**Вопрос 1. Распишите основные принципы ООП. Приведите пример работающего кода для иллюстрации выполнения или невыполнения этих принципов.**

**Основные принципы ООП** – инкапсуляция, наследование и полиморфизм (иногда также выделяется абстракция).

**Инкапсуляция** предполагает объединение данных и методов, работающих с этими данными, в одном объекте. В большинстве языков программирования она также подразумевает сокрытие внутренней реализации объекта от внешнего мира.

Пример выполнения инкапсуляции:

class BankAccount

def initialize(owner, balance)

@owner = owner

@balance = balance

end

def deposit(amount)

@balance += amount

end

def withdraw(amount)

@balance -= amount if amount <= @balance

end

def balance

@balance

end

end

В данном примере манипуляции с балансом на банковском счету могут производиться только с помощью выделенных в классе методов.

Переменная, хранящая значение баланса, напрямую недоступна. Её открытие через, например, *attr\_accessor*, нарушило бы инкапсуляцию, что привело бы к тому, что баланс мог бы изменять кто угодно вне класса, что недопустимо в рамках предметной области.

**Наследование** позволяет одному классу (дочернему) копировать свойства и методы другого класса (родительского), что способствует значительному сокращению повторения кода. Наследуемые классы могут как использовать классы родителя, так и переопределять их и добавлять свои. Наследование чётко определяет иерархию классов.

Пример выполнения наследования:

class Vehicle

attr\_accessor :speed, :capacity

def initialize(speed, capacity)

@speed = speed

@capacity = capacity

end

def description

"This vehicle moves at #{@speed} km/h and can carry #{@capacity} people."

end

def drive

...

end

end

class Car < Vehicle

def honk

"Beep! Beep!"

end

end

class Bicycle < Vehicle

def ring\_bell

"Ring! Ring!"

end

end

Классы *Car* и *Bicycle* наследуются от класса *Vehicle*. Они имеют общие атрибуты *speed* и *capacity* и общие методы *description* и *drive*. Класс *Car* добавляет метод *honk*, а класс *Bicycle* – *ring\_bell*. Без применения наследования эти классы имели бы много повторяющегося кода.

**Полиморфизм** предполагает возможность различным классам иметь общий интерфейс с различной, независимой друг от друга реализацией. Другими словами, полиморфизм позволяет решать одну и ту же задачу разными способами.

Пример выполнения полиморфизма:

class Animal

def speak

raise NotImplementedError, "No animal to speak"

end

end

class Dog < Animal

def speak

"Woof!"

end

end

class Cat < Animal

def speak

"Meow!"

end

end

def make\_animal\_speak(animal)

puts animal.speak

end

Согласно принципу полиморфизма, классы *Dog* и *Cat* имеют общий интерфейс, включающий в себя метод *speak*. Это позволяет внешней функции *make\_animal­\_speak* работать корректно независимо от того, был ли передан в качестве аргумента объект класса *Dog* или класса *Cat*. Нарушение полиморфизма предполагало бы использование жёсткого, немасштабируемого кода, подстраивающегося под каждый отдельный класс, т.е. в функции *make\_animal\_speak* было бы необходимо делать проверку на переданный класс, и в зависимости от фактических данных менять вывод.

**Абстракция** предполагает использование только определения характеристик объекта, без описания их конкретных/детальных реализаций. Основная идея состоит в том, чтобы предоставить объект с набором методов и не предоставлять при этом конкретную логику этих методов.

Пример выполнения абстракции:

class Shape

def area

raise NotImplementedError, "This method should be overridden by subclasses"

end

end

class Circle < Shape

def initialize(radius)

@radius = radius

end

def area

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

end

end

class Rectangle < Shape

def initialize(width, height)

@width = width

@height = height

End

def area

@width \* @height

end

end

shapes = [Circle.new(5), Rectangle.new(4, 6)]

shapes.each do |shape|

puts shape.area

end

Абстракция позволяет извне получить результаты метода *area* для каждого из классов фигур, при этом нет необходимости в знании точной реализации вычисления площади для каждой из них.

**Вопрос 2. 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.**

Существует 6 основных типов отношений между классами:

1) **Ассоциация**

 или

 (на концах указывается число экземпляров каждого класса, участвующих в связи)

Базовая связь, подразумевающая взаимодействие двух классов.

Подтипами ассоциации являются агрегация и композиция, но они подразумевают отношение «часть-целое, поэтому если ни к агрегации, ни к композиции связь не относится, то это означает, что обе сущности независимы друг от друга, и связь является простой ассоциацией.

В предметной области такие связи образуются между любыми сущностями, так или иначе взаимодействующих друг с другом, но при этом никак не зависящих друг от друга.

Например, студент может записаться на один и более курсов, но ни студент не зависит от курса, ни курс не зависит от студента.

В коде ассоциация предполагает, что свойство класса содержит ссылку на экземпляр (или экземпляры) другого класса. В контексте примера выше это выглядит так:

class Student

attr\_accessor :name, :courses

def initialize(name)

@name = name

@courses = []

end

def add\_course(course)

@courses << course

end

end

class Course

attr\_accessor :title

def initialize(title)

@title = title

end

end

2) **Наследование**



Также называется **обобщением**, используется для описания отношений между родительским и дочерним классами.

В предметной области описывает связь «является» («is a»), например «собака является животным».

В коде реализуется одноимённым принципом ООП.

class Vehicle

attr\_accessor :speed, :capacity

def initialize(speed, capacity)

@speed = speed

@capacity = capacity

end

def drive

...

end

end

class Car < Vehicle

def honk

"Beep! Beep!"

end

end

3) **Реализация**



В основном используется для указания связи между интерфейсами и реализующими их классами.

В предметной области предполагает, что родительская сущность обретает необходимый в рамках решаемой задачи смысл только в контексте конкретной дочерней сущности. Например, если водителю важен только способ вождения транспортным средством без каких-либо других его свойств, то сущности *машина* и *велосипед* будут реализациями сущности *транспортное средство*, так как неизвестно, как управлять абстрактным транспортным средством.

В коде для реализации данного отношения создаётся абстрактный класс (интерфейс), определяющий набор функций, но не содержащий никакой логики, кроме, разве что выбрасывания исключения. Классы, реализующие данный интерфейс, в зависимости от конкретных задач и языков программирования, либо наследуют его, либо «подключают» его (*include* в Ruby) и добавляют необходимую логику к абстрактным методам.

class Vehicle

def drive

raise NotImplementedError

end

end

class Car < Vehicle

def drive

"Driving the car"

end

end

class Bicycle < Vehicle

def drive

"Driving the bicycle"

end

end

4) **Зависимость**



Данное отношение предполагает временную связь, в которой изменения в одном объекте вызовут изменения в связанном с ним объекте. Также говорят, что один объект «использует» другой в своих методах, не храня при этом ссылку на экземпляр внутри своих атрибутов.

В предметной области данное отношение обозначает «использование» одной сущности другой сущностью. Зависимость не требует тесной связи между объектами, но подразумевает, что зависимая сущность может по-разному себя вести в зависимости от состояния используемой сущности.

В коде зависимость выражается в использовании внутри методов одного класса свойств другого класса. При этом напрямую экземпляры другого класса не хранятся внутри главного.

class Document

attr\_accessor :content

def initialize(content)

@content = content

end

end

class Printer

def print(document)

if document.nil? || document.content.nil?

raise ArgumentError, "No document or empty document"

else

puts "Printing document: #{document.content}"

end

end

end

5) **Агрегация**



Слабое отношение «часть-целое». В нём один объект входит в состав другого, но оба могут существовать независимо друг от друга.

В коде агрегирующий класс должен иметь в своём атрибуте ссылку на один или несколько (массив) экземпляров агрегируемого класса. При этом и тот, и другой могут иметь экземпляры, существующие вне описываемого контекста.

class Box

attr\_accessor :items

def initialize()

@items = []

end

def add\_item(item)

@items << item

end

end

class Item

...

end

Коробка содержит в себе предметы, но может существовать без них, как и предметы могут существовать без коробки.

6) **Композиция**



Сильное отношение «часть-целое». В нём один объект входит в состав другого, и часть не может существовать без целого. Уничтожение целого приведёт к уничтожению всех связанных с ним композицией частей

В коде композиция предполагает, что объект класса-части никогда не будет инициализирован в программе вне класса-целого. Обычно класс-целое при инициализации создаёт экземпляр(ы) класса-части. Наиболее сильная композиция обеспечивается двусторонним сохранением ссылок друг на друга между экземплярами части и целого при их инциализации. Это происходит, например, между представлением и контроллером в архитектуре MVC.

class Engine

def start

...

end

end

class Car

def initialize

@engine = Engine.new

end

def start\_car

@engine.start

end

end

Объекты класса *Engine* не инициализируются в коде вне объектов класса *Car*, содержащих их. При этом, если будет уничтожен объект *Car*, то уничтожатся и все входящие в него объекты *Engine*.

**Вопрос 3. Расскажите, что такое ассоциация между классами, приведите частные случаи, покажите примеры. Расскажите, какие паттерны проектирования используют данное отношение. Покажите примеры работающего кода с реализацией данного отношения.**

 или

 (на концах указывается число экземпляров каждого класса, участвующих в связи)

**Ассоциация** – отношение между классами объектами классов, которое позволяет одному экземпляру объекта вызвать другой, чтобы выполнить действие от его имени.

Частными случаями ассоциации являются агрегация и композиция, но они подразумевают отношение «часть-целое», поэтому если ни к агрегации, ни к композиции связь не относится, то это означает, что обе сущности независимы друг от друга, и связь является простой ассоциацией.

В предметной области такие связи образуются между любыми сущностями, так или иначе взаимодействующих друг с другом, но при этом никак не зависящих друг от друга.

Например, студент может записаться на один и более курсов, но ни студент не зависит от курса, ни курс не зависит от студента.

В коде ассоциация предполагает, что свойство класса содержит ссылку на экземпляр (или экземпляры) другого класса. В контексте примера выше это выглядит так:

class Student

attr\_accessor :name, :courses

def initialize(name)

@name = name

@courses = []

end

def add\_course(course)

@courses << course

end

end

class Course

attr\_accessor :title

def initialize(title)

@title = title

end

end

Частными случаями ассоциации являются агрегация и композиция – они реализуют разные виды отношения «частное-целое».

**Агрегация**



Слабое отношение «часть-целое». В нём один объект входит в состав другого, но оба могут существовать независимо друг от друга.

В коде агрегирующий класс должен иметь в своём атрибуте ссылку на один или несколько (массив) экземпляров агрегируемого класса. При этом и тот, и другой могут иметь экземпляры, существующие вне описываемого контекста.

class Box

attr\_accessor :items

def initialize()

@items = []

end

def add\_item(item)

@items << item

end

end

class Item

...

end

Коробка содержит в себе предметы, но может существовать без них, как и предметы могут существовать без коробки.

**Композиция**



Сильное отношение «часть-целое». В нём один объект входит в состав другого, и часть не может существовать без целого. Уничтожение целого приведёт к уничтожению всех связанных с ним композицией частей.

В коде композиция предполагает, что объект класса-части никогда не будет инициализирован в программе вне класса-целого. Обычно класс-целое при инициализации создаёт экземпляр(ы) класса-части. Наиболее сильная композиция обеспечивается двусторонним сохранением ссылок друг на друга между экземплярами части и целого при их инциализации. Это происходит, например, между представлением и контроллером в архитектуре MVC.

class Engine

def start

...

end

end

class Car

def initialize

@engine = Engine.new

end

def start\_car

@engine.start

end

end

Объекты класса *Engine* не инициализируются в коде вне объектов класса *Car*, содержащих их. При этом, если будет уничтожен объект *Car*, то уничтожатся и все входящие в него объекты *Engine*.

**Ассоциацию** используют следующие паттерны:

1. Одиночка – класс-одиночка ассоциируется сам с собой засчёт использования метода, инициализирующего единственную инстансную переменную.

2. Адаптер хранит в себе экземпляр адаптируемого класса и использует его методы для формирования необходимого клиенту интерфейса.

3. Итератор двусторонне ассоциируется с конкретной коллекцией.

**Агрегацию** используют следующие паттерны:

1. Компоновщик – дерево-компоновщик агрегирует в себе набор компонентов, но они могут существовать вне дерева.

2. Декоратор – каждая обёртка агрегирует в себе другие обёртки и оборачиваемый класс, но все они могут существовать по отдельности.

3. Наблюдатель – наблюдаемый класс агрегирует в себе классы-наблюдатели (подписчики), но они могут существовать независимо.

4. Стратегия – класс, использующий стратегию, агрегирует её в своём атрибуте. При этом стратегия может существовать сама по себе (например, использоваться при этом другими классами).

**Композицию** используют следующие паттерны:

1. Посредник содержит в себе несколько различных компонентов, которые не могут существовать без самого посредника.

**Вопрос 4. Расскажите, как Вы понимаете принцип - предпочитайте делегацию наследованию. Покажите примеры кода, покажите, в каком случае имеет смысл использовать любое из отношений. Расскажите, какие паттерны иллюстрируют данный подход.**

Принцип «Предпочитайте делегацию наследованию» происходит из следующих причин:

- подкласс не может отказаться от интерфейса или реализации своего родителя. Необходимо реализовать абстрактные методы родителя, даже если они не нужны для конкретного подкласса.

- переопределяя методы родителя нужно помнить о том, чтобы не сломать базовое поведение суперкласса. Это исходит из принципа подстановки Барбары Лисков.

- наследование нарушает инкапсуляцию суперкласса, так как подклассам доступны детали родителя.

- подклассы слишком тесно связаны с родительским классом. Любое изменение в родителе может сломать поведение в подклассах.

- повторное использование кода через наследование может привести к разрастанию иерархии классов.

Делегация также позволяет решать проблемы множественного наследования, которое может вообще не поддерживаться некоторыми языками программирования, например Ruby.

Наследование необходимо использовать в тех случаях, когда чётко известны масштабы иерархии классов из предметной области, подкласс имеет оправданно жёсткую связь с родителем, должен повторять его логику и при изменении логики родителя должен соответственно менять и свою логику.

Наследование также используется для реализации абстрактных классов, не имеющих конкретной логики вне контекста наследников, например в реализации стратегий паттерна Стратегия и, в некоторых случаях, фабрик паттерна Фабричный метод.

Пример наследования:

class User

attr\_accessor :name, :email

def initialize(name, email)

@name = name

@email = email

end

def login

"#{name} вошел в систему"

end

def logout

"#{name} вышел из системы"

end

def profile

"Имя: #{name}, Email: #{email}"

end

end

class Admin < User

def manage\_users

"Администратор #{name} управляет пользователями"

end

end

Класс *Admin* должен выполнять все функции *User* без изменений, а также расширять его функционал методом *manage\_users*, поэтому здесь корректно использовать наследование.

Пример делегации (паттерн Стратегия):

class ShippingStrategy

def calculate\_cost(distance)

raise NotImplementedError, "Этот метод должен быть реализован в подклассе"

end

end

class GroundShipping < ShippingStrategy

def calculate\_cost(distance)

5 + distance \* 0.5

end

end

class AirShipping < ShippingStrategy

def calculate\_cost(distance)

20 + distance \* 1.0

end

end

class ShippingCalculator

attr\_accessor :strategy

def initialize(strategy)

@strategy = strategy

end

def calculate\_cost(distance)

@strategy.calculate\_cost(distance)

end

end

Паттерн Стратегия наглядно демонстрирует преимущество делегации над наследованием. В данном случае класс *ShippingCalculator* использует стратегию для расчёта одной функции. Без использования делегации пришлось бы создавать множество классов-наследников для каждого способа доставки, что создало бы много лишнего повторяющегося кода.

