязательно,

# поскольку у хуков уже есть стандартная (но пустая) реализация. Хуки

# предоставляют дополнительные точки расширения в некоторых критических местах

# алгоритма.

**def** **hook1**; **end**

**def** **hook2**; **end**

**end**

# Конкретные классы должны реализовать все абстрактные операции базового класса.

# Они также могут переопределить некоторые операции с реализацией по умолчанию.

**class** **ConcreteClass1** < **AbstractClass**

**def** **required\_operations1**

puts 'ConcreteClass1 says: Implemented Operation1'

**end**

**def** **required\_operations2**

puts 'ConcreteClass1 says: Implemented Operation2'

**end**

**end**

# Обычно конкретные классы переопределяют только часть операций базового класса.

**class** **ConcreteClass2** < **AbstractClass**

**def** **required\_operations1**

puts 'ConcreteClass2 says: Implemented Operation1'

**end**

**def** **required\_operations2**

puts 'ConcreteClass2 says: Implemented Operation2'

**end**

**def** **hook1**

puts 'ConcreteClass2 says: Overridden Hook1'

**end**

**end**

# Клиентский код вызывает шаблонный метод для выполнения алгоритма. Клиентский

# код не должен знать конкретный класс объекта, с которым работает, при условии,

# что он работает с объектами через интерфейс их базового класса.

#

# @param [AbstractClass] abstract\_class

**def** **client\_code**(abstract\_class)

# ...

abstract\_class.template\_method

# ...

**end**

puts 'Same client code can work with different subclasses:'

client\_code(**ConcreteClass1**.new)

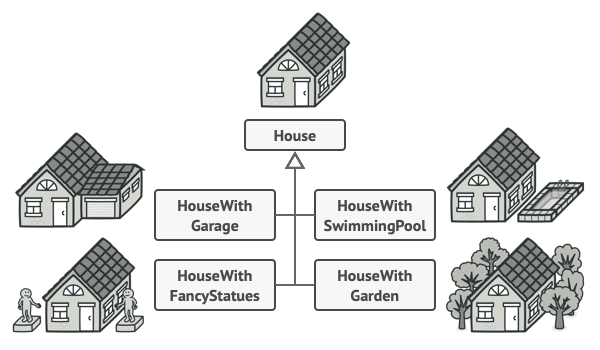
puts "\n"

puts 'Same client code can work with different subclasses:'

client\_code(**ConcreteClass2**.new)

ПО МИНИМУМУ ДРУГИЕ ПАТТЕРНЫ  
**Строитель** — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

Представьте сложный объект, требующий кропотливой пошаговой инициализации множества полей и вложенных объектов. Код инициализации таких объектов обычно спрятан внутри монструозного конструктора с десятком параметров. Либо ещё хуже — распылён по всему клиентскому коду.

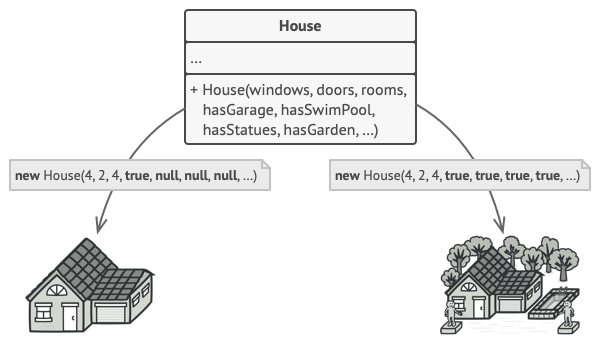


Создав кучу подклассов для всех конфигураций объектов, вы можете излишне усложнить программу.

Например, давайте подумаем о том, как создать объект Дом. Чтобы построить стандартный дом, нужно поставить 4 стены, установить двери, вставить пару окон и положить крышу. Но что, если вы хотите дом побольше да посветлее, имеющий сад, бассейн и прочее добро?

Самое простое решение — расширить класс Дом, создав подклассы для всех комбинаций параметров дома. Проблема такого подхода — это громадное количество классов, которые вам придётся создать. Каждый новый параметр, вроде цвета обоев или материала кровли, заставит вас создавать всё больше и больше классов для перечисления всех возможных вариантов.

Чтобы не плодить подклассы, вы можете подойти к решению с другой стороны. Вы можете создать гигантский конструктор Дома, принимающий уйму параметров для контроля над создаваемым продуктом. Действительно, это избавит вас от подклассов, но приведёт к другой проблеме.

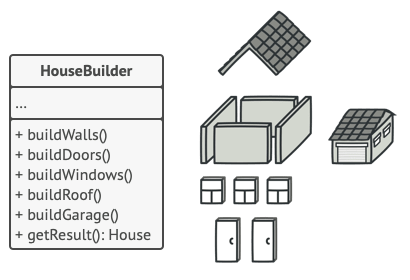


Конструктор со множеством параметров имеет свой недостаток: не все параметры нужны большую часть времени.

Большая часть этих параметров будет простаивать, а вызовы конструктора будут выглядеть монструозно из-за [**длинного списка параметров**](https://refactoringguru.cn/ru/smells/long-parameter-list). К примеру, далеко не каждый дом имеет бассейн, поэтому параметры, связанные с бассейнами, будут простаивать бесполезно в 99% случаев.

## Решение

Паттерн Строитель предлагает вынести конструирование объекта за пределы его собственного класса, поручив это дело отдельным объектам, которые следует называть строителями.

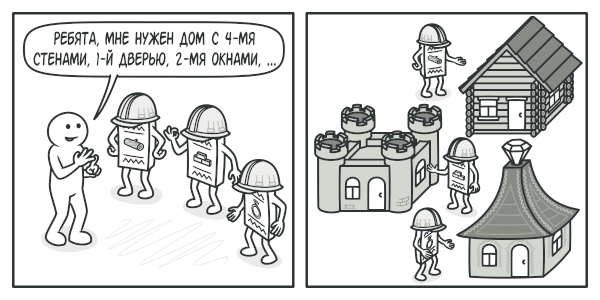


Строитель позволяет создавать сложные объекты пошагово. Промежуточный результат всегда остаётся защищён.

Паттерн предлагает разбить процесс конструирования объекта на отдельные шаги (например, построитьСтены, вставитьДвери и другие). Чтобы создать объект, вам нужно поочерёдно вызывать методы строителя. Причём не нужно запускать все шаги, а только те, что нужны для производства объекта определённой конфигурации.

Зачастую один и тот же шаг строительства может отличаться для разных вариаций производимых объектов. Например, деревянный дом потребует строительства стен из дерева, а каменный — из камня.

В этом случае вы можете создать несколько классов строителей, выполняющих одни и те же шаги по-разному. Используя этих строителей в одном и том же строительном процессе, вы сможете получать на выходе различные объекты.



