1. Назначение структурных паттернов?

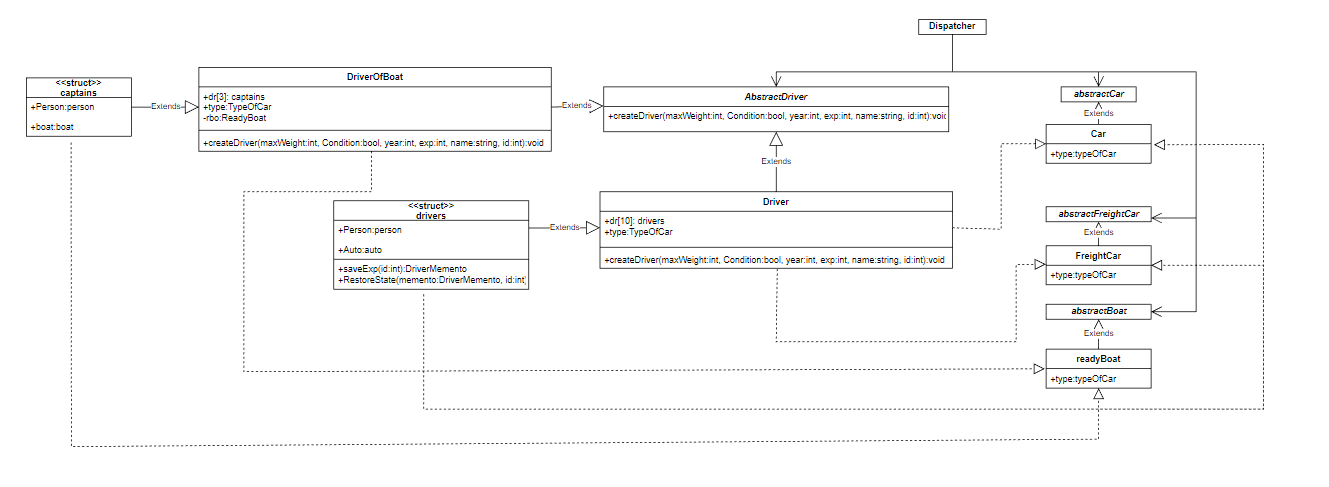
Структурные шаблоны — шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры.

Структурные шаблоны уровня класса используют наследование для составления композиций из интерфейсов и реализаций. Простой пример — использование множественного наследования для объединения нескольких классов в один. В результате получается класс, обладающий свойствами всех своих родителей. Особенно полезен этот шаблон, когда нужно организовать совместную работу нескольких независимо разработанных библиотек.

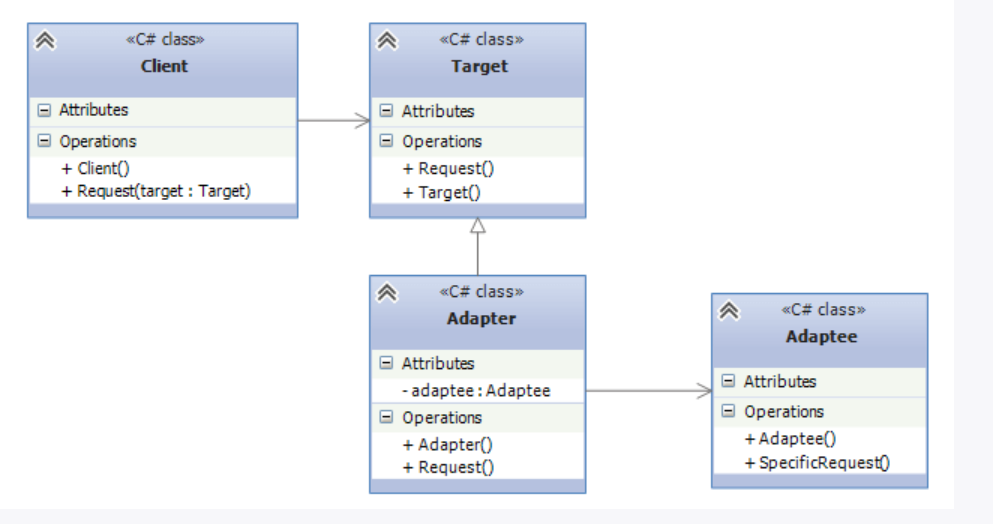
Шаблоны:Адаптер,Мост,Компоновщик,Декоратор,Фасад,Приспособленец,Заместитель.

2. Нарисуете диаграмму классов и поясните принцип работы паттерна Adapter. В каких случаях надо его применять.

Паттерн Адаптер (Adapter) предназначен для преобразования интерфейса одного класса в интерфейс другого. Благодаря реализации данного паттерна мы можем использовать вместе классы с несовместимыми интерфейсами.