Другой паттерн, демонстрирующий данный принцип, - паттерн Декоратор. Без использования делегации в «обёртках» данного паттерна пришлось бы создавать огромное число наследуемых классов для каждой возможной комбинации, например «кофе с молоком», «кофе с шоколадом», «кофе с сиропом и шоколадом», «кофе с шоколадом и молоком» и т.д.

**Вопрос 5. Расскажите, какие Вам известны способы создания экземпляров классов, покажите примеры некачественного и качественного подхода(работающий код). Расскажите, какие паттерны проектирования, описывающие эти подходы, Вам известны.**

1. Стандартный способ – метод *new*. Он определён в классе *Class* и служит для создания новых экземпляров класса.

При вызове данного метода неявно вызывается метод *allocate*, выделяющий память под новый объект, а затем вызывается конструктор класса (*initialize*). Другим способом вызвать метод *initialize* вне класса нельзя. Если у класса нет явно определённого метода *initialize*, вызывается *initialize* класса-родителя.

Метод *initialize* отвечает за присваивание начальных значений переменным экземпляра и выполнение другой логики настройки. Аргументы, переданные в *new*, автоматически передаются в *initialize*.

2. Вышеупомянутый метод *allocate* выделяет память под новый объект, но не вызывает сам по себе метод *initialize*. Это позволяет разработчику при необходимости переопределять метод *new*. Таким образом можно получить точный контроль над логикой создания экземпляра. Пример:

class SomeClass

def self.new(\*args)

obj = allocate

# initialize is a private instance method by default!

obj.send(:initialize, \*args)

end

end

Здесь метод класса *new* расписан в явном виде. Между вызовами *allocate* и *initialize* можно добавить дополнительную логику, которую, например, неудобно или невозможно разместить в *intialize*.

3. В некоторых случаях появляется необходимость в специфичном создании экземпляров, без использования *new*. В частности, это используется в следующих паттернах проектирования:

Одиночка:

class SingletonExample

@instance = nil

private\_class\_method :new

def self.instance

@instance ||= new

end

def some\_method

...

end

end

Здесь метод *new* становится приватным, и обращение с единственным экземпляром класса (в том числе и создание) происходит через явно указанный метод *self.instance*.

Фабричный метод:

class Product

def initialize(name)

@name = name

end

def description

"This is product #{name}"

end

attr\_accessor :name

end

class Chair < Product

def description

"This is chair #{name}"

end

end

class Sofa < Product

def description

"This is sofa #{name}"

end

end

class Factory

def create\_product(name)

raise NotImplementedError # необязательно абстрактный

end

attr\_reader :product

private

attr\_writer :product

end

class ChairFactory < Factory

def create\_product(name)

Chair.new(name)

end

end

class SofaFactory < Factory

def create\_product(name)

Sofa.new(name)

end

end

Здесь различные фабрики для каждого типа продуктов реализуют свой метод *create\_product*, который берёт на себя создание продукта.

При выборе конкретного типа продукта создаётся соответствующая ему фабрика, имеющая стандартный набор свойств, но собственную реализацию фабричного метода.

Такой подход может быть необходим, если создание различных подклассов имеет различные нюансы, которые необходимо учитывать, и если неизвестно заранее, сколько разных подклассов вообще будет использоваться.

Фабричный метод лишь абстрагирует метод *new*, и необязательно запрещает им пользоваться, но обычно строго делегирует создание указанных объектов на себя.

В примитивной реализации фабричного метода не создаются отдельные подклассы фабрики, и сам фабричный метод имеет длинный условный оператор для каждого возможного варианта. Это некачественный подход, так как он загрязняет код и значительно уменьшает его масштабируемость – при добавлении новых вариантов будет необходимо добавлять в условный оператор новую проверку.

4. Экземпляры класса также неявно создаются методами *dup* и *clone*(из класса *Object*), выполняющими копирование объектов. При копировании через *clone*, в отличие от *dup*, также копируются метки его состояния, такие как *freeze* (при этом *clone* не является как таковым глубоким копированием).

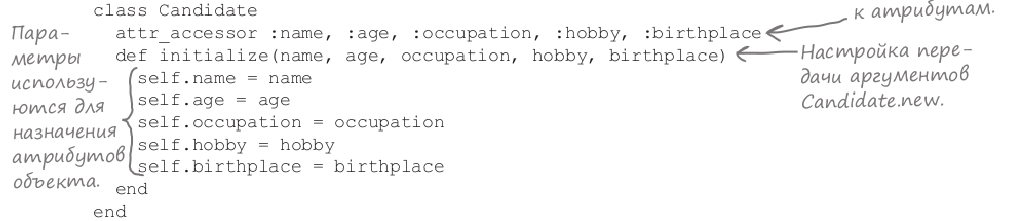
5. В Ruby литеральные объекты можно записывать без использования *new*, например:

x = 123

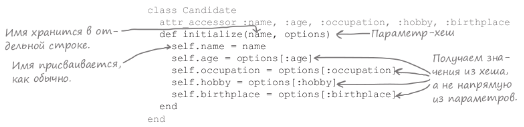
y = "abc"

Это возможно засчёт динамической типизации и интерпретируемости языка.

Из некачественных подходов к созданию экземпляров классов можно выделить написание конструктора с длинным рядом параметров, например:



Это решается выделением параметров в словарь (в Ruby - хэш)



Такой подход позволяет записывать необязательные параметры в любом порядке, и вообще пропускать необязательные параметры (В первом способе для этого необходимо писать конструкцию вида *age=nil*).

**Вопрос 6. Расскажите ООП структуру Вашего языка программирования. Расскажите, какие методы принято переопределять и в каком случае.**

В Ruby есть все элементы, которые принято ассоциировать с объектно-ориентированными языками: объекты с инкапсуляцией и сокрытием данных, методы с полиморфизмом и переопределением, классы с иерархией и наследованием. Но он идёт дальше, добавляя ограниченные возможности создания метаклассов, синглетные методы, модули и класс-примеси.

В Ruby все части кода являются объектами. Соответственно, любой блок кода возвращает некоторое значение.

О встроенных классах:

Ссылки на объекты хранятся в переменных. Сами переменные не имеют типа и не являются объектами, а лишь ссылаются на них. Тем не менее, небольшие неизменяемые объекты некоторых встроенных классов, например *Fixnum*, непосредственно копируются в переменные, которые на них ссылаются, так их хранить эффективнее. Во время присваивания таких объектов делается их копия, а куча не используется.

При присваивании одной переменной другой они начинают ссылаться на один и тот же объект, поэтому, если он изменяемый, то модификация значения одной переменной отразится на другой (например при замене одного из элементов массива). Однако присваивание нового значения переменной не влияет на другую.

Изменяемый объект можно сделать неизменяемым методом *freeze*.

Ruby также предоставляет работу с «Символами». Символ ведёт себя как неизменяемая строка, все вхождения которой ссылаются на одно и то же значение. Его можно преобразовать в строку методом *to\_s*, и он имеет вид:

*:symbol*

Модули и примеси:

Модуль представляет собой набор методов и констант, внешних по отношению к программе на Ruby. Его можно использовать просто для управления пространством имён, но основное применение модулей связано с «подмешиванием» его возможностей в класс (с помощью *include*). В таком случае он используется как класс-примесь.

Все классы в Ruby наследуются от класса *BasicObject*, содержащего основные методы классов и экземпляров, такие как *new, !, !=, ==, equal?, send* и другие.

Большинство пользовательских и встроенных классов наследуются от класса *Object*, который, в свою очередь, наследуется от *BasicObject* с подмешанной к нему примесью *Kernel*, содержащей так называемые методы ядра. Они включают в себя методы для

- преобразования в другие классы

- запросов конкретного состояния объекта (*block\_given?, class, frozen?, global\_variables, local\_variables* и другие)

- завершения процессов (*exit* и др.)

- работы с исключениями (*raise, catch, throw*)

- работы с файлами (*open, print, readline* и др.)

- создания лямбда-функций (*lambda, proc*)

- загрузки файлов Ruby (*require, load* и др.)

- псевдослучайных значений (*rand, srand*)

- и других

Сам класс *Object* включает в себя такие методы, как *is\_a?, ===, eql?, methods, nil?* и др.

Создание классов:

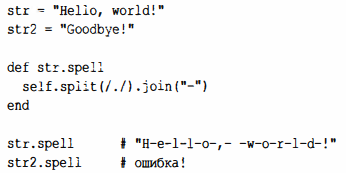
Строго говоря, классы в Ruby не имеют имён. Имя класса – это глобальная константа, ссылающаяся на объект типа *Class*. Поэтому на один и тот же класс могут ссылаться несколько констант и их можно присваивать переменным точно так же, как и с любыми другими объектами (*Class* – это и класс, и объект).

В классе можно определять константы, переменные класса, методы класса, переменные экземпляра и методы экземпляра. Данные уровня класса доступны всем объектам этого класса, тогда как данные уровня экземпляра доступны только одному объекту. Соответственно, данные класса могут получены без инициализации конкретного экземпляра.

Для определения геттеров и сеттеров в Ruby имеются методы *attr\_accessor, attr\_reader* и *attr\_writer*. Они заменяют методы вида *myvar* и *myvar=* в короткой записи с помощью символов.

По умолчанию все определённые в классе методы открыты, за исключением метода *initialize*. Переменные экземпляра всегда закрыты, к ним можно обращаться только с помощью методов доступа.

В Ruby есть возможность создавать методы на уровне объекта, а не класса. Такие методы называются синглетными и принадлежат единственному объекту, не оказывая влияния на другие объекты класса. Например:



Такое поведение реализуется с помощью метаклассов. Каждый объект в Ruby помимо своего класса также имеет скрытый метакласс (он же *singleton\_class*), описывающий поведение конкретного экземпляра. Именно к метаклассу добавляются синглетные методы объекта.

**Переопределение методов**:

1. Часто при создании классов необходимо переопределять его конструктор *initialize* для задания начальных параметров и/или передачи необходимых параметров.

2. Метод *to\_s* переопределяется для читабельного вывода информации о классе, например, в консоль (так, метод *puts* по умолчанию вызывает метод *to\_s* у класса перед выводом).

3. В аналогичных случаях может быть переопределён метод *inspect* для вывода информации о классе для отладки.

4. При подмешивании *Comparable* необходимо переопределить метод *<=>*. Это является обязательным условием для использования всех остальных методов данной примеси.

5. При подмешивании *Enumerable* необходимо переопределить метод *each*. Это является обязательным условием для использования всех остальных методов данной примеси.

6. Метод *method\_missing* можно переопределить в классе для логирования ошибочных вызовов несуществующих методов в классе и записи переданных при этом аргументов.

7. Если не используется *Comparable*, но нужно проверять равенство объектов классов, иногда необходимо переопределить методы *==* и *eql?*.

**Вопрос 7. Какие паттерны проектирования Вам известны, что это такое, как их принято классифицировать? Раскройте взаимосвязь паттернов проектирования, архитектурных паттернов и фреймворков. Покажите на примерах.**

**Паттерн проектирования** – это часто встречаемое решение определённой проблемы при проектировании архитектуры программ.

В отличие от готовых функций или библиотек, паттерн нельзя просто взять и скопировать в программу. Паттерн представляет собой не конкретный код, а общую концепцию или пример решения той или иной проблемы, которое нужно будет подстроить под нужды конкретной программы.

Среди паттернов обычно выделяются:

- идиомы. Они не очень универсальны, так как применимы только в рамках одного языка программирования. Например, инкремент, обычно имеющий вид *i = i+1,* в C-подобных языках может записываться как *i++* или *++i*, а обмен значениями переменных может выражаться в Python как *a, b = b, a*, в отличие от многих других языков, где это делается с созданием временной переменной.

- архитектурные паттерны. Они наиболее универсальны, их можно реализовать практически на любом языке. Описывают проектирование всей программы, а не отдельных её элементов. К ним относятся MVC, MVP, MVVM и другие.

Остальные паттерны обычно разделяют на 3 группы:

- порождающие паттерны – обеспечивают гибкое создание объектов без внесения в программу лишних зависимостей.

- структурные паттерны – показывают различные способы построения связей между объектами.

- поведенческие паттерны – обеспечивают эффективную коммуникацию между объектами.

Паттерны не придумывают, а скорее «открывают». Наиболее общепринятый набор изначальных паттернов приведён в книге «банды четырёх» - Эриха Гаммы, Ричарда Хелма, Ральфа Джонсона и Джона Влиссидеса. Она включает в себя описание 23 паттернов проектирования:

Порождающие:

- Абстрактная фабрика, Строитель, Фабричный метод, Прототип, Одиночка

Структурные:

- Адаптер, Мост, Компоновщик, Декоратор, Фасад, Приспособленец, Заместитель

Поведенческие:

- Цепочка обязанностей, Команда, Интерпретатор, Итератор, Посредник, Хранитель, Наблюдатель, Состояние, Стратегия, Шаблон, Посетитель

Говоря о взаимосвязи паттернов проектирования, архитектурных паттернов и фреймворков, стоит сказать следующее:

В общности фреймворк – это конкретный набор инструментов и технологий, которым пользуется программист для разработки приложений.

Архитектурный паттерн – это набор правил и методов создания данного приложения, то есть он не зависит от конкретной физической реализации и предоставляет лишь абстракцию, на основе которой фреймворки организуют свою структуру.

Другие паттерны проектирования предоставляют правила для оптимизации кода отдельных частей программы, они оперируют на уровне классов и объектов, тогда как архитектурные паттерны – на уровне целой программы.

Например, фреймворк Ruby on Rails предоставляет инструменты для разработки веб-приложений на Ruby на основе архитектурного паттерна MVC, разделяющего логику программы на модели, контроллеры и представления, а паттерн Наблюдатель, в числе многих других возможных, реализуется между конкретной моделью и представлением, например, для оповещения об изменениях базы данных, информация о которых была получена из DML-триггера БД.

**Вопрос 8. Расскажите, как составлять диаграмму классов, что на ней принято отмечать и каким образом. Раскройте все составляющие, покажите взаимосвязь диаграммы классов и кода.**

**Диаграмма классов** – структурная диаграмма языка моделирования UML, демонстрирующая общую структуру иерархии классов системы, их коопераций, атрибутов (полей), методов, интерфейсов и взаимосвязей (отношений) между ними.

Правила построения диаграммы классов:

- Классы представлены в рамках, содержащих 3 элемента:

1. В верхней части написано имя класса полужирным шрифтом

2. Посередине располагаются поля (атрибуты) класса. В некоторых случаях через двоеточие указывается тип поля.

3. В нижней части находятся методы класса. В некоторых случаях через двоеточие указывается тип возвращаемого значения.

- Имена всех абстрактных элементов (классов, методов) выделяются *курсивом*

- Классы-интерфейсы могут явно помечаться надписью «interface» над именем класса.