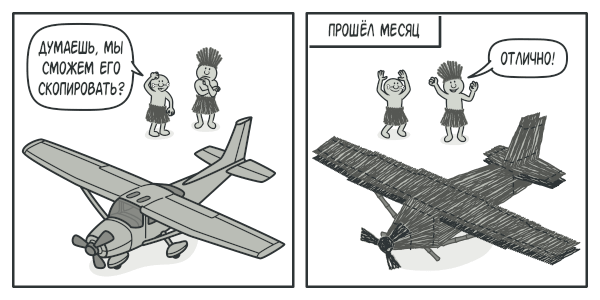
Разные строители выполнят одну и ту же задачу по-разному.

Например, один строитель делает стены из дерева и стекла, другой из камня и железа, третий из золота и бриллиантов. Вызвав одни и те же шаги строительства, в первом случае вы получите обычный жилой дом, во втором — маленькую крепость, а в третьем — роскошное жилище. Замечу, что код, который вызывает шаги строительства, должен работать со строителями через общий интерфейс, чтобы их можно было свободно взаимозаменять.

**Прототип** — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

У вас есть объект, который нужно скопировать. Как это сделать? Нужно создать пустой объект такого же класса, а затем поочерёдно скопировать значения всех полей из старого объекта в новый.

Прекрасно! Но есть нюанс. Не каждый объект удастся скопировать таким образом, ведь часть его состояния может быть приватной, а значит — недоступной для остального кода программы.



Копирование «извне» [**не всегда**](https://refactoringguru.cn/cargo-cult) возможно в реальности.

Но есть и другая проблема. Копирующий код станет зависим от классов копируемых объектов. Ведь, чтобы перебрать все поля объекта, нужно привязаться к его классу. Из-за этого вы не сможете копировать объекты, зная только их интерфейсы, а не конкретные классы.

## Решение

Паттерн Прототип поручает создание копий самим копируемым объектам. Он вводит общий интерфейс для всех объектов, поддерживающих клонирование. Это позволяет копировать объекты, не привязываясь к их конкретным классам. Обычно такой интерфейс имеет всего один метод clone.

Реализация этого метода в разных классах очень схожа. Метод создаёт новый объект текущего класса и копирует в него значения всех полей собственного объекта. Так получится скопировать даже приватные поля, так как большинство языков программирования разрешает доступ к приватным полям любого объекта текущего класса.

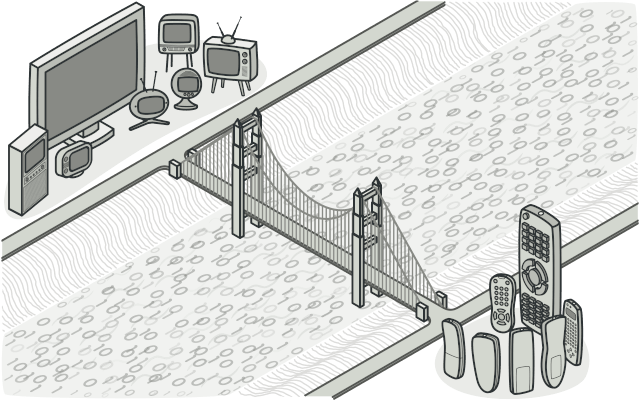
Объект, который копируют, называется прототипом (откуда и название паттерна). Когда объекты программы содержат сотни полей и тысячи возможных конфигураций, прототипы могут служить своеобразной альтернативой созданию подклассов.



Предварительно заготовленные прототипы могут стать заменой подклассам.

В этом случае все возможные прототипы заготавливаются и настраиваются на этапе инициализации программы. Потом, когда программе нужен новый объект, она создаёт копию из приготовленного прототипа.

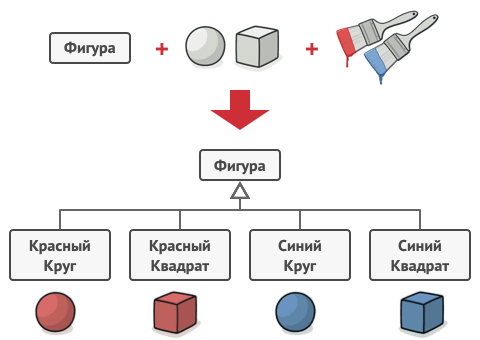
**Мост** — это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.



## Проблема

Абстракция? Реализация?! Звучит пугающе! Чтобы понять, о чём идёт речь, давайте разберём очень простой пример.

У вас есть класс геометрических Фигур, который имеет подклассы Круг и Квадрат. Вы хотите расширить иерархию фигур по цвету, то есть иметь Красные и Синие фигуры. Но чтобы всё это объединить, вам придётся создать 4 комбинации подклассов, вроде СиниеКруги и КрасныеКвадраты.



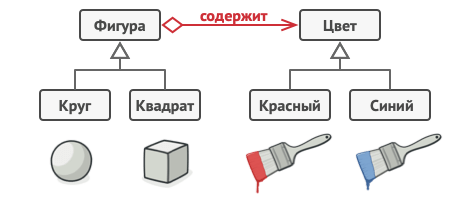
Количество подклассов растёт в геометрической прогрессии.

При добавлении новых видов фигур и цветов количество комбинаций будет расти в геометрической прогрессии. Например, чтобы ввести в программу фигуры треугольников, придётся создать сразу два новых подкласса треугольников под каждый цвет. После этого новый цвет потребует создания уже трёх классов для всех видов фигур. Чем дальше, тем хуже.

## Решение

Корень проблемы заключается в том, что мы пытаемся расширить классы фигур сразу в двух независимых плоскостях — по виду и по цвету. Именно это приводит к разрастанию дерева классов.

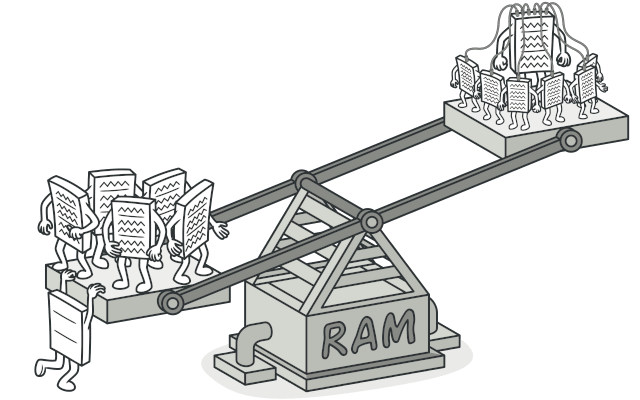
Паттерн Мост предлагает заменить наследование агрегацией или композицией. Для этого нужно выделить одну из таких «плоскостей» в отдельную иерархию и ссылаться на объект этой иерархии, вместо хранения его состояния и поведения внутри одного класса.



Размножение подклассов можно остановить, разбив классы на несколько иерархий.