- Для задания видимости атрибутов и методов перед их именем указываются обозначения:

+ публичный

- приватный

# защищённый

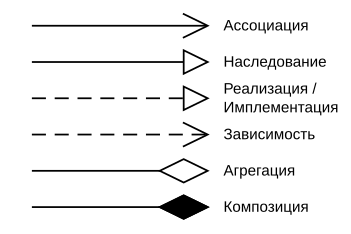
~ пакет

- UML определяет два типа областей действия для методов и атрибутов:

1. Экземпляр (методы и атрибуты объекта) – значения полей могут отличаться в разных объектах, методы могут изменять поля.

2. Классификатор (методы и атрибуты класса, в других языках - статические) – значения полей одинаковы для всех экземпляров, вызов метода не меняет состояние объекта. Их имена на диаграмме подчёркиваются.

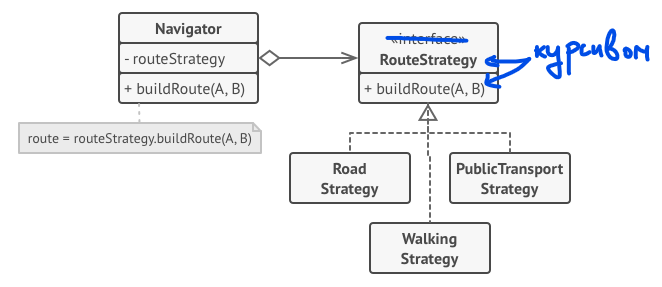
- Взаимосвязи между объектами устанавливаются согласно их отношениям:



- Отношения могут уточняться количеством участвующих экземпляров на линии, связывающих их, например 0..1, 1, 0..\* или \*, 1..\*.

Пример связи диаграммы классов с кодом на примере паттерна Стратегия:

Диаграмма:



Код:

class RouteStrategy

def buildRoute(A, B)

raise NotImplementedError

end

end

class RoadStrategy < RouteStrategy

def buildRoute(A, B)

...

end

end

class WalkingStrategy < RouteStrategy

def buildRoute(A, B)

...

end

end

class PublicTransportStrategy < RouteStrategy

def buildRoute(A, B)

...

end

end

class Navigator

def initialize(strategy)

@routeStrategy = strategy

end

def buildRoute(A, B)

@routeStrategy.buildRoute(A, B)

end

private

attr\_accessor :routeStrategy

end

walking\_strategy = WalkingStrategy.new

navigator = Navigator.new(walking\_strategy)

A = ...

B = ...

route = navigator.buildRoute(A, B)

Стратегия агрегируется в классе навигатора и имеет различные реализации.

**Вопрос 9. Какие структурные паттерны проектирования Вам известны? Перечислите и расскажите базовые принципы всех паттернов. Раскройте подробно два паттерна, покажите проблему, покажите предлагаемое решение на диаграммах и в коде. Укажите связь этих паттернов с другими известными Вам паттернами.**

В книге «банды четырёх» приведены следующие 7 структурных паттернов проектирования:

- Адаптер – позволяет объектам с несовместимыми интерфейсами работать вместе засчёт создания класса-адаптера, трансформирующего интерфейс или данные одного объекта в вид, понятный другому объекту.

- Мост – разделяет один или несколько классов на отдельные иерархии – абстракцию и реализацию, позволяя изменять их независимо друг от друга. В основе паттерна лежит замена наследования на агрегацию или композицию – абстрактный класс (например, Геометрическая фигура) содержит в себе экземпляр конкретного класса-реализации (например, Цвет), получая тем самым гибкую систему (например, Синий квадрат, Красный треугольник).

- Компоновщик – позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, будто это единичный объект. Имеет смысл только тогда, когда основная модель программы может быть структуирована в виде дерева. Такое хранение объектов даёт возможность быстро рекурсивно обращаться ко всем «листьям» структуры.

- Декоратор – позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

- Фасад – предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку. Он может иметь урезанный интерфейс, не имеющий полной функциональности внешнего инструмента, но предоставляет именно те функции, которые нужны клиенту, и скрывает остальные.

- Приспособленец (Легковес) – позволяет вместить большее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо храненипя одинаковых данных в каждом объекте. Каждый отдельный объект изменяет только «внутреннее состояние», тогда как «внешнее состояние», одинаковое для однотипных объектов, хранится в отдельном классе.

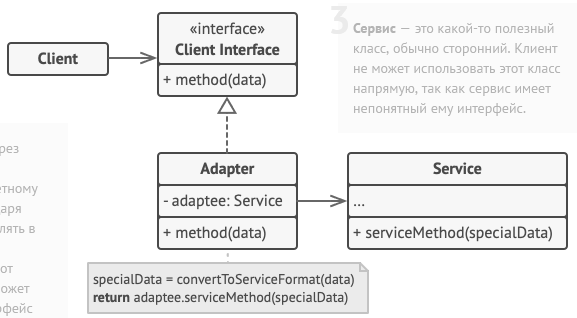
- Заместитель – позволяет предоставлять вместо реальных объектов специальные объекты-заместители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то до или после передачи вызова оригиналу, при этом объекты-заместители имеют тот же интерфейс, что и объекты – оригиналы, поэтому их можно внедрять незаметно для клиента.

**Подробное описание паттернов**

**Адаптер**

Проблема: имеется некоторый внешний класс, который проблематично, либо вообще невозможно изменить в контексте задачи. Наш клиентский класс должен использовать его функционал, но не может использовать предоставляемый внешним классом интерфейс (например, мы можем использовать только данные в формате XML, а внешний класс возвращает только JSON).

Решение: вводится класс-адаптер. Он имеет интерфейс, совместимый с клиентом, и берёт на себя преобразование результатов логики внешнего класса в нужный вид. Клиент обращается к методу адаптера, адаптер обращается к внешнему классу, попутно добавляя свою логику, и возвращает нужное значение клиенту.



class Service

def service\_method(data)

...

end

end

class Adapter

def initialize(adaptee)

@adaptee = adaptee

end

def method(data)

result = @adaptee.service\_method(data)

convert\_to\_special\_format(result)

end

private

attr\_accessor :adaptee

end

# Вызов метода адаптера вместо оригинального класса в клиентском коде

service = Service.new

service\_adapter = Adapter.new(service)

data = ...

result = service\_adapter.method(data)

Связь с другими паттернами:

- Мост проектируют заранее, чтобы развивать большие части приложения отдельно друг от друга. Адаптер применяют постфактум, чтобы заставить несовместимые классы работать вместе.

- Адаптер предоставляет совершенно другой интерфейс для доступа к существующему объекту, тогда как Декоратор либо оставляет прежний интерфейс, либо расширяет его.

- Фасад задаёт новый интерфейс, тогда как Адаптер повторно использует старый. Адаптер оборачивает только один класс, а Фасад оборачивает целую подсистему. Кроме того Адаптер позволяет двум интерфейсам работать сообща, вместо того, чтобы задать полностью новый.

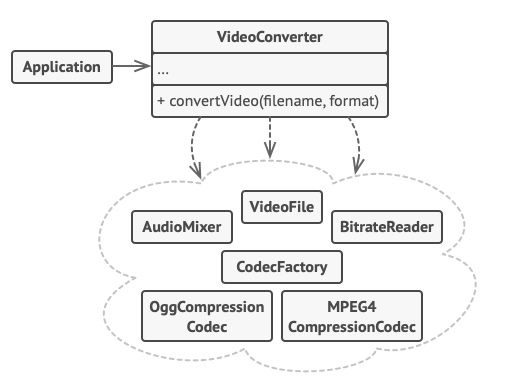
- Как и Мост, Стратегия и Состояние, Адаптер основан на делегировании работы другим объектам.

**Фасад**

Проблема: коду приходится работать с большим количеством объектов некой сложной библиотеки или фреймворка. Нужно самостоятельно инициализировать эти объекты, следить за правильным порядком зависимостей и так далее. В результате бизнес-логика тесно переплетается с деталями реализации сторонних классов. Такой код сложно понимать и поддерживать.

Решение: создаётся класс-фасад. Он имеет относительно простой интерфейс и берёт на себя работу со сложной подсистемой. Ненужные функции он не использует (абстрагирует от пользователя), и оставляет именно то, что нужно конечному пользователю. Он может включать в себя различные функции, добавлять свою логику и как угодно манипулировать данными внешней системы для получения наиболее эффективного и полезного результата в контексте решаемой задачи.

Пример конвертатора видео, работающего с различными конкретными классами обработки видеофайлов:



# Классы сложного стороннего фреймворка конвертации видео.

class VideoFile

# ...

end

class OggCompressionCodec

# ...

end

class MPEG4CompressionCodec

# ...

end

class CodecFactory

def extract(file)

# ...

end

end

class BitrateReader

def self.read(filename, codec)

# ...

end

def self.convert(buffer, codec)

# ...

end

end

class AudioMixer

def fix(result)

# ...

end

end

# Фасад

class VideoConverter

def convert(filename, format)

file = VideoFile.new(filename)

source\_codec = CodecFactory.new.extract(file)

destination\_codec = if format == "mp4"

MPEG4CompressionCodec.new

else

OggCompressionCodec.new

end

buffer = BitrateReader.read(filename, source\_codec)

result = BitrateReader.convert(buffer, destination\_codec)

result = AudioMixer.new.fix(result)

File.new(result)

end

end

# Клиентский класс

class Application

def main

converter = VideoConverter.new

mp4 = converter.convert("funny-cats-video.ogg", "mp4")

mp4.save

end

end

Связь с другими паттернами:

- Фасад задаёт новый интерфейс, тогда как Адаптер повторно использует старый.

- Легковес показывает, как создавать много мелких объектов, а Фасад показывает, как создать один объект, отражающий целую подсистему.

- Фасад можно сделать Одиночкой, так как обычно нужен только один объект-фасад.

- Фасад похож на Заместитель тем, что замещает сложную подсистему и может сам её инициализировать. Но, в отличие от Фасада, Заместитель имеет тот же интерфейс, что и его служебный объект, что позволяет использовать его без подстраивания остальной программы под новый интерфейс.

**Вопрос 10. Какие порождающие паттерны проектирования Вам известны? Перечислите и расскажите базовые принципы всех паттернов. Раскройте подробно два паттерна, покажите проблему, покажите предлагаемое решение на диаграммах и в коде. Укажите связь этих паттернов с другими известными Вам паттернами.**

В книге «банды четырёх» приведены следующие 5 порождающих паттернов проектирования:

- Фабричный метод – определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов. В примитивной реализации это один метод, заменяющий new для создания различных видов объектов и добавляющий необходимую логику. В более корректной реализации для каждого вида создаётся наследник, по-своему переопределяющий фабричный метод.

- Абстрактная фабрика – позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов. Создаётся абстрактная фабрика с определённым интерфейсом создания отдельных объектов, а затем конкретные фабрики для каждого семейства объектов.

- Строитель – позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов. Он предлагает разбить процесс конструирования сложного объекта на отдельные шаги, которые можно выполнять по необходимости.

- Прототип – позволяет копировать объекты, не вдаваясь в подробности их реализации. Поскольку извне нельзя получить, например, приватные поля класса, Прототип поручает копирование объектов самим объектам. Опционально внешний класс – хранилище прототипов может хранить в себе эти прототипы, которые затем будут легко клонироваться.

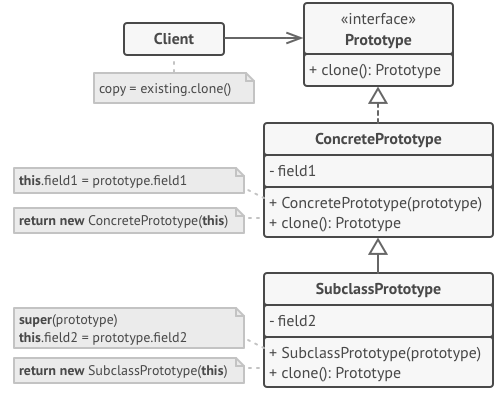
- Одиночка – гарантирует, что у класса есть только один экземпляр и предоставляет к нему глобальную точку доступа.

**Подробное описание паттернов**

**Прототип**

Проблема: есть объект, который нужно скопировать. Для этого нужно создать новый объект такого же класса, а затем поочерёдно скопировать значения всех полей из старого объекта в новый. Однако у такого подхода есть две проблемы: первая – часть состояния объекта может быть приватной, а значит недоступной для остального кода программы; вторая – копирующий код в таком случае будет зависим от конкретного кода класса, а не от его интерфейса.

Решение: Прототип поручает создание копий самим объектам. Он вводит общий интерфейс для всех объектов, поддерживающих клонирование (обычно он включает только метод *clone*). Копируемый объект и становится прототипом. Опционально можно задать внешний класс – хранилище прототипов, которое будет хранить в себе эти прототипы, которые затем будут легко клонироваться (наглядный пример – префабы в Unity).



class Prototype

def initialize(field1)

@field1 = field1

end

def clone

Prototype.new(@field1)

end

private

attr\_accessor :field1

end

class SubPrototype < Prototype

def initialize(field1, field2)

@field1 = field1

@field2 = field2

end

def clone

SubPrototype.new(@field1, @field2)

end

private

attr\_accessor :field2

end

Связь с другими паттернами:

- Прототип не опирается на наследование, но ему нужна сложная операция инициализации. Фабричный метод, наоборот, построен на наследовании, но не требует сложной реализации.

- Архитектура, построенная на Компоновщиках и Декораторах часто может быть улучшена засчёт внедрения Прототипа. Он позволяет клонировать сложные структуры объектов, а не собирать их заново.

- Прототип, как и Абстрактная фабрика и Строитель, может быть реализован с помощью Одиночки.

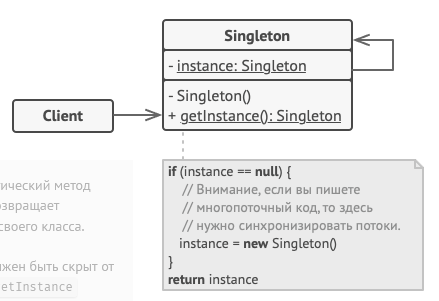
**Одиночка**

Проблема: Одиночка решает две проблемы, нарушая таким образом принцип единственной ответственности класса:

1. Гарантирует наличие единственного экземпляра класса. То есть при создании объекта дальнейшая попытка создания вернёт лишь существующий объект. Это может быть полезно для доступа к общему ресурсу, например, базе данных.

2. Предоставляет глобальную точку доступа. В отличие от простой глобальной переменной, такой подход защищает объект от перезаписи.

Решение: все реализации Одиночки сводятся к тому, чтобы скрыть конструктор по умолчанию и создать публичный метод класса, который и будет контролировать жизненный цикл объекта-одиночки.



class Singleton

@instance = new

private\_class\_method :new

# Статический метод, управляющий доступом к экземпляру одиночки.

def self.instance

@instance

end

def some\_business\_logic

...

end

end

Связь с другими паттернами:

- Фасад можно сделать Одиночкой, поскольку обычно нужен только один его объект.

- Абстрактная фабрика, Строитель и Прототип могут быть реализованы при помощи Одиночки.

- Легковес может быть приведён к подобию Одиночки, но имеет 2 важных отличия: можно создавать много объектов-легковесов, и они должны быть неизменяемыми, в отличие от Одиночки, допускающего изменение своего состояния.

Важно заметить, что с паттерном Одиночкой связан ряд проблем: он нарушает принцип единственной ответственности класса, может маскировать плохой дизайн, сталкивается с проблемами при мультипоточности и требует постоянного создания Mock-объектов при юнит-тестировании.

По этим причинам Одиночку часто относят к антипаттернам, поэтому его всё реже и реже можно встретить в реальном коде.

**Вопрос 11. Какие поведенческие паттерны проектирования Вам известны? Перечислите и расскажите базовые принципы всех паттернов. Раскройте подробно два поведенчиских паттерна, покажите проблему, покажите предлагаемое решение на диаграммах и в коде. Укажите связь этих паттернов с другими известными Вам паттернами.**

В книге «банды четырёх» приведены следующие 11 поведенческих паттернов проектирования:

- Цепочка обязанностей – 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.

- Команда – превращает запросы в объекты, позволяя передавать их как аргументы при вызове методов, ставить запросы в очередь, логировать их, а также поддерживать отмену операций. Похож на Стратегию, но если Стратегия выделяет решение проблемы в отдельный объект для переключения между разными решениями, Команда выделяет решение в отдельный объект для разных целей, в том числе для смены проблемы как таковой.

- Интерпретатор – редкоиспользуемый паттерн, применяемый для определения грамматики простого языка, представления и интерпретации предложений на этом языке. Позволяет парсить сложные выражения, передавая логику выполнения выражения конкретному подклассу класса-интерпретатора.

- Итератор – даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

- Посредник – позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

- Хранитель (Снимок) – позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации. Объекты сами могут создавать «снимки» - объекты данных своего текущего состояния со всеми приватными полями, по которым затем они же смогут к нему вернуться. Могут также использоваться «объекты-опекуны», хранящие стек снимков для дальнейшего последовательного восстановления.

- Наблюдатель – создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах. Класс, за которым наблюдают, хранит в себе список всех подписчиков и метод для оповещения их об изменении своего состояния.

- Состояние – позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта. Является надстройкой над Стратегией в том плане, что идентичен в реализации, за исключением того, что класс-состояние может менять контекст объекта, в том числе менять и само состояние.

- Стратегия – определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

- Шаблон – определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.

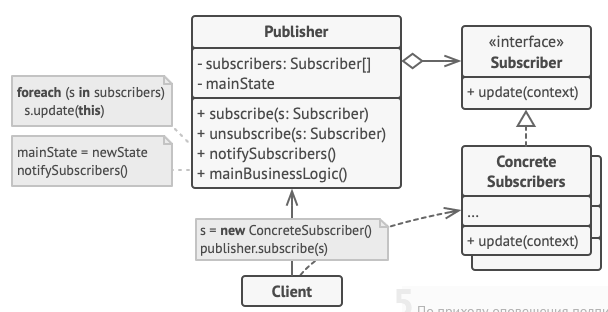
- Посетитель – позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.

**Подробное описание паттернов**

**Наблюдатель**

Проблема: один объект (*наблюдатель*) должен получать своевременную информацию об изменении другого объекта (*издателя*). С одной стороны, *наблюдатель* может с некоторым интервалом посылать запросы *издателю*, но это занимает лишние ресурсы, и *наблюдатель* может пропустить некоторое время после обновления *издателя*. С другой стороны, *издатель* может посылать информацию о своём изменении всем подряд связанным с ним объектам, но это может быть не нужно никакому объекту кроме *наблюдателя*.

Решение: в класс-*издатель* добавляется механизм подписки: он включает в себя поле с массивом всех подписчиков-*наблюдателей*, методы для их добавления и удаления и метод для их оповещения (выполнения какой-либо логики *наблюдателя* в соответствии с изменением).



class Observer

def update

...

end

end

class Publisher

def initialize

@subscribers = []

end

def add\_subscriber(subscriber)

@subscribers << subscriber

end

def remove\_subscriber(subscriber)

@subscribers.delete(subscriber)

end

def notify

subscribers.each do |subscriber|

subscriber.update

end

end

private

attr\_accessor :subscribers

end

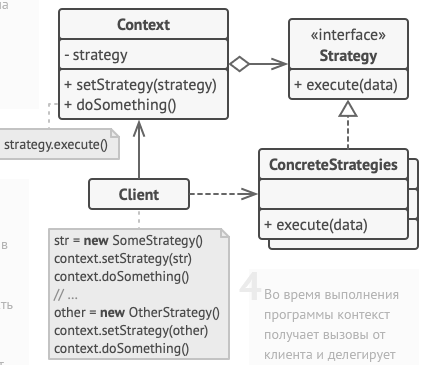
Связь с другими паттернами:

- Разница между Посредником и Наблюдателем не всегда очевидна. Чаще всего они выступают как конкуренты, но могут работать вместе. Цель Посредника – убрать обоюдные зависимости между компонентами, делая их зависимыми от самого посредника. Цель Наблюдателя – обеспечить динамическую одностороннюю связь, в которой одни объекты косвенно зависят от других. Посредник можно реализовать при помощи Наблюдателя. Тогда класс-посредник будет выступать *издателем*, а остальные компоненты – *наблюдателями*.

**Стратегия**

Проблема: нужно, чтобы поведение некоторых методов класса менялось в зависимости от контекста его работы. Можно попробовать обрабатывать эти разные контексты, но при добавлении новых вариантов класс может сильно раздуться.

Решение: алгоритмы решения отдельных задач, разнящихся от контекста, выносятся в классы-стратегии, имеющие общий интерфейс. Далее в основной класс добавляется поле, хранящее текущюю стратегию, и логика всех методов, делегированных стратегии, заменяется на вызов одноимённого метода в текущей стратегии.



class Strategy

def execute(data)

raise NotImplementedError

end

end

class StrategyA < Strategy

def execute(data)

# Логика стратегии A

end

end

class StrategyB < Strategy

def execute(data)

# Логика стратегии B

end

end

def MainClass

def initialize(strategy)

@strategy = strategy

end

def set\_strategy(strategy)

@strategy = strategy

end

def execute(data)

strategy.execute(data)

end

private

attr\_accessor :strategy

end

Связь с другими паттернами:  
- Шаблон использует наследование, чтобы расширять части алгоритма. Стратегия использует делегирование, чтобы изменять выполняемые алгоритмы на лету.

- Команда похожа на Стратегию, но отличается масштабом и применением: Стратегия описывает взаимозаменяемые способы решения одной проблемы. Команда превращает разнородные действия в объекты для разных целей: логирования, хранения в истории, передачи во внешние сервисы и т.д.

- Стратегия изменяет поведение объекта «внутри», а Декоратор изменяет его «снаружи».

- Стратегия почти идентична в принципе работы Состоянию. Однако в Стратегии объекты не знают друг о друге и никак не связаны, тогда как в Состоянии сами конкретные состояния могут переключать состояния в объекте.

**Вопрос 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.**

**Сравнение на равенство**:

Для сравнения двух объектов на равенство в Ruby представлено 4 различных метода:

1. *==*

Обычное равенство (generic equality). Сравнивает значения двух объектов. В пользовательских классах переопределяется наиболее часто для указания необходимой логики сравнения на равенство двух объектов. На уровне *Object* возвращает *true* только если сравнивается один и тот же объект.

2. *===*

«Case equality». Может выполнять обычное сравнение *==*, но имеет несколько другое значение и применение.

Во-первых, не является коммутативной операцией.

Во-вторых, определяет, принадлежит ли правое значение контексту левого. Другими словами, может быть использован для проверки объекта на принадлежность диапазону (например, *2..5*), типу данных (например, *Integer*), значение *true* при подстановке в лямбда-функцию (например, *lambda {|x| x < 5}*), а также удовлетворение регулярному выражению {например, */d+/*}.

Идентичным оператором является *=?*, но он, как правило, используется в регулярных выражениях.

3. *eql?*

Хэш-равенство. Определяет, одинаковы ли объекты в контексте хэшей (то есть когда они используются в качестве ключей в хэшах (*Hash*) или в элементах множеств (*Set*)).

Среди встроенных типов отличия *eql?* от *==* наиболее выражены в числовых типах. Если *==* выполняет преобразования типов (*1 == 1.0* будет равняться *true*), то *eql?* этого не делает (*1.eql? 1.0* равно *false*).

4. *equal?*

Никогда не должен переопределяться в подклассах. Сравнивает идентичность объектов (является ли объект a объектом b?). Другими словами, определяет, указывают ли переменные на одну и ту же ячейку в памяти.

Сравнение объектов на «больше-меньше» в Ruby традиционно реализуется подмешиванием модуля *Comparable*. Он предоставляет все основные методы сравнения объектов. Для корректной работы всех его методов в классе достаточно переопределить операцию *<=>* («космический корабль»).

При реализации сравнения двух объектов пользовательских классов могут возникнуть определённые проблемы:

- Может понадобиться сравнивать по разным полям класса. В таком случае можно воспользоваться паттерном Стратегия, вынеся оператор *<=>* в классы-стратегии.

- Объекты, построенные с помощью паттерна Декоратор, могут накладывать дополнительные условия для сравнения (например, по нескольким полям).

- Цепочка обязанностей может быть использована, если сравнение полей нужно выполнять в определённом порядке или до тех пор, пока какое-то условие не будет выполнено.

- Может понадобиться сравнивать множество объектов разных классов с разной логикой сравнения (например, *Куб*, *Шар*, *Конус*, нет возможности напрямую получить их объём). Во избежание раздувания кода условными операторами сравнение можно выделить в Посредник, содержащий логику сравнения всех объектов.

Сравнение объектов на «больше-меньше» в пользовательских классах наглядно демонстрирует принцип ***Абстракции*** ООП.

Если у нас имеется класс *Шар*, то довольно очевидно, что один шар больше другого, если он имеет больший радиус. Изначально, чтобы сравнить два шара, нам нужно было лезть в их переменные радиуса и сравнивать их. При подмешивании *Comparable* и переопределении оператора *<=>* в классе мы помещаем в него логику сравнения по радиусу, тем самым разграничивая слой абстракции и слой реализации. Теперь пользователь может напрямую спросить: «Какой шар больше?», а не делать это через реализацию «Какой из двух шаров имеет больший радиус?».

Этот же пример показывает и использование принципа ***Инкапсуляции***, а точнее *Сокрытия*. Предоставляя внешнему миру лишь оператор сравнения, мы защищаем важную для класса переменную радиуса от несанкционированного доступа.

Кроме того, этот пример демонстрирует принцип ***Полиморфизма***. Добавляя всем классам один набор операций из модуля *Comparable*, мы организуем для них единый интерфейс, позволяющий работать с ними, не зная о конкретной реализации. Классы *Шар*, *Куб* и *Конус* могут все иметь набор операторов сравнения, при этом пользователю не нужно заботиться о вычислении размеров для каждой отдельной геометрической фигуры.

**Вопрос 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.**

**Диаграмма последовательностей** – UML-диаграмма, на которой для некоторого набора объектов на единой временной оси показан жизненный цикл объекта (создание-деятельность-удаление) и взаимодействие акторов (действующих лиц) информационной системы в рамках прецедента (конкретной последовательности действий в некотором целостном фрагменте программы, описывающей функциональное требование).

Диаграмма последовательностей включает в себя:

- объекты – обозначаются прямоугольником с названием в центре, располагающимся вверху диаграммы (в некоторых версиях диаграммы могут иметь разные обозначения в зависимости от типа объекта).

Если в качестве объекта рассматривается некоторый класс программы, и имя объекта не определено или несущественно, то оно указывается на диаграмме в формате *:<имя\_класса>*. Если же объект имеет конкретное имя, то формат *<имя\_объекта>:<имя\_класса>*.

- линии жизни – пунктирные вертикальные линии, проведённые от каждого объекта, обозначающие течение времени (сверху вниз).

- блоки активности – прямоугольники на линиях жизни, обозначающие одно законченное действие (выполнение функции) объекта.

- сообщения – показывают обмен информацией между объектами. Обозначаются стрелками, подписанными именем действия. Имеют разновидности, основные среди которых:

1. *синхронное сообщение* – отправитель передаёт управление получателю и не может совершать никакие другие действия, пока действие получателя не будет завершено (то есть не будет получено ответное сообщение). 

2. *ответное сообщение* – ответ на синхронное сообщение. Обычно содержит какое-либо возвращаемое значение и возвращает управление актору-отправителю.

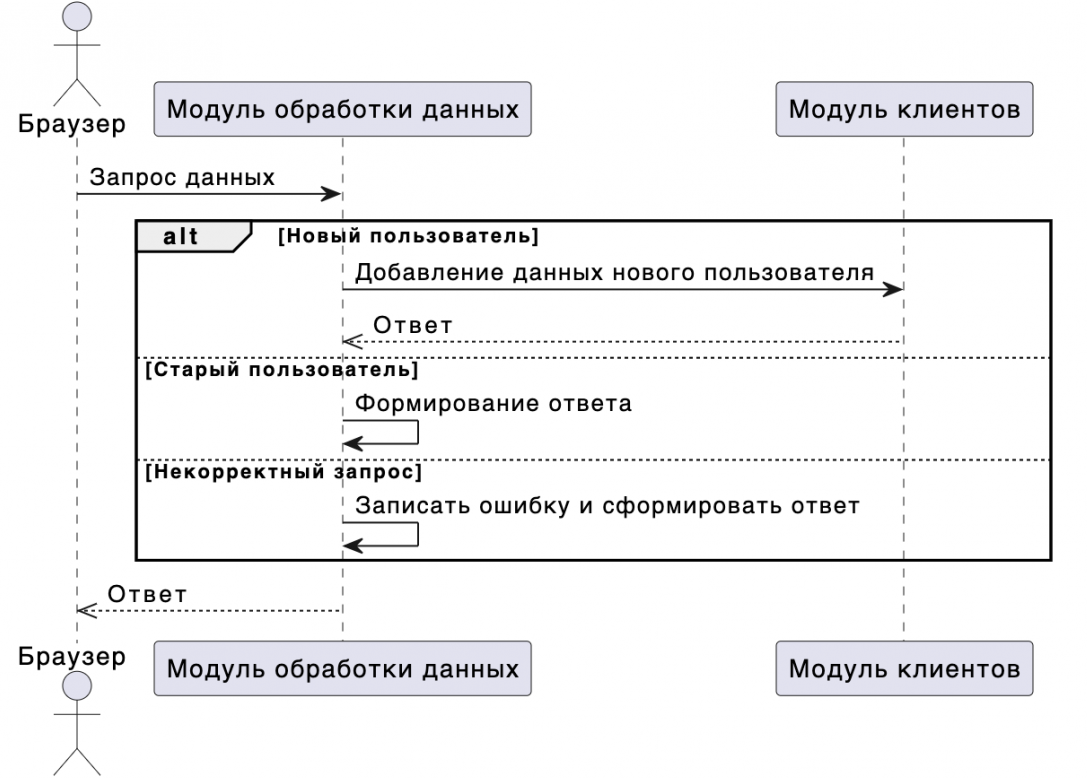


3. *асинхронное сообщение* – отправитель передаёт ход управления получателю, но и сам не теряет возможности совершать другие действия.