Таким образом, мы можем сделать Цвет отдельным классом с подклассами Красный и Синий. Класс Фигур получит ссылку на объект Цвета и сможет делегировать ему работу, если потребуется. Такая связь и станет мостом между Фигурами и Цветом. При добавлении новых классов цветов не потребуется трогать классы фигур и наоборот.

**Легковес** — это структурный паттерн проектирования, который позволяет вместить бóльшее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.

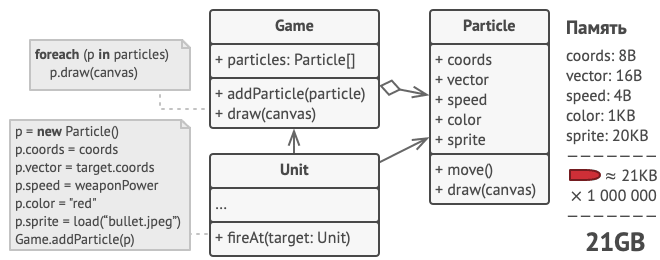


## Проблема

На досуге вы решили написать небольшую игру, в которой игроки перемещаются по карте и стреляют друг в друга. Фишкой игры должна была стать реалистичная система частиц. Пули, снаряды, осколки от взрывов — всё это должно красиво летать и радовать взгляд.

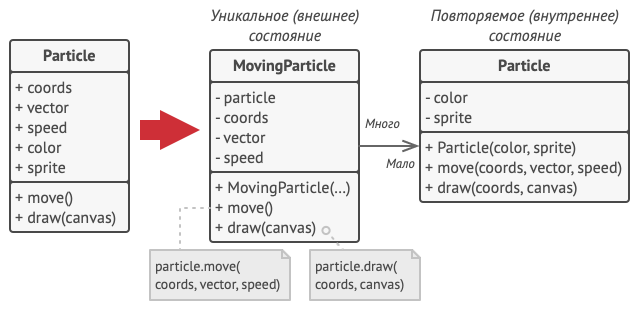
Игра отлично работала на вашем мощном компьютере. Однако ваш друг сообщил, что игра начинает тормозить и вылетает через несколько минут после запуска. Покопавшись в логах, вы обнаружили, что игра вылетает из-за недостатка оперативной памяти. У вашего друга компьютер значительно менее «прокачанный», поэтому проблема у него и проявляется так быстро.

И действительно, каждая частица представлена собственным объектом, имеющим множество данных. В определённый момент, когда побоище на экране достигает кульминации, новые объекты частиц уже не вмещаются в оперативную память компьютера, и программа вылетает.



## Решение

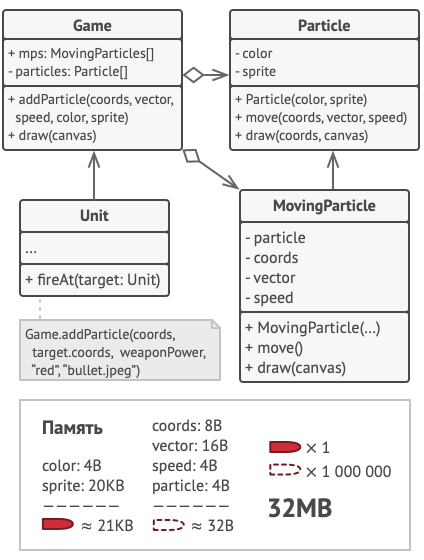
Если внимательно посмотреть на класс частиц, то можно заметить, что цвет и спрайт занимают больше всего памяти. Более того, они хранятся в каждом объекте, хотя фактически их значения одинаковы для большинства частиц.



Остальное состояние объектов — координаты, вектор движения и скорость — отличаются для всех частиц. Таким образом, эти поля можно рассматривать как контекст, в котором частица используется. А цвет и спрайт — это данные, не изменяющиеся во времени.

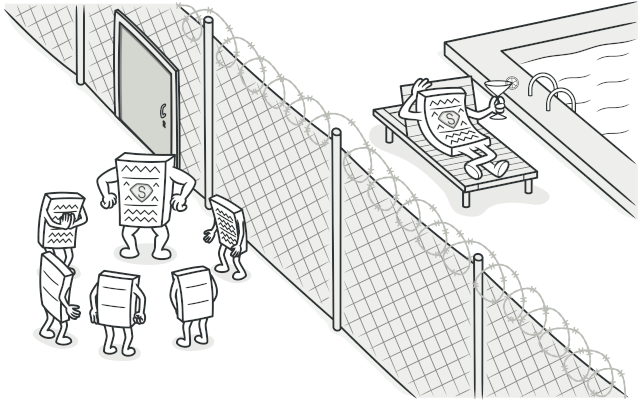
Неизменяемые данные объекта принято называть «внутренним состоянием». Все остальные данные — это «внешнее состояние».

Паттерн Легковес предлагает не хранить в классе внешнее состояние, а передавать его в те или иные методы через параметры. Таким образом, одни и те же объекты можно будет повторно использовать в различных контекстах. Но главное — понадобится гораздо меньше объектов, ведь теперь они будут отличаться только внутренним состоянием, а оно имеет не так много вариаций.



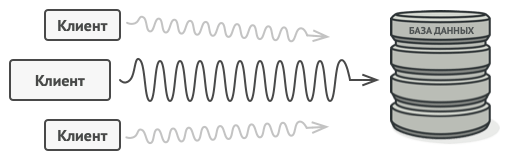
В нашем примере с частицами достаточно будет оставить всего три объекта с отличающимися спрайтами и цветом — для пуль, снарядов и осколков. Несложно догадаться, что такие облегчённые объекты называют легковéсами

**Заместитель** — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу.



## Проблема

Для чего вообще контролировать доступ к объектам? Рассмотрим такой пример: у вас есть внешний ресурсоёмкий объект, который нужен не все время, а изредка.



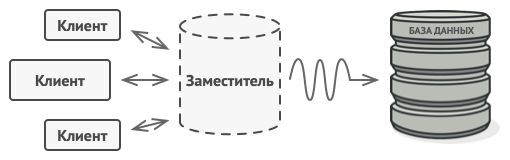
Запросы к базе данных могут быть очень медленными.

Мы могли бы создавать этот объект не в самом начале программы, а только тогда, когда он кому-то реально понадобится. Каждый клиент объекта получил бы некий код отложенной инициализации. Но, вероятно, это привело бы к множественному дублированию кода.

В идеале, этот код хотелось бы поместить прямо в служебный класс, но это не всегда возможно. Например, код класса может находиться в закрытой сторонней библиотеке.