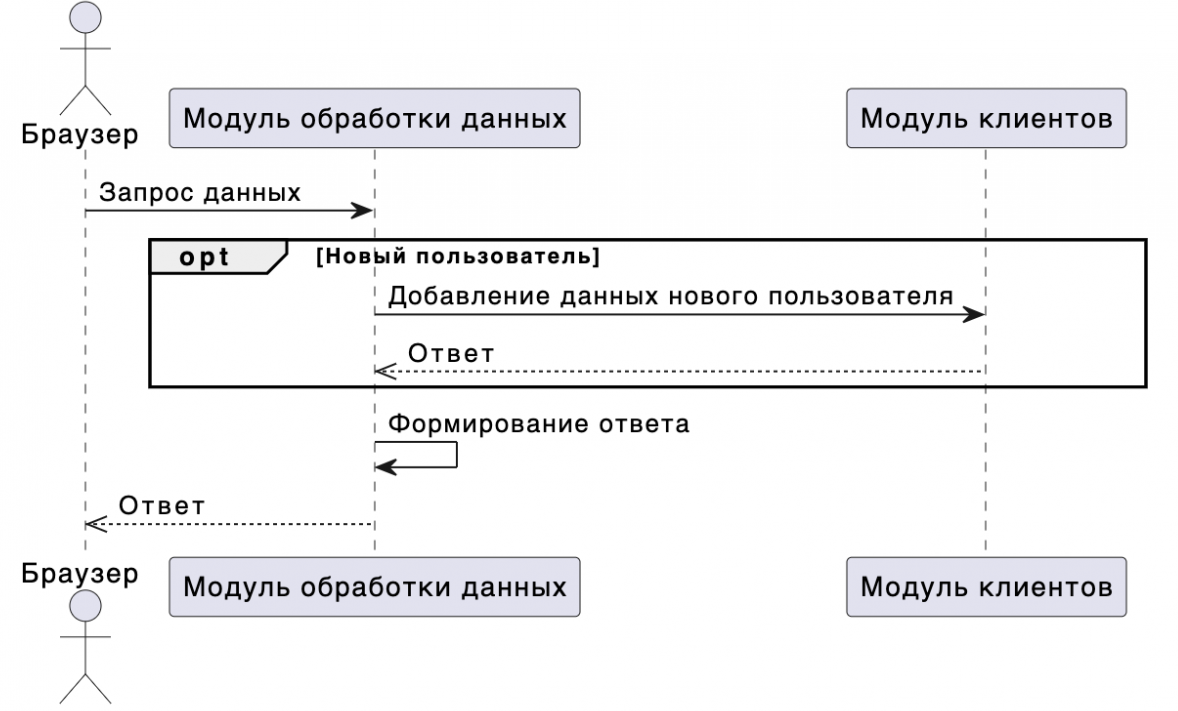


Отдельные наборы сообщений могут группироваться по разным причинам:

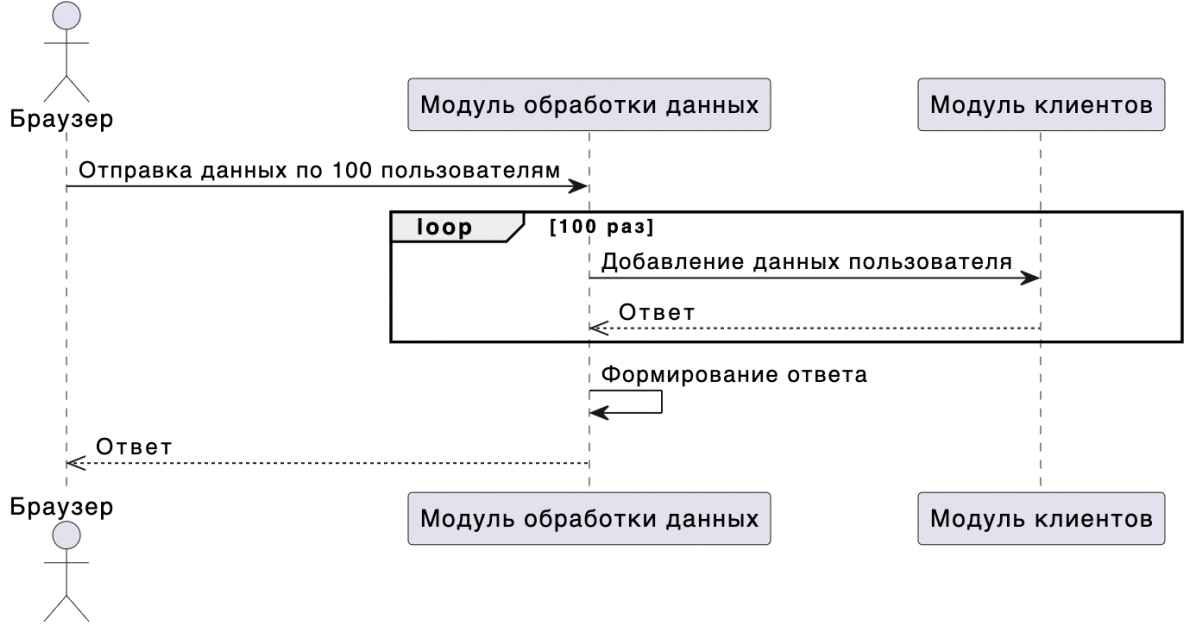
1. Альтернативные варианты



2. Опциональные действия

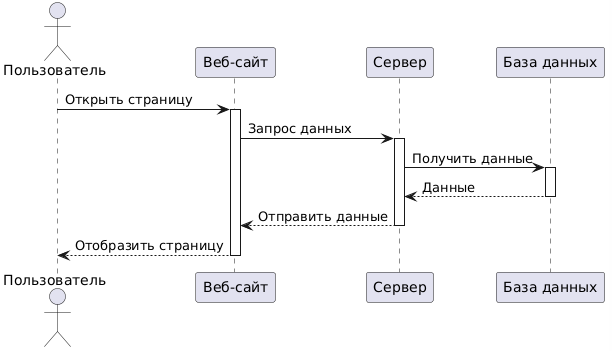


3. Циклы

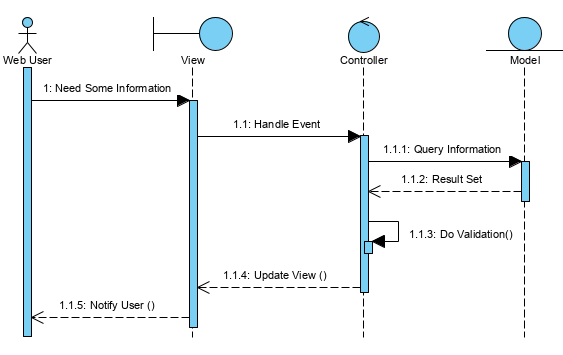


...и другие.

Тривиальный пример диаграммы последовательности:



Простейшая диаграмма последовательности в рамках паттерна MVC выглядит следующим образом:



**Вопрос 14. Архитектурный паттерн MVC. Расскажите базовый подход, расскажите, какие паттерны проектирования используются при его реализации. Покажите взаимодействие объектов в рамках реализации некоторых операций CRUD на диаграмме последовательностей. Покажите, в каких фреймворках он применяется.**

Основная цель всех архитектурных паттернов – разделение ответственности, повторное использование кода, тестируемость. Если мы хотим изменить что-то, мы не должны ходить по разным участкам кода. Без разделения ответственности ни повторное использование кода, ни тестируемость практически невозможно реализовать.

Термин **модель-представление-контроллер** используется с конца 70-х гг. прошлого столетия. Эта модель стала результатом проекта Smalltalk в компании Xerox PARC, где она была задумана как способ организации некоторых из ранних приложений графического пользовательского интерфейса.

MVC разделяет программу на 3 части:

1. Модели содержат или представляют данные, с которыми работают пользователи. Они могут быть простыми моделями представлений, которые только представляют данные, передаваемые между представлениями и контроллерами; или же они могут быть моделями предметной области, которые содержат бизнес-данные, а также операции, преобразования и правила для манипулирования данными.

2. Представления применяются для визуализации некоторой части модели в виде пользовательского интерфейса. Не содержат никакой логики программы, кроме визуализации и передачи данных. Предназначены только для получения и отображения данных пользователя.

3. Контроллеры обрабатывают поступающие запросы, выполняют операции с моделью и выбирают представления для визуализации пользователю.

В классическом представлении MVC контроллер является Стратегией для представления, т.е. одно представление хранит в себе ссылку на контроллер и может работать с несколькими разными контроллерами, имеющими одинаковый интерфейс.

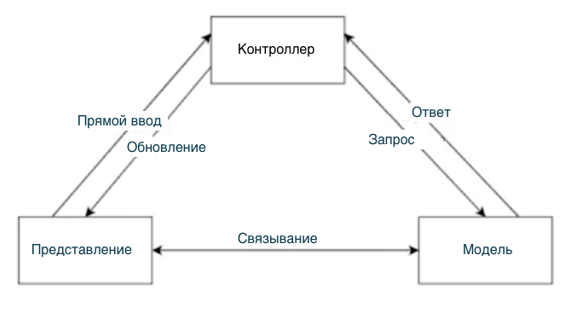
Архитектуру MVC можно охарактеризовать двумя пунктами:

1. Представление – это визуальная проекция модели.

2. Контроллер – это соединение между пользователем и системой.

Имеется 2 подхода к реализации MVC:

1. **Контроллер-супервизор**



Основная особенность: представление связано с моделью.

Идея: разделение между вводом и выводом.

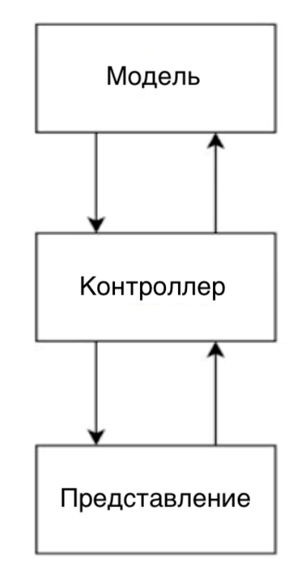
Преимущество: меньше кода

Недостатки: сложное unit-тестирование, сложная инкапсуляция, слабое разделение ответственности.

Для связи модели и представления используется паттерн Наблюдатель: представление подписывается на модель, и модель напрямую вызывает изменение представления при своём обновлении. Таким образом, представление может, наблюдая за моделью, динамически отображать хранящиеся в ней данные.

Хорош для небольших проектов, плохо масштабируется.

2. **Пассивное представление**



Основная особенность: представление полностью управляется контроллером.

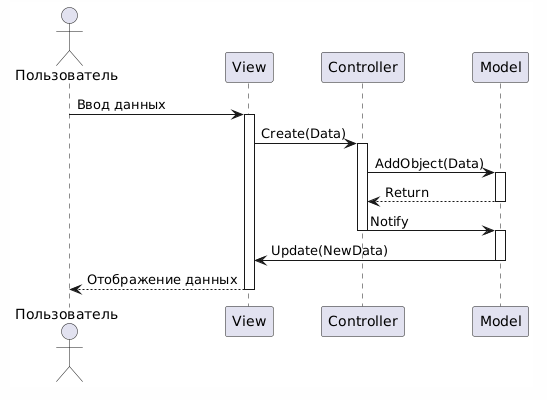
Идея: разделение между бизнес-логикой и логикой отображения.

Преимущество: облегчение тестирования, хорошая читаемость кода.

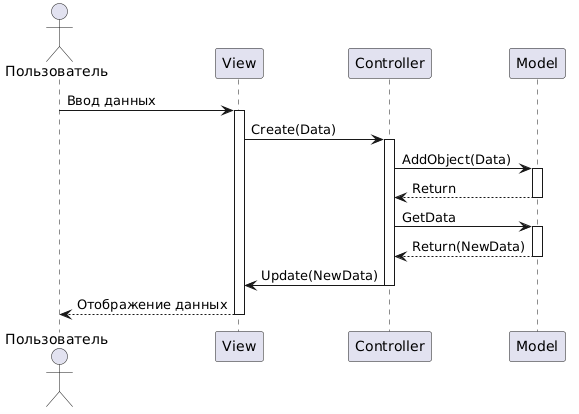
Недостатки: больше кода, необходимость использования контроллера для любых взаимодействий между представлением и моделью.

Примеры диаграмм последовательности для операции создания в MVC:

- Для контроллера-супервизора:



- Для пассивного представления:



MVC применяется в следующих фреймворках:

- в качестве GUI-фреймворка: Cocoa Framework, Java Swing, Qt4, GTK+

- в качестве веб-фреймворка:

Java: JSF, Oracle App Framework, Play Framework

C#: ASP.NET MVC Framework, PureMVC

Ruby: Ruby on Rails

**Вопрос 15. Архитектурный паттерн MVP. Расскажите проблематику в двух подходах в MVC, которая привела к появлению данного архитектурного паттерна. Расскажите базовый подход. Покажите взаимодействие объектов в рамках реализации некоторых операций CRUD на диаграмме последовательностей. Покажите, в каких фреймворках он применяется.**

Основная цель всех архитектурных паттернов – разделение ответственности, повторное использование кода, тестируемость. Если мы хотим изменить что-то, мы не должны ходить по разным участкам кода. Без разделения ответственности ни повторное использование кода, ни тестируемость практически невозможно реализовать.

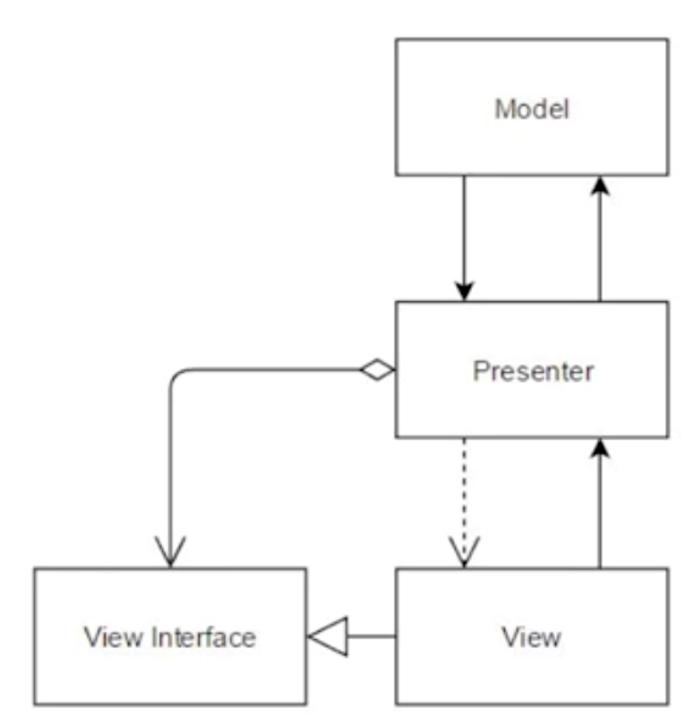
MVP продолжает идею MVC, но решает некоторые проблемы, возникающие при создании нескольких представлений и контроллеров, что может понадобиться во многих сложных приложениях.

Представления можно разделить или определить субпредставления с их собственными контроллерами, однако при таком подходе появляются проблемы:

- объединение логики отображения и бизнес-логики

- трудности при тестировании

Решение этих проблем кроется в создании абстрактного интерфейса для представления – это и есть главная **MVP**.



Основной подход MVP состоит в создании абстрактного интерфейса для представления. В него выносятся заранее определённые интерфейсы View, с которым и работает Presenter. По сути единственным отличием MVC от MVP является способ структурирования View.

Таким образом, логика View полностью закрыта за интерфейсом.

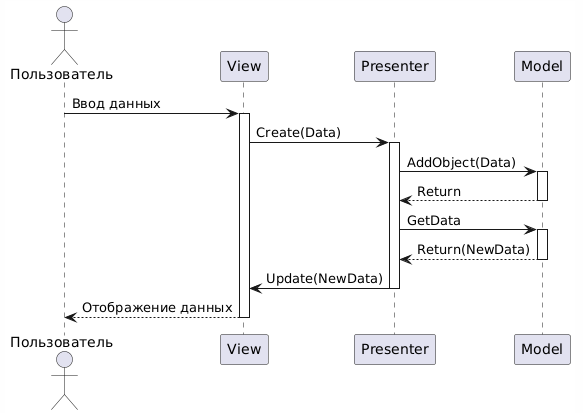
Presenter полностью независим от View, поэтому может использовать разные View, имеющие общий интерфейс.

Благодаря этому Presenter может использоваться повторно и использоваться в общих модулях.

Presenter общается с представлением через интерфейс, который позволяет увеличить тестируемость, так как модель может быть заменена на специальный макет для модульных тестов.

В MVP представление жёстко отделяется от модели, при этом связь организуется через Presenter. В отличие от MVC, MVP допускает повторное использование бизнес-логики без непосредственного видоизменения модели. Это позволяет, например, перенести приложение на другую платформу просто заменив View, а поскольку интерфейс будет тем же, Presenter и Model можно использовать повторно.

С точки зрения диаграммы последовательностей, она практически не отличается от MVC с пассивным представлением, поскольку отличия заметны только в структуре.



MVP реализуется в таких платформах, как:

- Google Web Toolkit

- ASP.NET Web Forms Model-View-Presenter (MVP)

- Java Swing/AWT

- Silverlight

- и другие

**Вопрос 16. Архитектурный паттерн MVVM. Расскажите проблематику, которая привела к появлению данного архитектурного паттерна. Расскажите базовый подход. Покажите взаимодействие объектов в рамках реализации некоторых операций CRUD на диаграмме последовательностей. Покажите, в каких фреймворках он применяется.**

Основная цель всех архитектурных паттернов – разделение ответственности, повторное использование кода, тестируемость. Если мы хотим изменить что-то, мы не должны ходить по разным участкам кода. Без разделения ответственности ни повторное использование кода, ни тестируемость практически невозможно реализовать.

В паттернах MVC и MVP связь между представлением и моделью часто становится громоздкой. Контроллер или презентер выступают в роли посредника, и все методы взаимодействия представления и модели так или иначе связаны с посредническим участием промежуточного слоя.

Промежуточный слой помогает связывать логику данных и логику отображения, но требует поддержки, а также может вызывать проблемы рассинхронизации данных между представлением и моделью, поскольку он должен связывать независимые по своей сути сущности, которые в идеале должны работать синхронно.

Решение проблемы предлагается в паттерне **MVVM – Model-View-ViewModel**. Данный паттерн основан на механизме **связывания данных**.

**Связывание данных** заключается в автоматической синхронизации источника данных – свойств или коллекций в ViewModel, предоставляющих данные или команды для интерфейса и цели данных – элементов пользовательского интерфейса View.

Важно понимать, что связывание данных осуществляется не вручную написанием кода, а технологиями фреймворка. Именно это является ключевым преимуществом паттерна MVVM – избавление разработчика от написания шаблонного кода синхронизации данных.

Благодаря связыванию данных View может привязываться к свойствам ViewModel, чтобы выполнять свои действия в виде команд. При этом View не обязана знать о ViewModel, достаточно лишь чтобы ViewModel имела необходимый интерфейс.

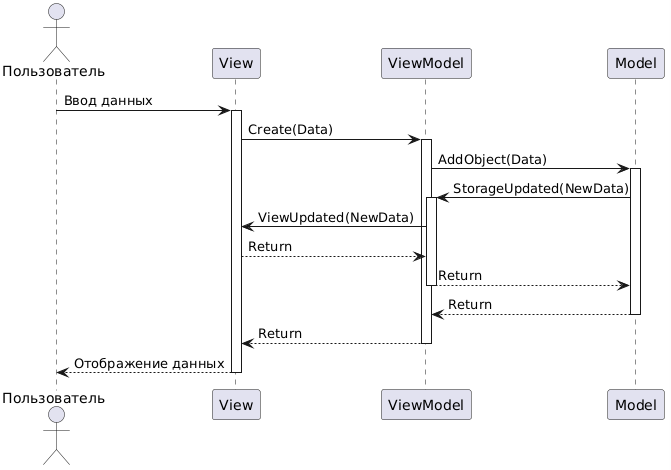
Вообще ViewModel имеет следующие обязанности:

- содержит логику пользовательского интерфейса

- хранит состояние пользовательского интерфейса

- предоставляет данные из модели для отображения экрана (при необходимости преобразуя их)

Связывание данных в MVVM основано на паттерне Наблюдатель: View подписывается на обновления данных в ViewModel, а ViewModel – на обновления Model. За счёт этого механизма обеспечивается синхронизация данных представления и модели. Таким образом, диаграмма последовательности для операции добавления CRUD в MVVM выглядит следующим образом:



Как упоминалось ранее, MVVM используется в системах, поддерживающих автоматическое связывание данных:

- в стеке технологий Microsoft за связывание данных отвечает язык разметки XAML. Его используют такие платформы, как WPF (Windows Presentation Foundation) и Silverlight.

- .NET-фреймворки: .NET Community MVVM Toolkit, ReactiveUI, Prism Library

- Java-Ajax-фреймворк ZK

- JavaScript-фреймворки: Angular, Ember.js, Knockout.js, Vue.js, Oracle JET

**Вопрос 17. Архитектурные паттерны. Расскажите о всех архитектурных паттернах, которые Вам известны, базовые подходы, в каких языках и фреймворках применяются. Раскройте паттерн, применённый в фреймворке Вашей задачи. Покажите взаимодействие объектов в рамках реализации некоторых операций CRUD на диаграмме последовательностей.**

Основная цель всех архитектурных паттернов – разделение ответственности, повторное использование кода, тестируемость. Если мы хотим изменить что-то, мы не должны ходить по разным участкам кода. Без разделения ответственности ни повторное использование кода, ни тестируемость практически невозможно реализовать.

Во всех MV\*-архитектурных паттернах логика данных и логика пользовательского интерфейса выделяются в, соответственно, Model и View. Взаимодействие между ними поручается промежуточным слоям, уникальным для каждого подхода.

К трём основным архитектурным паттернам относятся:

- MVC (Model-View-Controller) – связь между Model и View осуществляется Контроллером. Контроллер является Стратегией для View. Имеются два подхода: *контроллер-супервизор*, где модель имеет прямой доступ к представлению, и *пассивное представление*, где модель и представление могут взаимодействовать только через контроллер.

Реализуется в: JSF, Oracle App Framework, Play Framework [Java], ASP.NET MVC Framework, PureMVC [C#], Ruby on Rails [Ruby].

- MVP (Model-View-Presenter) исправляет зависимость представления и контроллера в MVC добавлением абстрактного интерфейса к View. Именно через него Presenter взаимодействует с View. При этом View становится пассивным и не знает о модели, полагаясь на Presenter для обмена данными (аналогично пассивному представлению MVC).

Реализуется в: Google Web Toolkit, ASP.NET Web Forms Model-View-Presenter (MVP), Java Swing/AWT, Silverlight

- MVVM (Model-View-ViewModel) обеспечивает синхронизацию данных представления и модели засчёт использования механизма свзяывания данных (databinding), предоставляемого фреймворком. Избавляет разработчика от написания шаблонного кода.

Реализуется в системах, поддерживающих автоматическое связывание данных: WPF, Silverlight [Microsoft], .NET Community MVVM Toolkit, ReactiveUI, PrismLibrary [.NET], ZK [Java], Angular, Ember.js, Knockout.js, Vue.js, Oracle JET [JavaScript].

В рамках задачи был применён фреймворк Ruby on Rails, основанный на архитектурном паттерне MVC.

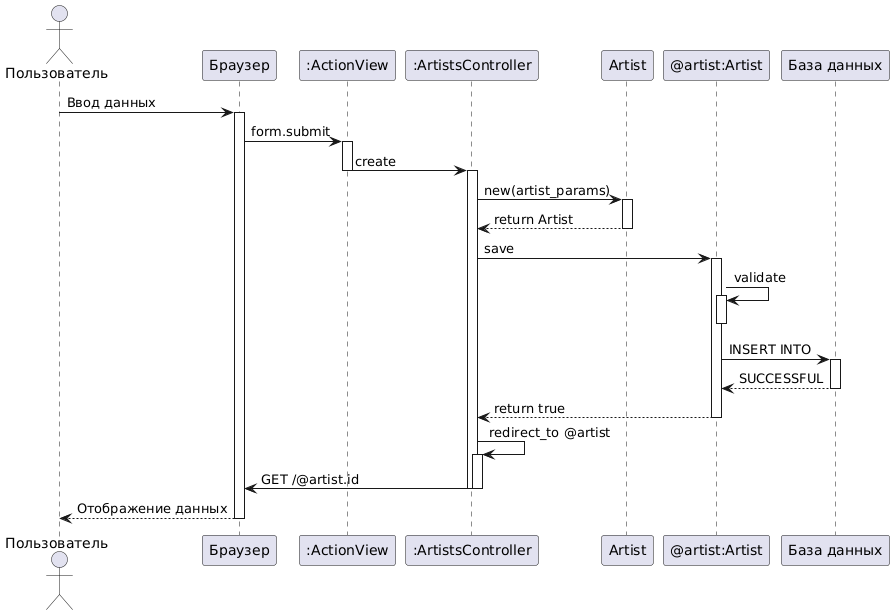
В Ruby on Rails основная логика HTTP-запросов содержится в суперклассе *ActionController*. От него наследуется глобальный контроллер *ApplicationController*, а от него – все остальные контроллеры для каждой из сущностей, находящихся в модели.

Контроллер содержит шаблонные методы, соответствующие запросам REST. После выполнения запроса контроллер может либо отобразить новый шаблон View, либо перенаправить на новый запрос с помощью метода *redirect\_to*. *redirect\_to* может как перенаправить на новый запрос HTTP, так и на другой метод контроллера.

Представление организуется встроенным классом *ActionView*, выполняющим рендер описанных разработчиком шаблонов html.erb – HTML с Embedded Ruby. Контроллер может передавать глобальные переменные в View, которые затем помещаются в шаблонные переменные Embedded Ruby и позволяют отображать необходимые данные на веб-странице.

Модели в Ruby on Rails наследуются от суперкласса *ActiveRecord*. Он содержит множество методов для связи класса модели с сущностью, помещаемой в базу данных, в том числе сериализаторы, валидаторы, конфиги БД, методы для связи различных сущностей, создания миграций, выполнения запросов к БД и другие.

Операция добавления нового исполнителя в базу данных на диаграмме последовательности выглядит следующим образом:



**Вопрос 18. Что такое рефакторинг? Когда его требуется применять? Приведите примеры методов рефакторинга и примеры антипаттернов, которые данные методы позволяют исправить.**

**Рефакторинг** – это контролируемый процесс улучшения кода без написания новой функциональности. Результат рефакторинга – это чистый код и простой дизайн.

В первую очередь, цель рефакторинга – превращение грязного кода в чистый.

Чистый код:

- проходит все тесты

- очевиден для других программистов

- не содержит дублирования

- содержит минимум классов и других движущихся частей

- легче и дешевле поддерживать

**Когда применять рефакторинг**:

1. Правило трёх:

1) Делая что-то в первый раз, вы просто это делаете.

2) Делая что-то аналогичное во второй раз, вы морщитесь от необходимости повторения, но всё-таки повторяете то же самое.

3) Делая что-то похожее в третий раз, вы начинаете рефакторинг.

2. При создании новой фичи. Рефакторинг помогает понять чужой код, а также написать новый. После рефакторинга добавление новой фичи происходит значительно более гладко и занимает меньше времени.

3. При исправлении багов.

4. Во время код-ревью. Ревью новой фичи может оказаться последним шансом почистить код перед тем, как он станет доступным публично.

При выполнении рефакторинга существует ряд общепринятых методов рефакторинга, решающих ряд различных задач по избавлению от антипаттернов. Эти методы группируются в следующие семейства:

1. Составление методов

*Пример*:

Извлечение метода – заключается в выделении участка кода в новый метод и вызове этого метода вместо старого кода.

Борется с: дублирование кода, длинный метод, завистливые функции, цепочка вызовов, класс данных, комментарии.

2. Перемещение функций между объектами

*Пример*:

Перемещение метода – создание нового метода в классе, который использует его больше других, и перенос туда старого метода. Код оригинального метода необходимо превратить в обращение к новому методу в другом классе либо убрать вовсе.

Борется с: стрельба дробью, завистливые функции, цепочка вызовов, неуместная близость, класс данных.

3. Организация данных

*Пример*:

Инкапсуляция поля – поле становится приватным, создаются методы доступа для него.

Борется с: класс данных.

4. Упрощение условных выражений

*Пример*:

Объединение условных операторов – несколько условных операторов, ведущих к одному результату, объединяются в одном условном операторе.

Борется с: дублирование кода.

5. Упрощение вызовов методов

*Пример*:

Передача всего объекта – вместо того, чтобы получать несколько значений из объекта, а затем передавать их в качестве параметров, передавать объект целиком.

Борется с: одержимость элементарными типами, длинный список параметров, длинный метод.

6. Решение задач обобщения

*Пример*:

Свёртывание иерархии – подклассы, мало отличающиеся от суперкласса, сливаются с суперклассом воедино.

Борется с: ленивый класс.

**Вопрос 19. Раскройте антипаттерны: Неуместная близость, завистливые функции, классы данных, одержимость элементарными типами, ленивый класс. РАскройте почему они считаются антипаттернами, привидите примеры кода, покажите методы рефакторинга, позволяющие их исправить. Укажите паттерны разработки, позволяющие не допускать таких ошибок.**

1. Неуместная близость возникает, когда один класс использует служебные поля и методы другого класса.

Это проблема, поскольку появляется слишком жёсткая связь между классами, затрудняющая их поддержку и повторное использование, нарушается структура программы и принцип инкапсуляции.

Пример:

class Order

attr\_accessor :items

def initialize

@items = []

end

def calculate\_total

total = 0

@items.each do |item|

total += item.price \* item.quantity

end

total

end

end

class Item

attr\_accessor :price, :quantity

def initialize(price, quantity)

@price = price

@quantity = quantity

end

end

Класс *Order* использует поля *price* и *quantity* класса *Item* для подсчёта общей стоимости, становясь зависимым от конкретной версии реализации. Если структура *Item* изменится и поменяется способ подсчёта стоимости, то придётся менять и класс *Order*.

Методы рефакторинга:

- перемещение метода/поля из одного класса в другой (только в случае, если оригинальный класс их не использует)

- извлечение зависимых частей в отдельный класс и сокрытие делегирования к этому классу

- если возникает между подклассом и родительским классом, стоит рассмотреть замену делегации наследованием

Паттерны, предотвращающие этот антипаттерн:

- Адаптер позволяет вынести взаимодействие классов в отдельный класс с удобным интерфейсом, преобразующий данные в необходимый формат.

- множество связей можно вынести в класс-Посредник.