## Решение

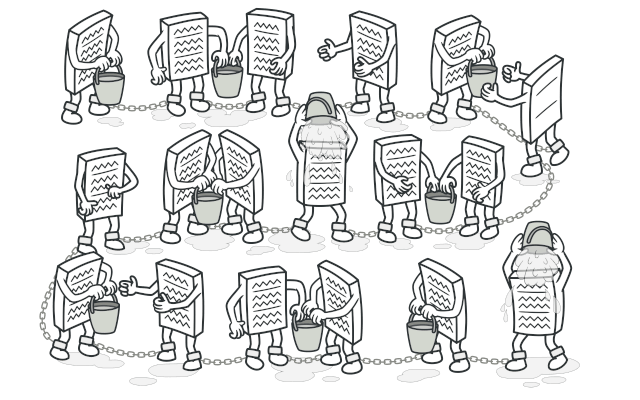
Паттерн Заместитель предлагает создать новый класс-дублёр, имеющий тот же интерфейс, что и оригинальный служебный объект. При получении запроса от клиента объект-заместитель сам бы создавал экземпляр служебного объекта и переадресовывал бы ему всю реальную работу.



Заместитель «притворяется» базой данных, ускоряя работу за счёт ленивой инициализации и кеширования повторяющихся запросов.

Но в чём же здесь польза? Вы могли бы поместить в класс заместителя какую-то промежуточную логику, которая выполнялась бы до (или после) вызовов этих же методов в настоящем объекте. А благодаря одинаковому интерфейсу, объект-заместитель можно передать в любой код, ожидающий сервисный объект.

**Цепочка обязанностей** — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.



## Проблема

Представьте, что вы делаете систему приёма онлайн-заказов. Вы хотите ограничить к ней доступ так, чтобы только авторизованные пользователи могли создавать заказы. Кроме того, определённые пользователи, владеющие правами администратора, должны иметь полный доступ к заказам.

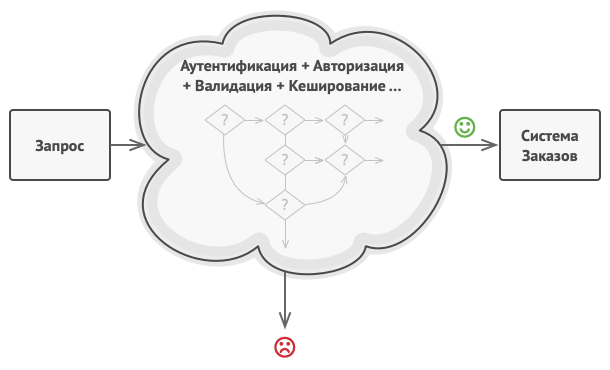
Вы быстро сообразили, что эти проверки нужно выполнять последовательно. Ведь пользователя можно попытаться «залогинить» в систему, если его запрос содержит логин и пароль. Но если такая попытка не удалась, то проверять расширенные права доступа попросту не имеет смысла.



Запрос проходит ряд проверок перед доступом в систему заказов.

На протяжении следующих нескольких месяцев вам пришлось добавить ещё несколько таких последовательных проверок.

* Кто-то резонно заметил, что неплохо бы проверять данные, передаваемые в запросе перед тем, как вносить их в систему — вдруг запрос содержит данные о покупке несуществующих продуктов.
* Кто-то предложил блокировать массовые отправки формы с одним и тем же логином, чтобы предотвратить подбор паролей ботами.
* Кто-то заметил, что форму заказа неплохо бы доставать из кеша, если она уже была однажды показана.



Со временем код проверок становится всё более запутанным.

С каждой новой «фичей» код проверок, выглядящий как большой клубок условных операторов, всё больше и больше раздувался. При изменении одного правила приходилось трогать код всех проверок. А для того, чтобы применить проверки к другим ресурсам, пришлось продублировать их код в других классах.

Поддерживать такой код стало не только очень хлопотно, но и затратно. И вот в один прекрасный день вы получаете задачу рефакторинга...

## Решение

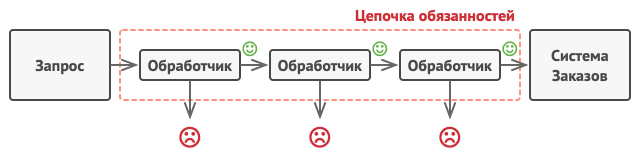
Как и многие другие поведенческие паттерны, Цепочка обязанностей базируется на том, чтобы превратить отдельные поведения в объекты. В нашем случае каждая проверка переедет в отдельный класс с единственным методом выполнения. Данные запроса, над которым происходит проверка, будут передаваться в метод как аргументы.

А теперь по-настоящему важный этап. Паттерн предлагает связать объекты обработчиков в одну цепь. Каждый из них будет иметь ссылку на следующий обработчик в цепи. Таким образом, при получении запроса обработчик сможет не только сам что-то с ним сделать, но и передать обработку следующему объекту в цепочке.

Передавая запросы в первый обработчик цепочки, вы можете быть уверены, что все объекты в цепи смогут его обработать. При этом длина цепочки не имеет никакого значения.

И последний штрих. Обработчик не обязательно должен передавать запрос дальше, причём эта особенность может быть использована по-разному.

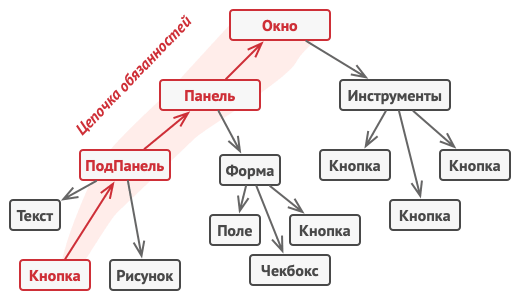
В примере с фильтрацией доступа обработчики прерывают дальнейшие проверки, если текущая проверка не прошла. Ведь нет смысла тратить попусту ресурсы, если и так понятно, что с запросом что-то не так.



Обработчики следуют в цепочке один за другим.

Но есть и другой подход, при котором обработчики прерывают цепь только когда они могут обработать запрос. В этом случае запрос движется по цепи, пока не найдётся обработчик, который может его обработать. Очень часто такой подход используется для передачи событий, создаваемых классами графического интерфейса в результате взаимодействия с пользователем.

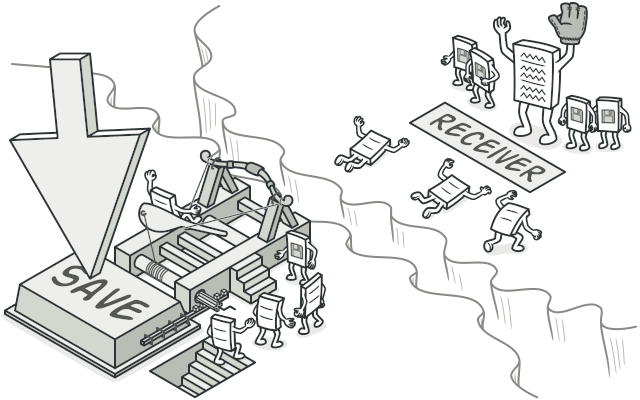
Например, когда пользователь кликает по кнопке, программа выстраивает цепочку из объекта этой кнопки, всех её родительских элементов и общего окна приложения на конце. Событие клика передаётся по этой цепи до тех пор, пока не найдётся объект, способный его обработать. Этот пример примечателен ещё и тем, что цепочку всегда можно выделить из древовидной структуры объектов, в которую обычно и свёрнуты элементы пользовательского интерфейса.



Цепочку можно выделить даже из дерева объектов.

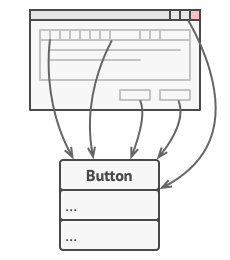
Очень важно, чтобы все объекты цепочки имели общий интерфейс. Обычно каждому конкретному обработчику достаточно знать только то, что следующий объект в цепи имеет метод выполнить. Благодаря этому связи между объектами цепочки будут более гибкими. Кроме того, вы сможете формировать цепочки на лету из разнообразных объектов, не привязываясь к конкретным классам.

**Команда** — это поведенческий паттерн проектирования, который превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций.



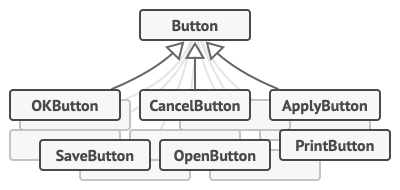
## Проблема

Представьте, что вы работаете над программой текстового редактора. Дело как раз подошло к разработке панели управления. Вы создали класс красивых Кнопок и хотите использовать его для всех кнопок приложения, начиная от панели управления, заканчивая простыми кнопками в диалогах.



Все кнопки приложения унаследованы от одного класса.

Все эти кнопки, хоть и выглядят схоже, но делают разные вещи. Поэтому возникает вопрос: куда поместить код обработчиков кликов по этим кнопкам? Самым простым решением было бы создать подклассы для каждой кнопки и переопределить в них метод действия под разные задачи.



Множество подклассов кнопок.

Но скоро стало понятно, что такой подход никуда не годится. Во-первых, получается очень много подклассов. Во-вторых, код кнопок, относящийся к графическому интерфейсу, начинает зависеть от классов бизнес-логики, которая довольно часто меняется.



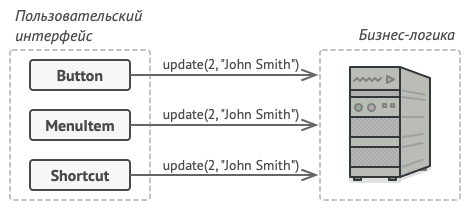
Несколько классов дублируют одну и ту же функциональность.

Но самое обидное ещё впереди. Ведь некоторые операции, например, «сохранить», можно вызывать из нескольких мест: нажав кнопку на панели управления, вызвав контекстное меню или просто нажав клавиши Ctrl+S. Когда в программе были только кнопки, код сохранения имелся только в подклассе SaveButton. Но теперь его придётся продублировать ещё в два класса.

## Решение

Хорошие программы обычно структурированы в виде слоёв. Самый распространённый пример — слои пользовательского интерфейса и бизнес-логики. Первый всего лишь рисует красивую картинку для пользователя. Но когда нужно сделать что-то важное, интерфейс «просит» слой бизнес-логики заняться этим.

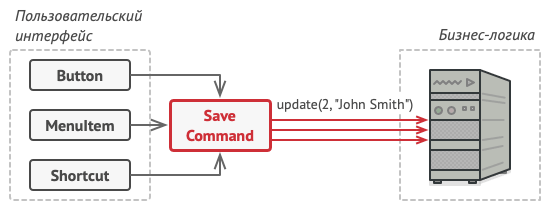
В реальности это выглядит так: один из объектов интерфейса напрямую вызывает метод одного из объектов бизнес-логики, передавая в него какие-то параметры.



Прямой доступ из UI в бизнес-логику.

Паттерн Команда предлагает больше не отправлять такие вызовы напрямую. Вместо этого каждый вызов, отличающийся от других, следует завернуть в собственный класс с единственным методом, который и будет осуществлять вызов. Такие объекты называют командами.

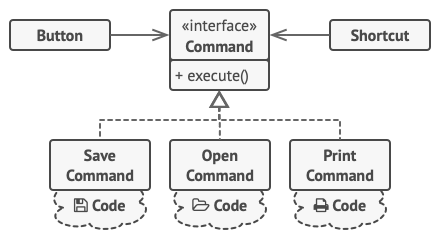
К объекту интерфейса можно будет привязать объект команды, который знает, кому и в каком виде следует отправлять запросы. Когда объект интерфейса будет готов передать запрос, он вызовет метод команды, а та — позаботится обо всём остальном.



Доступ из UI в бизнес-логику через команду.

Классы команд можно объединить под общим интерфейсом c единственным методом запуска. После этого одни и те же отправители смогут работать с различными командами, не привязываясь к их классам. Даже больше: команды можно будет взаимозаменять на лету, изменяя итоговое поведение отправителей.

Параметры, с которыми должен быть вызван метод объекта получателя, можно загодя сохранить в полях объекта-команды. Благодаря этому, объекты, отправляющие запросы, могут не беспокоиться о том, чтобы собрать необходимые для получателя данные. Более того, они теперь вообще не знают, кто будет получателем запроса. Вся эта информация скрыта внутри команды.



Классы UI делегируют работу командам.

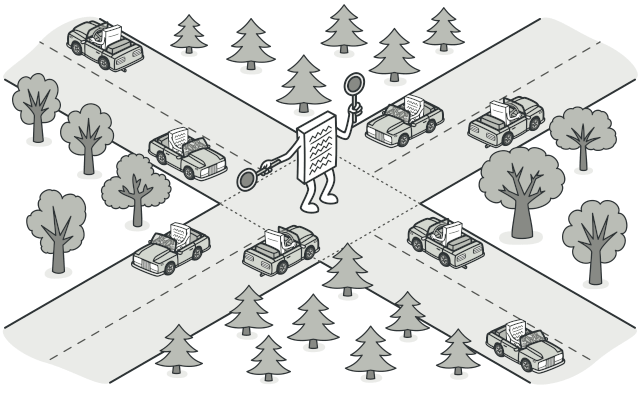
После применения Команды в нашем примере с текстовым редактором вам больше не потребуется создавать уйму подклассов кнопок под разные действия. Будет достаточно единственного класса с полем для хранения объекта команды.

Используя общий интерфейс команд, объекты кнопок будут ссылаться на объекты команд различных типов. При нажатии кнопки будут делегировать работу связанным командам, а команды — перенаправлять вызовы тем или иным объектам бизнес-логики.

Так же можно поступить и с контекстным меню, и с горячими клавишами. Они будут привязаны к тем же объектам команд, что и кнопки, избавляя классы от дублирования.