- проблематичные методы можно вынести в Стратегии.

2. Завистливые функции – функции, обращающиеся к данным другого объекта чаще, чем к собственным данным.

Антипаттерн проблематичен в первую очередь из-за нарушения инкапсуляции – согласно этому принципу, данные и работающие с ними методы должны находиться в одном классе.

Пример:

class Customer

attr\_accessor :first\_name, :last\_name, :email

def initialize(first\_name, last\_name, email)

@first\_name = first\_name

@last\_name = last\_name

@email = email

end

end

class CustomerService

def send\_welcome\_email(customer)

email\_content = "Dear #{customer.first\_name} #{customer.last\_name}, welcome to our platform!"

EmailSender.send\_email(customer.email, email\_content)

end

end

class EmailSender

def self.send\_email(to, content)

puts "Sending email to #{to}: #{content}"

end

end

Метод *send\_welcome\_email* использует все поля класса *Customer* для своего выполнения.

Методы рефакторинга:

- перемещение метода, если его явно стоит переместить

- извлечение метода, если только часть метода обращается к данным другого объекта. Если при этом метод обращается к нескольким классам, можно разбить метод на несколько частей и поместить в эти классы.

Паттерны, предотвращающие этот антипаттерн:

- Стратегия позволяет хранить логику взаимодействия двух классов в соответствующем классе-стратегии, избавляя основной класс от лишних связей

3. Классы данных – это классы, которые содержат только поля и простейшие методы для доступа к ним (геттеры и сеттеры). Это просто контейнеры для данных, используемые другими классами. Сами они не содержат никакой дополнительной функциональности и не могут самостоятельно работать с данными, которыми владеют.

Нормально, если класс в начале своей жизни имеет лишь несколько полей, но основная сила объектов заключается именно в способности хранить типы поведения или операции над собственными данными.

Пример:

class Order

attr\_accessor :items

def initialize

@items = []

end

def calculate\_total

total = 0

@items.each do |item|

total += item.price \* item.quantity

end

total

end

end

class Item

attr\_accessor :price, :quantity

def initialize(price, quantity)

@price = price

@quantity = quantity

end

end

Использованный ранее пример для Неуместной близости также демонстрирует Класс данных: *Item*. Он не имеет в себе никаких методов, и логично было бы добавить в него вычисление стоимости.

Методы рефакторинга:

- все данные, хранящиеся в таких классах необходимо инкапсулировать

- методы доступа к полям этого класса переместить в него, если они встречаются где-то в других классах, а если не удаляется выделить их целиком, использовать выделение метода

- когда такой класс наполнится методами, стоит задуматься об удалении старых методов доступа к данным, которые дают слишком открытый доступ, и применить удаление сеттера и сокрытие метода

Паттерны, предотвращающие этот антипаттерн:

- Если класс данных имеет множество полей, можно разделить его создание на несколько частей паттерном Строитель.

- Если другие классы следят за изменениями в классе данных, можно организовать механизм подписки применением паттерна Наблюдатель.

4. Одержимость элементарными типами возникает в следующих случаях:

- использование элементарных типов вместо маленьких объектов для небольших задач (например, валюта, диапазоны, телефонные номера и т.п.)

- использование констант для кодирования какой-то информации (например, USER\_ADMIN\_ROLE = 1 для обозначения пользователя-администратора)

- использование строковых констант в качестве названий полей в массивах

Пример:

class Order

attr\_accessor :items

def initialize

@items = []

end

def add\_item(name, price, quantity)

@items << { name: name, price: price, quantity: quantity }

end

def total

@items.reduce(0) { |sum, item| sum + item[:price] \* item[:quantity] }

end

end

Предметы *@items* хранятся в виде хэша, а не специализированного объекта.

Методы рефакторинга:

- множество разнообразных полей примитивных типов логически сгруппировать и перенести в собственный класс. С этим поможет замена значения данных объектом.

- если значения примитивных полей используются в параметрах методов, использовать замену параметров объектом или передачу всего объекта.

- если среди переменных есть массивы, использовать замену массива объектом

- в случаях, когда в переменных закодированы сложные данные, использовать замену кодирования типа классом, подклассом или состоянием/стратегией

Паттерны, предотвращающие этот антипаттерн:

- правильно реализованный Фабричный метод может обеспечивать отсутствие некорректных переменных элементарных типов.

5. Ленивый класс – класс, который делает недостаточно много, чтобы уделять ему достаточно внимания.

Пример:

class Invoice

def initialize(amount)

@amount = amount

end

def amount

@amount

end

end

class PaymentProcessor

def initialize(invoice)

@invoice = invoice

end

def process\_payment

# Логика по обработке платежа

end

end

class PaymentStatusChecker

def initialize(invoice)

@invoice = invoice

end

def check\_status

# Логика проверки статуса платежа

end

end

Классы *PaymentProcessor* и *PaymentStatusChecker* являются Ленивыми классами, поскольку предназначены для очень узких задач, и их можно беспрепятственно встроить в класс *Invoice*.

Методы рефакторинга:

- встраивание класса в другой

- если является подклассом, можно воспользоваться свёртыванием иерархии

Паттерны, предотвращающие этот антипаттерн:

- Многие паттерны, выделяющие небольшие классы с малым функционалом, делают это намеренно для улучшения качества остального кода. К таким паттернам можно отнести: Легковес, Декоратор, Итератор, Состояние, Стратегия.

**Вопрос 20. Раскройте антипаттерны: дублирование кода, длинный метод, длинный класс, длинный список параметров, повторы условных операторов, альтернативные классы с разными интерфейсами. Раскройте почему они считаются антипаттернами, привидите примеры кода, покажите методы рефакторинга, позволяющие их исправить. Укажите паттерны разработки, позволяющие не допускать таких ошибок.**

1. Дублирование кода – два фрагмента кода выглядят почти одинаковыми.

Возникает, когда над проектом работает несколько человек или программист по разным причинам копирует уже готовый участок кода в другое место. Бывает и косвенное дублирование, когда участки кода отличаются внешне, но выполняют примерно одинаковую функцию.

Пример:

class Report

def generate\_daily\_report

data = fetch\_daily\_data

formatted\_data = format\_data(data)

save\_to\_file(formatted\_data, "daily\_report.txt")

end

def generate\_weekly\_report

data = fetch\_weekly\_data

formatted\_data = format\_data(data)

save\_to\_file(formatted\_data, "weekly\_report.txt")

end

private

def fetch\_daily\_data

# Получение данных по дням

end

def fetch\_weekly\_data

# Получение данных по неделям

end

def format\_data(data)

# Форматирование данных

end

def save\_to\_file(content, filename)

File.open(filename, "w") do |file|

file.write(content)

end

end

end

Методы *generate\_daily\_report* и *generate\_weekly\_report* имеют очень похожую логику, за исключением одного шага алгоритма.

Методы рефакторинга:

Дублирование кода выражается многими разными способами, поэтому подходов для его исправления существует много:

- извлечение метода, если в двух методах одного класса имеется одинаковый участок кода

- если одинаковый участок кода находится в двух подклассах на одном уровне, то необходимо применить извлечение методас подъёмом полей, использующихся в нём; подъём тела конструктора (если одинаковый код находится в конструкторе); создание шаблонного метода (если код похож, но не идентичен); или замещение алгоритма (если оба метода делают одно и то же, но с помощью разных алгоритмов)

- извлечение суперкласса или класса, если дублирующийся код находится в разных классах

- объединение условных операторов, если условные операторы выполняют один и тот же код, но с разными условиями

- объединение дублирующихся фрагментов в условных операторах, если проблема в ветках условного оператора

Паттерны, предотвращающие этот антипаттерн:

- Шаблон отлично помогает от дублирования кода в методах подклассов.

- Декоратор помогает от дублирования кода во множестве вариаций подклассов.

2. Длинный метод – метод содержит слишком большое число строк кода (более десяти).

Антипаттерн-раздувальщик. Затрудняет читабельность кода.

Пример:

class OrderProcessor

def process\_order(order)

puts "Processing order ##{order.id}"

# Шаг 1: Проверка наличия товаров

if order.items.empty?

puts "Order ##{order.id} has no items."

return

end

# Шаг 2: Проверка наличия товара на складе

order.items.each do |item|

if item.stock\_quantity < item.quantity

puts "Item #{item.name} is out of stock."

return

end

end

# Шаг 3: Расчет стоимости

total\_price = 0

order.items.each do |item|

total\_price += item.price \* item.quantity

end

puts "Total price: $#{total\_price}"

# Шаг 4: Сохранение заказа

puts "Saving order ##{order.id}..."

order.status = "Processed"

order.save

# Шаг 5: Уменьшение количества товара на складе

order.items.each do |item|

item.stock\_quantity -= item.quantity

item.save

end

# Шаг 6: Уведомление клиента

puts "Sending confirmation email to #{order.customer\_email}..."

send\_email(order.customer\_email, "Your order ##{order.id} has been processed.")

end

private

def send\_email(email, message)

# Логика отправки письма

end

end

Метод *process\_order* очень длинный. Он содержит много строк кода и выполняет слишком много различных функций, чем ухудшает читабельность и поддерживаемость.

Методы рефакторинга:

- чаще всего достаточно применить извлечение метода.

- если локальные переменные или параметры мешают извлечению метода, применить замену временной переменной вызовом метода, замену параметров объектом или передачу всего объекта.

- если предыдущие способы не помогли, можно попробовать выделить весь метод в отдельный объект

- при наличии условных операторов применить их декомпозицию

Паттерны, предотвращающие этот антипаттерн:

- Шаблон по своему принципу сокращает длину метода выделением шаблонного.

- Абстрактная фабрика помогает сократить длину конструктора, если применяются семейства объектов.

- Адаптер может помочь, если длинный метод связан с неудобным интерфейсом внешнего класса.

- Аналогично Фасад может сократить длину метода, если она связана с применением различных классов или методов внешней библиотеки или модуля.

- Цепочка обязанностей позволяет выделить различные шаги в последовательном алгоритме в отдельные классы, сократив при этом оригинал.

3. Длинный класс – класс, содержащий множество полей/методов/строк кода.

Классы закономерно расширяются с развитием программы, но слишком раздутые классы могут привести к ухудшению читабельности, производительности, поддерживаемости, а также к нарушению принципа единственной ответственности.

Пример:

class ECommercePlatform

def initialize

@users = []

@products = []

@orders = []

end

# Управление пользователями

def add\_user(name, email)

@users << { name: name, email: email }

end

def find\_user(email)

@users.find { |user| user[:email] == email }

end

# Управление товарами

def add\_product(name, price, stock)

@products << { name: name, price: price, stock: stock }

end

def find\_product(name)

@products.find { |product| product[:name] == name }

end

# Управление заказами

def create\_order(user\_email, product\_name, quantity)

user = find\_user(user\_email)

product = find\_product(product\_name)

if user && product && product[:stock] >= quantity

product[:stock] -= quantity

@orders << { user: user, product: product, quantity: quantity, total\_price: product[:price] \* quantity }

else

puts "Order could not be created"

end

end

def list\_orders

@orders.each do |order|

puts "User: #{order[:user][:name]}, Product: #{order[:product][:name]}, Quantity: #{order[:quantity]}, Total: $#{order[:total\_price]}"

end

end

# Отчеты

def generate\_sales\_report

puts "Sales Report:"

@orders.each do |order|

puts "Product: #{order[:product][:name]}, Quantity: #{order[:quantity]}, Total: $#{order[:total\_price]}"

end

end

def generate\_user\_report

puts "User Report:"

@users.each do |user|

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

end

end

end

В данном примере класс *EcommercePlatform* - наивно раздутый класс, очень явно нарушающий принцип единственной ответственности.

Методы рефакторинга:

Все методы исправления Длинного класса сводятся к разделению его на части. Это включает:

- извлечение класса, если часть поведения может быть выделена в отдельный компонент.

- извлечение подкласса, если часть поведения может иметь альтернативные реализации, либо используются в редких случаях.

- извлечение интерфейса, если нужно иметь список операций и поведений, которые клиент может использовать.

Паттерны, предотвращающие этот антипаттерн:

Большие классы предотвращают многие паттерны, извлекающие часть логики класса в отдельный класс:

- Фабричный метод извлекает логику создания объектов.

- Легковес разделяет внешнее и внутреннее состояние объектов.

- Цепочка обязанностей – разделяет многочисленные этапы одного алгоритма.

- Команда – отделяет запросы от их вызовов.

- Состояние и Стратегия извлекают меняющуюся от контекста логику.

4. Длинный список параметров – количество параметров метода больше трёх-четырёх.

Может появиться при попытке объединения нескольких вариантов алгоритма или уменьшения связанности между классами. В них трудно разбираться, они становятся противоречивыми и сложными в использовании.

Пример:

class Order

def initialize(customer\_name, customer\_email, shipping\_address, billing\_address, order\_items, payment\_method, discount\_code, gift\_wrap, special\_instructions)

@customer\_name = customer\_name

@customer\_email = customer\_email

@shipping\_address = shipping\_address

@billing\_address = billing\_address

@order\_items = order\_items

@payment\_method = payment\_method

@discount\_code = discount\_code

@gift\_wrap = gift\_wrap

@special\_instructions = special\_instructions

end

def display\_order\_details

...

end

end

Конструктор принимает огромное число параметров, часть которых вовсе может быть необязательной. В данном случае лучше всего поможет паттерн Строитель.

Методы рефакторинга:

- если передаваемые данные можно получить путём вызова метода другого объекта, применить замену параметра вызовом метода.

- если большинство параметров получаются из какого-то одного объекта, можно заменить параметры передачей всего объекта.

- несколько несвязанных элементов данных можно объединить в один объект-параметр, применив замену параметров объектом.

Паттерны, предотвращающие этот антипаттерн:

- Строитель позволяет избавиться от длинного списка параметров в конструкторе выделением отдельных методов.

- Стратегия и Состояние могут организовать альтернативное выполнение метода, которое по-другому вызывалось бы передачей параметров контекста.

5. Повторы условных операторов – несколько условных операторов ведут к одинаковому результату или действию.

Это узконаправленный антипаттерн, являющийся подвидом дублирования кода.

Пример:

class DiscountCalculator

def calculate\_discount(order)

if order.total\_price > 1000

order.apply\_discount(10)

elsif order.items\_count > 10

order.apply\_discount(10)

elsif order.customer.loyalty\_points > 500

order.apply\_discount(10)

end

end

end

Несколько условий приводят к одному результату. Логично было бы выделить их в отдельный метод, вроде *eligible\_for\_discount?*, проверяющий все эти условия.

Методы рефакторинга:

Антипаттерн не имеет вариаций, поэтому, имеет одно конкретное решение – объединение условных операторов. Условие дублирующихся операторов выносится в отдельный метод, отражающий суть проверяемого выражения, и остаётся один условный оператор, использующий именно его в качестве условия.

Паттерны, предотвращающие этот антипаттерн:

- Посредник может извлечь повторяющиеся условия при взаимодействии разных классов в отдельный объект.

6. Альтернативные классы с разными интерфейсами – два класса выполняют одинаковые функции, но имеют разные названия методов.

Антипаттерн является одним из случаев косвенного дублирования кода.