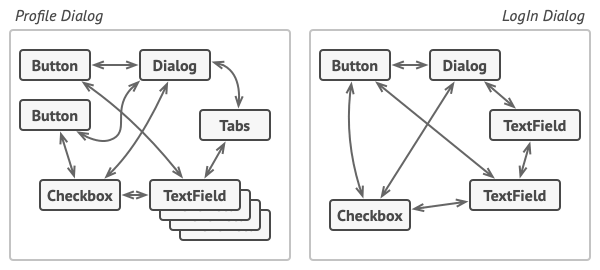
Таким образом, команды станут гибкой прослойкой между пользовательским интерфейсом и бизнес-логикой. И это лишь малая доля пользы, которую может принести паттерн Команда!

**Посредник** — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.



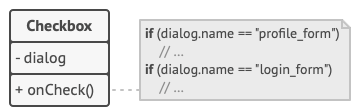
## Проблема

Предположим, что у вас есть диалог создания профиля пользователя. Он состоит из всевозможных элементов управления — текстовых полей, чекбоксов, кнопок.



Беспорядочные связи между элементами пользовательского интерфейса.

Отдельные элементы диалога должны взаимодействовать друг с другом. Так, например, чекбокс «у меня есть собака» открывает скрытое поле для ввода имени домашнего любимца, а клик по кнопке отправки запускает проверку значений всех полей формы.



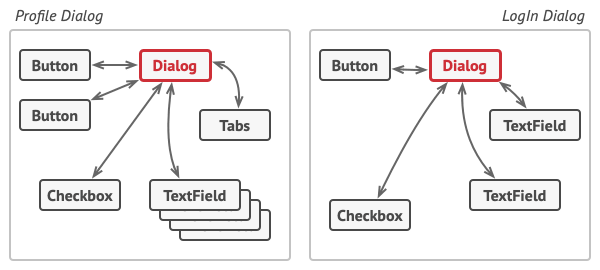
Код элементов нужно трогать при изменении каждого диалога.

Прописав эту логику прямо в коде элементов управления, вы поставите крест на их повторном использовании в других местах приложения. Они станут слишком тесно связанными с элементами диалога редактирования профиля, которые не нужны в других контекстах. Поэтому вы сможете использовать либо все элементы сразу, либо ни одного.

## Решение

Паттерн Посредник заставляет объекты общаться не напрямую друг с другом, а через отдельный объект-посредник, который знает, кому нужно перенаправить тот или иной запрос. Благодаря этому, компоненты системы будут зависеть только от посредника, а не от десятков других компонентов.

В нашем примере посредником мог бы стать диалог. Скорее всего, класс диалога и так знает, из каких элементов состоит, поэтому никаких новых связей добавлять в него не придётся.



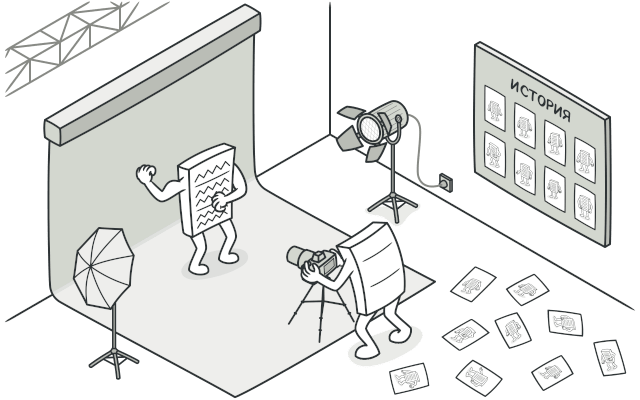
Элементы интерфейса общаются через посредника.

Основные изменения произойдут внутри отдельных элементов диалога. Если раньше при получении клика от пользователя объект кнопки сам проверял значения полей диалога, то теперь его единственной обязанностью будет сообщить диалогу о том, что произошёл клик. Получив извещение, диалог выполнит все необходимые проверки полей. Таким образом, вместо нескольких зависимостей от остальных элементов кнопка получит только одну — от самого диалога.

Чтобы сделать код ещё более гибким, можно выделить общий интерфейс для всех посредников, то есть диалогов программы. Наша кнопка станет зависимой не от конкретного диалога создания пользователя, а от абстрактного, что позволит использовать её и в других диалогах.

Таким образом, посредник скрывает в себе все сложные связи и зависимости между классами отдельных компонентов программы. А чем меньше связей имеют классы, тем проще их изменять, расширять и повторно использовать.

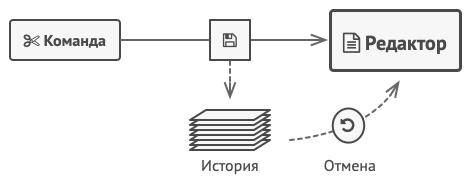
**Снимок** — это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.



## Проблема

Предположим, что вы пишете программу текстового редактора. Помимо обычного редактирования, ваш редактор позволяет менять форматирование текста, вставлять картинки и прочее.

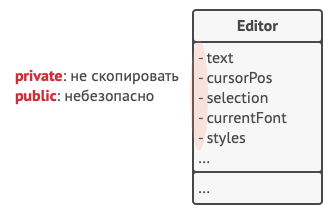
В какой-то момент вы решили сделать все эти действия отменяемыми. Для этого вам нужно сохранять текущее состояние редактора перед тем, как выполнить любое действие. Если потом пользователь решит отменить своё действие, вы достанете копию состояния из истории и восстановите старое состояние редактора.



Перед выполнением команды вы можете сохранить копию состояния редактора, чтобы потом иметь возможность отменить операцию.

Чтобы сделать копию состояния объекта, достаточно скопировать значение его полей. Таким образом, если вы сделали класс редактора достаточно открытым, то любой другой класс сможет заглянуть внутрь, чтобы скопировать его состояние.

Казалось бы, что ещё нужно? Ведь теперь любая операция сможет сделать резервную копию редактора перед своим действием. Но такой наивный подход обеспечит вам уйму проблем в будущем. Ведь если вы решите провести рефакторинг — убрать или добавить парочку полей в класс редактора — то придётся менять код всех классов, которые могли копировать состояние редактора.



Как команде создать снимок состояния редактора, если все его поля приватные?

Но это ещё не все. Давайте теперь рассмотрим сами копии состояния редактора. Из чего состоит состояние редактора? Даже самый примитивный редактор должен иметь несколько полей для хранения текущего текста, позиции курсора и прокрутки экрана. Чтобы сделать копию состояния, вам нужно записать значения всех этих полей в некий «контейнер».

Скорее всего, вам понадобится хранить массу таких контейнеров в качестве истории операций, поэтому удобнее всего сделать их объектами одного класса. Этот класс должен иметь много полей, но практически никаких методов. Чтобы другие объекты могли записывать и читать из него данные, вам придётся сделать его поля публичными. Но это приведёт к той же проблеме, что и с открытым классом редактора. Другие классы станут зависимыми от любых изменений в классе контейнера, который подвержен тем же изменениям, что и класс редактора.

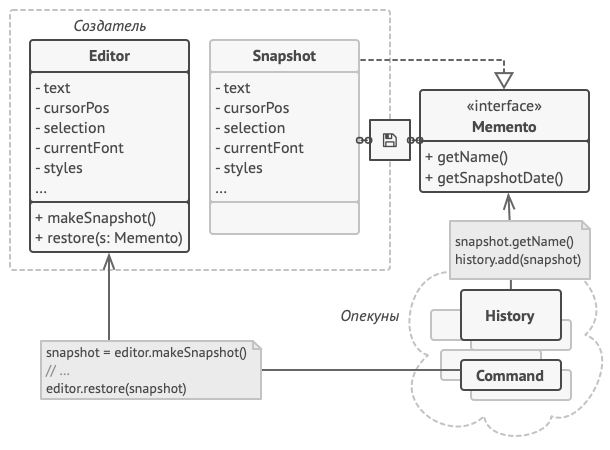
Получается, нам придётся либо открыть классы для всех желающих, испытывая массу хлопот с поддержкой кода, либо оставить классы закрытыми, отказавшись от идеи отмены операций. Нет ли какого-то другого пути?

## Решение

Все проблемы, описанные выше, возникают из-за нарушения инкапсуляции. Это когда одни объекты пытаются сделать работу за других, влезая в их приватную зону, чтобы собрать необходимые для операции данные.

Паттерн Снимок поручает создание копии состояния объекта самому объекту, который этим состоянием владеет. Вместо того, чтобы делать снимок «извне», наш редактор сам сделает копию своих полей, ведь ему доступны все поля, даже приватные.

Паттерн предлагает держать копию состояния в специальном объекте-снимке с ограниченным интерфейсом, позволяющим, например, узнать дату изготовления или название снимка. Но, с другой стороны, снимок должен быть открыт для своего создателя, позволяя прочесть и восстановить его внутреннее состояние.

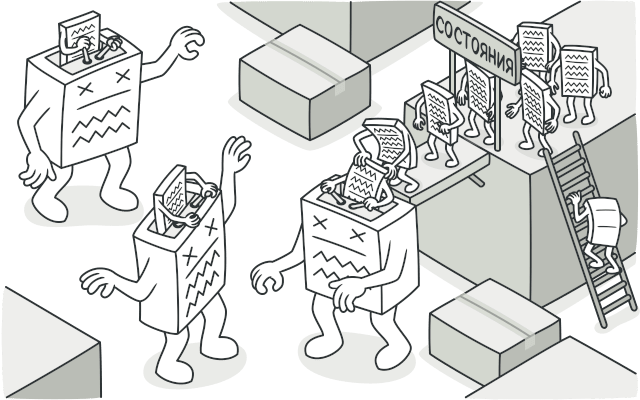


Снимок полностью открыт для создателя, но лишь частично открыт для опекунов.

Такая схема позволяет создателям производить снимки и отдавать их для хранения другим объектам, называемым опекунами. Опекунам будет доступен только ограниченный интерфейс снимка, поэтому они никак не смогут повлиять на «внутренности» самого снимка. В нужный момент опекун может попросить создателя восстановить своё состояние, передав ему соответствующий снимок.

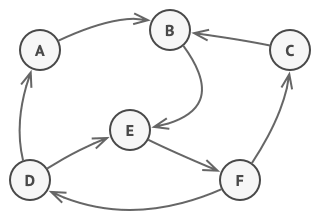
В примере с редактором вы можете сделать опекуном отдельный класс, который будет хранить список выполненных операций. Ограниченный интерфейс снимков позволит демонстрировать пользователю красивый список с названиями и датами выполненных операций. А когда пользователь решит откатить операцию, класс истории возьмёт последний снимок из стека и отправит его объекту редактор для восстановления.

**Состояние** — это поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.



## Проблема

Паттерн Состояние невозможно рассматривать в отрыве от концепции машины состояний, также известной как стейт-машина или конечный автомат .

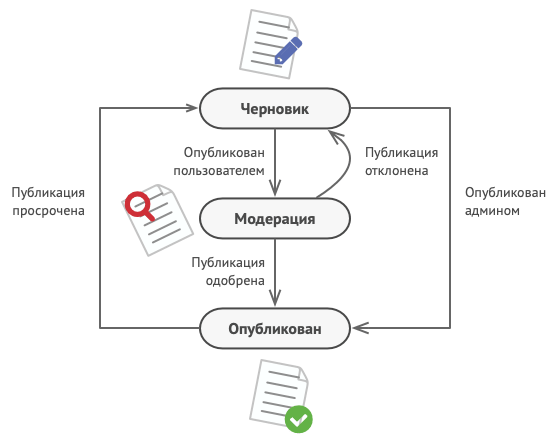


Конечный автомат.

Основная идея в том, что программа может находиться в одном из нескольких состояний, которые всё время сменяют друг друга. Набор этих состояний, а также переходов между ними, предопределён и конечен. Находясь в разных состояниях, программа может по-разному реагировать на одни и те же события, которые происходят с ней.

Такой подход можно применить и к отдельным объектам. Например, объект Документ может принимать три состояния: Черновик, Модерация или Опубликован. В каждом из этих состоянии метод опубликовать будет работать по-разному:

* Из черновика он отправит документ на модерацию.
* Из модерации — в публикацию, но при условии, что это сделал администратор.
* В опубликованном состоянии метод не будет делать ничего.



Возможные состояния документа и переходы между ними.

Машину состояний чаще всего реализуют с помощью множества условных операторов, if либо switch, которые проверяют текущее состояние объекта и выполняют соответствующее поведение. Наверняка вы уже реализовали хотя бы одну машину состояний в своей жизни, даже не зная об этом. Как насчёт вот такого кода, выглядит знакомо?

**class** **Document** **is**

**field** state: string

// ...

**method** publish() **is**

switch (state)

"draft":

state = "moderation"

break

"moderation":

**if** (currentUser.role == "admin")

state = "published"

break

"published":

// Do nothing.

break

// ...

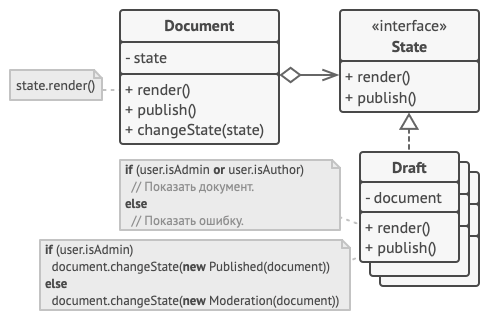
Основная проблема такой машины состояний проявится в том случае, если в Документ добавить ещё десяток состояний. Каждый метод будет состоять из увесистого условного оператора, перебирающего доступные состояния. Такой код крайне сложно поддерживать. Малейшее изменение логики переходов заставит вас перепроверять работу всех методов, которые содержат условные операторы машины состояний.