Пример:

class EmailNotifier

def send\_email(to, subject, body)

puts "Sending email to #{to}: #{subject} - #{body}"

end

end

class SMSNotifier

def send\_sms(phone\_number, message)

puts "Sending SMS to #{phone\_number}: #{message}"

end

end

Данные классы занимаются отправкой уведомлений – их логика идентична, но сигнатура методов и интерфейс отличаются.

Методы рефакторинга:

- переименование методов, имеющих одинаковый функционал.

- приведение сигнатуры и реализации методов к одинаковой при помощи перемещения и параметризации метода и добавления параметров.

- если только часть функционала классов идентична, можно выделить её в общий суперкласс.

- после проведения всех действий можно рассмотреть вариант удаления одного из классов.

Паттерны, предотвращающие этот антипаттерн:

- в некоторых случаях похожие, но не идентичные классы можно структурировать с помощью паттерна Декоратор (но только после выделения общего интерфейса).

**Вопрос 21. Раскройте антипаттерны: стрела дробью, расходящиеся модификации, посредник, теоретическая общность, параллельные иерархии наследования. Покажите, как они взаимосвязаны. РАскройте почему они считаются антипаттернами, привидите примеры кода, покажите методы рефакторинга, позволяющие их исправить. Укажите паттерны разработки, позволяющие не допускать таких ошибок, или приводящие к ним.**

1. Стрельба дробью – при выполнении любых модификаций приходится вносить множество мелких изменений в большое число классов.

**Стрельба дробью похожа на Расходящиеся модификации**, но является его противоположностью. Расходящиеся модификации имеют место, когда есть одитн класс, в котором производится много различных изменений, а Стрельба дробью – одно изменение, затрагивающее много классов.

Может появиться после фанатичного исправления Расходящихся модификаций. Усложняет модификацию кода.

Пример:

class Order

attr\_accessor :customer, :total\_price

def initialize(customer, total\_price)

@customer = customer

@total\_price = total\_price

end

end

class Customer

attr\_accessor :email

def initialize(email)

@email = email

end

end

class EmailService

def send\_email(email, message)

puts "Sending email to #{email}: #{message}"

end

end

class InventoryService

def update\_inventory(order)

puts "Updating inventory for order total: #{order.total\_price}"

end

end

class OrderProcessor

def process\_order(order)

email\_service = EmailService.new

email\_service.send\_email(order.customer.email, "Your order has been placed.")

inventory\_service = InventoryService.new

inventory\_service.update\_inventory(order)

end

end

Если придётся добавить отправку уведомлений по SMS, придётся изменять сразу несколько классов.

Методы рефакторинга:

- перемещением метода/поля вынести изменяющиеся свойства в один класс. Если перед выполнением нет подходящего класса, создать новый.

- если после вынесения в один класс в оригинальных классах мало что осталось, можно воспользоваться встраиванием класса.

Паттерны, предотвращающие этот антипаттерн:

- Шаблон позволяет вынести повторяющуюся логику методов в один метод, что избавит от необходимости изменения нескольких классов.

Паттерны, приводящие к этому антипаттерну:

- Легковес разделяет логику объекта на два состояния, поэтому при его общем изменении придётся менять оба класса.

- Абстрактная фабрика и Строитель могут привести к необходимости модифицировать сразу несколько классов.

2. Расходящиеся модификации – при внесении изменений в класс приходится изменять большое число различных методов.

**Расходящиеся модификации похожи на Стрельбу дробью**, но являются её противоположностью. Расходящиеся модификации имеют место, когда есть один класс, в котором производится много разных изменений, а Стрельба дробью – одно изменение, затрагивающее одновременно много классов.

Подобно длинному классу затрудняет модификацию кода, ухудшает читабельность.

Пример:

class ProductCatalog

def search\_products\_by\_category(category)

if category == "electronics"

puts "Searching for electronics"

elsif category == "clothing"

puts "Searching for clothing"

elsif category == "books"

puts "Searching for books"

end

end

def display\_product\_details(product)

if product[:category] == "electronics"

puts "Displaying details for electronic product: #{product[:name]}"

elsif product[:category] == "clothing"

puts "Displaying details for clothing product: #{product[:name]}"

elsif product[:category] == "books"

puts "Displaying details for book: #{product[:name]}"

end

end

def place\_order(product, quantity)

if product[:category] == "electronics"

puts "Placing order for #{quantity} electronics: #{product[:name]}"

elsif product[:category] == "clothing"

puts "Placing order for #{quantity} clothing items: #{product[:name]}"

elsif product[:category] == "books"

puts "Placing order for #{quantity} books: #{product[:name]}"

end

end

end

Примитивный пример, показывающий излишнюю зависимость нескольких методов от конкретных атрибутов.

Методы рефакторинга:

- разделить поведение класса, используя извлечение класса.

- если различные классы имеют одно и то же поведение, возможно, следует их объединить, используя наследование (извлечение суперкласса/подкласса).

Паттерны, предотвращающие этот антипаттерн:

- Паттерны, разделяющие логику класса: Цепочка обязанностей, Команда, Состояние и Стратегия.

Паттерны, приводящие к этому антипаттерну:

- Шаблон добавляет методы в класс, поэтому может привести к Расходящимся модификациям.

3. Посредник – всё, что делает класс – делегирует работу другому классу.

Может появиться при борьбе с цепочками вызовов, или когда вся полезная нагрузка класса постепенно перемещается в другие классы.

Подобно ленивому классу неоправданно занимает место в программе.

Пример:

class OrderManager

def initialize(order\_service)

@order\_service = order\_service

end

def create\_order(customer, items)

@order\_service.create\_order(customer, items)

end

def cancel\_order(order\_id)

@order\_service.cancel\_order(order\_id)

end

def get\_order\_details(order\_id)

@order\_service.get\_order\_details(order\_id)

end

end

class OrderService

def create\_order(customer, items)

puts "Order created for #{customer} with items: #{items.join(', ')}"

end

def cancel\_order(order\_id)

puts "Order #{order\_id} has been canceled."

end

def get\_order\_details(order\_id)

puts "Details for order #{order\_id}: ..."

end

end

В данном примитивном примере класс *OrderManager* является посредником для класса *OrderService* и не имеет собственного функционала.

Методы рефакторинга:

- Лечится единым способом – удалением посредника. Классы, использующие методы посредника, вместо этого должны вызывать методы целевого класса.

Паттерны, предотвращающие этот антипаттерн:

- Фасад по определению создаёт новый интерфейс, поэтому маловероятно, что он станет Посредником, и напротив предотвратит создание маленьких Посредников для отдельных классов.

Паттерны, приводящие к этому антипаттерну:

- Паттерны, разделяющие логику на несколько подклассов: Компоновщик, Декоратор, Цепочка обязанностей могут привести к данному антипаттерну, если в какой-то момент одна из их частей утратят свой функционал.

4. Теоретическая общность – класс, метод, поле или параметр не используются.

Иногда код создаётся про запас, и в конце концов так и не используется. В результате этот код становится труднее понимать и поддерживать.

Антипаттерн **Посредник** может быть результатом Теоретической общности, если изначально задумывалось поместить в него некоторый функционал для промежуточной обработки.

Пример:

class ReportGenerator

def generate\_report(data, format = :pdf)

if format == :pdf

puts "Generating PDF report with data: #{data}"

elsif format == :csv

puts "CSV format is not supported yet."

else

raise "Unknown format: #{format}"

end

end

end

На будущее добавлен аргумент *format* и его вариации, но по факту реализуется только формат pdf.

Методы рефакторинга:

- Все приёмы, удаляющие ненужный код: сворачивание иерархии, встраивание класса/метода, удаление параметров/полей.

Паттерны, предотвращающие этот антипаттерн:

- При использовании Адаптера или Фасада будет предоставлен весь необходимый интерфейс для взаимодействия классов, что снизит вероятность появления логики «про запас».

- Шаблон позволяет выделить всегда повторяющуюся логику, что предотвратит создание ненужной логики для подклассов.

Паттерны, приводящие к этому антипаттерну:

- При написании Адаптера или Фасада может возникнуть желание добавить часть интерфейса, которая в итоге не будет использоваться.

- При выделении необязательных параметров Строителем в отдельный метод может оказаться, что он не будет использоваться.

- Заместитель может быть размещён для реализации некоторой промежуточной логики, которая в итоге не понадобиться.

5. Параллельные иерархии наследования – всякий раз при создании подкласса какого-то класса приходится создавать ещё один подкласс для другого класса.

Затрудняет поддерживаемость кода.

Пример:

# Иерархия продуктов

class Product

end

class Electronics < Product

end

class Clothing < Product

end

class Food < Product

end

# Иерархия доставки

class Delivery

end

class ElectronicsDelivery < Delivery

def deliver(product)

...

end

end

class ClothingDelivery < Delivery

def deliver(product)

...

end

end

class FoodDelivery < Delivery

def deliver(product)

...

end

end

Всякий раз, когда добавляется новый класс-продукт, необходимо добавить новый класс для обработки его доставки.

Методы рефакторинга:

- можно попытаться устранить дублирование параллельных классов в два этапа. Во-первых, нужно заставить экземпляры одной иерархии ссылаться на экземпляры другой иерархии. Затем следует убрать иерархию в ссылающемся классе с помощью перемещения метода и перемещения поля.

Паттерны, предотвращающие этот антипаттерн:

- Компоновщик может верно структурировать иерархию, которую можно представить в виде дерева, что избавит от появления параллельных.

Паттерны, приводящие к этому антипаттерну:

- Фабричный метод чаще всего может привести к этому антипаттерну, так как для каждого класса продукта надо создать свой подкласс создателя.

**Вопрос 22. Раскройте принцип единой отвественности. Покажите примеры антипаттернов его нарушающих, паттернов, его реализующих. Покажите на работающих примерах.**

*У класса должен быть только один мотив для изменения*.

Согласно данному принципу, необходимо стремиться к тому, чтобы каждый класс отвечал только за одну часть функциональности программы, причём она должна быть полностью инкапсулирована в этот класс (скрыта в нём).

Предназначен для борьбы со сложностью. Если один класс делает слишком много, то он становится трудночитаемым и плохо поддерживаемым. Повышается риск сломать остальные части класса, отвечающие за другой функционал.

Реализующие паттерны:

- Цепочка обязанностей – каждая часть цепочки имеет конкретную цель.

- Итератор – создаваемый итератор предназначен для прохода по пользовательскому классу-коллекции.

Нарушающие антипаттерны:

- Длинный метод может содержать логику, не относящуюся к области ответственности класса.

- Большой класс иногда может нарушать данный принцип

- Расходящиеся модификации могут быть связаны с нарушением данного принципа

- Также стоит заметить, что этот принцип нарушает паттерн Одиночка.

Пример:

class Employee

attr\_reader :name

def initialize(name)

@name = name

end

def generate\_report

...

end

def save\_report\_to\_file

report = generate\_report

...

end

private

attr\_writer :name

end

Класс *Employee* должен содержать только методы и поля, необходимые для работы с сотрудниками. В данном случае принцип единой ответственности нарушается добавлением методов для генерации и записи в файл отчётов. Логику работы с отчётами стоит вынести в отдельный файл, и при необходимости можно добавить ссылку на него в классе сотрудника, либо ссылку на сотрудника в классе генератора отчётов.

class Employee

attr\_reader :name

def initialize(name)

@name = name

end

private

attr\_writer :name

end

class ReportGenerator

def initialize(employee)

@employee = employee

end

def generate\_report

...

end

def save\_report\_to\_file

report = generate\_report

...

end

private

attr\_accessor :employee

end

**Вопрос 23. Раскройте принцип открытости - закрытости. Покажите примеры антипаттернов его нарушающих, паттернов, его реализующих. Покажите на работающих примерах.**

*Расширяйте классы, но не изменяйте их первоначальный код*

Данный принцип говорит о том, что нужно стремиться к тому, чтобы классы были открыты для расширения, но закрыты для изменения. Главная идея – не ломать существующий код при внесении изменений в программу.

Класс можно назвать *открытым*, если он доступен для расширения. Например, есть возможность расширить набор его операций или добавить новые поля, создав собственный подкласс.

В то же время класс можно назвать *закрытым* (*законченным*), если он готов для использования другими классами. Это означает, что интерфейс класса уже окончательно определён и не будет изменяться в будущем.

Модификация класса, который уже был написан, одобрен, протестирован и включён в проект, нежелательна. Вместо этого лучше создать подкласс, расширяющий поведение родителя.

Тем не менее, не стоит следовать этому принципу буквально для каждого изменения. Если нужно исправить ошибку в основном классе, то можно просто это сделать, а не решать проблему в дочернем классе.

Реализующие паттерны:

- Фабричный метод

- Абстрактная фабрика

- Мост

- Цепочка обязанностей

- Команда

- Наблюдатель

- Стратегия

Нарушающие антипаттерны:

- Расходящиеся модификации и Стрельба дробью нарушают данный принцип большим количеством изменений существующего кода.

Пример на основе паттерна Стратегия:

class Order

def initialize(line\_items, shipping\_type)

@line\_items = line\_items

@shipping\_type = shipping\_type

end

def get\_total

...

end

def get\_total\_weight

...

end

def get\_shipping\_cost

if shipping\_type == "ground" do

unless get\_total > 100 do

return max(10, get\_total\_weight \* 1.5)

end

return

elsif shipping\_type == "air" do

max(20, get\_total\_weight \* 3)

end

end

private

attr\_accessor :line\_items, :shipping\_type

end

В данном примере вычисление стоимости доставки завязано на её типе, и происходит через условные операторы внутри класса. Из-за этого при добавлении нового типа доставки придётся трогать весь класс *Order*. Эту проблему решает паттерн Стратегия:

class Order

def initialize(line\_items, shipping\_strategy)

@line\_items = line\_items

@shipping = shipping\_strategy

end

def get\_total

...

end

def get\_total\_weight

...

end

def get\_shipping\_cost

@shipping.get\_cost

end

private

attr\_accessor :line\_items, :shipping

end

class Shipping\_strategy

def get\_cost

raise NotImplementedError

end

end

class Ground\_strategy < Shipping\_strategy

def get\_cost

...

end

end

class Air\_strategy < Shipping\_strategy

def get\_cost

...

end

end

Благодаря вынесению методов подсчёта в стратегии, можно легко добавлять новые способы доставки, не меняя при этом ничего в классу *Order*.

Вопрос 24. Раскройте принцип подстановки Барбары Лисков. Покажите примеры антипаттернов его нарушающих, паттернов, его реализующих. Покажите на работающих примерах.

*Подклассы должны дополнять, а не замещать поведение базового класса*

Реализующие паттерны:

Нарушающие антипаттерны:

Вопрос 25. Раскройте принцип подстановки разделения интерфейсов. Покажите примеры антипаттернов его нарушающих, паттернов, его реализующих. Покажите на работающих примерах.

*Клиенты не должны зависеть от методов, которые они не используют*

Реализующие паттерны:

Нарушающие антипаттерны:

Вопрос 26. Раскройте принцип инверсии зависимостей. Покажите примеры антипаттернов его нарушающих, паттернов, его реализующих. Покажите на работающих примерах.

*Классы верхних уровней не должны зависеть от классов нижних уровней. Оба должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.*

Реализующие паттерны:

Нарушающие антипаттерны:

Вопрос 27. Принципы SOLID. Зачем нужны, о чём они, как взаимосвязаны с паттернами и антипаттернами? Покажите на примерах. Как реализуются в Вашей задаче и в выбранном Вами фреймворке.