Путаница и нагромождение условий особенно сильно проявляется в старых проектах. Набор возможных состояний бывает трудно предопределить заранее, поэтому они всё время добавляются в процессе эволюции программы. Из-за этого решение, которое выглядело простым и эффективным в самом начале разработки, может впоследствии стать проекцией большого макаронного монстра.

## Решение

Паттерн Состояние предлагает создать отдельные классы для каждого состояния, в котором может пребывать объект, а затем вынести туда поведения, соответствующие этим состояниям.

Вместо того, чтобы хранить код всех состояний, первоначальный объект, называемый контекстом, будет содержать ссылку на один из объектов-состояний и делегировать ему работу, зависящую от состояния.

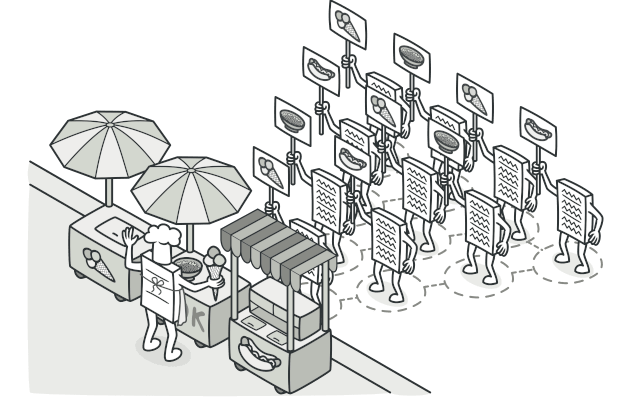


Документ делегирует работу своему активному объекту-состоянию.

Благодаря тому, что объекты состояний будут иметь общий интерфейс, контекст сможет делегировать работу состоянию, не привязываясь к его классу. Поведение контекста можно будет изменить в любой момент, подключив к нему другой объект-состояние.

Очень важным нюансом, отличающим этот паттерн от [**Стратегии**](https://refactoringguru.cn/ru/design-patterns/strategy), является то, что и контекст, и сами конкретные состояния могут знать друг о друге и инициировать переходы от одного состояния к другому.

**Посетитель** — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.



## Проблема

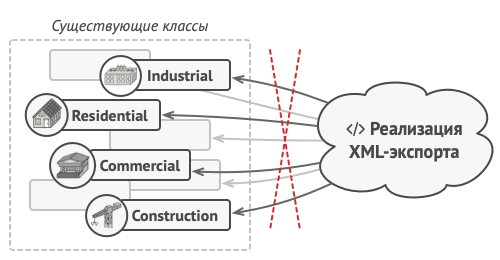
Ваша команда разрабатывает приложение, работающее с геоданными в виде графа. Узлами графа являются городские локации: памятники, театры, рестораны, важные предприятия и прочее. Каждый узел имеет ссылки на другие, ближайшие к нему узлы. Каждому типу узлов соответствует свой класс, а каждый узел представлен отдельным объектом.



Экспорт геоузлов в XML.

Ваша задача — сделать экспорт этого графа в XML. Дело было бы плёвым, если бы вы могли редактировать классы узлов. Достаточно было бы добавить метод экспорта в каждый тип узла, а затем, перебирая узлы графа, вызывать этот метод для каждого узла. Благодаря полиморфизму, решение получилось бы изящным, так как вам не пришлось бы привязываться к конкретным классам узлов.

Но, к сожалению, классы узлов вам изменить не удалось. Системный архитектор сослался на то, что код классов узлов сейчас очень стабилен, и от него многое зависит, поэтому он не хочет рисковать и позволять кому-либо его трогать.



Код XML-экспорта придётся добавить во все классы узлов, а это слишком накладно.

К тому же он сомневался в том, что экспорт в XML вообще уместен в рамках этих классов. Их основная задача была связана с геоданными, а экспорт выглядит в рамках этих классов чужеродно.

Была и ещё одна причина запрета. Если на следующей неделе вам бы понадобился экспорт в какой-то другой формат данных, то эти классы снова пришлось бы менять.

## Решение

Паттерн Посетитель предлагает разместить новое поведение в отдельном классе, вместо того чтобы множить его сразу в нескольких классах. Объекты, с которыми должно было быть связано поведение, не будут выполнять его самостоятельно. Вместо этого вы будете передавать эти объекты в методы посетителя.

Код поведения, скорее всего, должен отличаться для объектов разных классов, поэтому и методов у посетителя должно быть несколько. Названия и принцип действия этих методов будут схожи, но основное отличие будет в типе принимаемого в параметрах объекта, например:

**class** **ExportVisitor** **implements** Visitor **is**

**method** doForCity(City c) { ... }

**method** doForIndustry(Industry f) { ... }

**method** doForSightSeeing(SightSeeing ss) { ... }

// ...

Здесь возникает вопрос: как подавать узлы в объект-посетитель? Так как все методы имеют отличающуюся сигнатуру, использовать полиморфизм при переборе узлов не получится. Придётся проверять тип узлов для того, чтобы выбрать соответствующий метод посетителя.

**foreach** (Node node in graph)

**if** (node instanceof City)

exportVisitor.doForCity((City) node)

**if** (node instanceof Industry)

exportVisitor.doForIndustry((Industry) node)

// ...

Тут не поможет даже механизм перегрузки методов (доступный в Java и C#). Если назвать все методы одинаково, то неопределённость реального типа узла всё равно не даст вызвать правильный метод. Механизм перегрузки всё время будет вызывать метод посетителя, соответствующий типу Node, а не реального класса поданного узла.

Но паттерн Посетитель решает и эту проблему, используя механизм [**двойной диспетчеризации**](https://refactoringguru.cn/ru/design-patterns/visitor-double-dispatch). Вместо того, чтобы самим искать нужный метод, мы можем поручить это объектам, которые передаём в параметрах посетителю. А они уже вызовут правильный метод посетителя.

// Client code

**foreach** (Node node in graph)

node.accept(exportVisitor)

// City

**class** **City** **is**

**method** accept(Visitor v) **is**

v.doForCity(**this**)

// ...

// Industry

**class** **Industry** **is**

**method** accept(Visitor v) **is**

v.doForIndustry(**this**)

// ...

Как видите, изменить классы узлов всё-таки придётся. Но это простое изменение позволит применять к объектам узлов и другие поведения, ведь классы узлов будут привязаны не к конкретному классу посетителей, а к их общему интерфейсу. Поэтому если придётся добавить в программу новое поведение, вы создадите новый класс посетителей и будете передавать его в методы узлов.