\*Из лабораторной работы:

В общем виде:



3. В чем разница между адаптером класса и адаптером объекта?

Основное отличие:

* **Адаптер класса** использует *наследование* и может переносить только **класс**. Он не может обернуть интерфейс, поскольку по определению он должен быть производным от некоторого базового класса.
* **Объектный адаптер** использует *композицию* и может переносить классы или интерфейсы, или и то, и другое. Он может это сделать, поскольку содержит в качестве частного инкапсулированного члена **экземпляр объекта** класса или интерфейса, который он обертывает.
* у адаптеров объекта гораздо больше возможностей)

4. Назначение и принцип организации паттерна Декоратор.

Декоратор (Decorator) представляет структурный шаблон проектирования, который позволяет динамически подключать к объекту дополнительную функциональность.

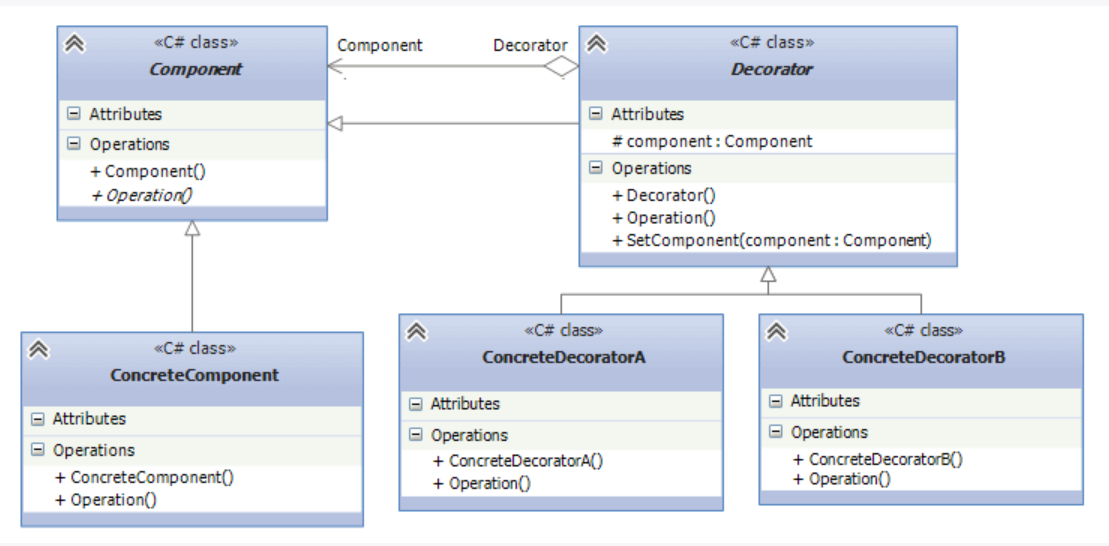
Для определения нового функционала в классах нередко используется наследование. Декораторы же предоставляет наследованию более гибкую альтернативу, поскольку позволяют динамически в процессе выполнения определять новые возможности у объектов.

**Когда следует использовать декораторы?**

Когда надо динамически добавлять к объекту новые функциональные возможности. При этом данные возможности могут быть сняты с объекта

Когда применение наследования неприемлемо. Например, если нам надо определить множество различных функциональностей и для каждой функциональности наследовать отдельный класс, то структура классов может очень сильно разрастись. Еще больше она может разрастись, если нам необходимо создать классы, реализующие все возможные сочетания добавляемых функциональностей.

Схематически шаблон "Декоратор" можно выразить следующим образом:



5. Нарисуйте диаграмму классов для паттерна Proxy. Поясните его назначение. Какие разновидности proxy вы знаете?

## Назначение:

Изменение требований и эволюция системы могут вызвать необходимость внесения серьезных архитектурных изменений. Если на ранних этапах некая операция выполнялась на стороне клиента или же приложение состояло из одного процесса, то со временем исполнение операции может быть перенесено на сервер, а приложение разбито на несколько процессов. В результате возникает задача взаимодействия с удаленным процессом, реализация которой должна быть максимально похожей на локальное взаимодействие. Именно для таких целей предназначен паттерн «Заместитель».

**Proxy (заместитель)**— ​является суррогатом другого объекта и контролирует доступ к нему.​

## Когда использовать Proxy?

«Заместитель» является одним из немногих паттернов проектирования, который с течением времени претерпел довольно серьезные изменения. В классическом труде «банды четырех» описаны три основных сценария использования паттерна «Заместитель».

* **Удаленный заместитель (remote proxies)**— ​отвечает за кодирование запроса и его аргументов для работы с компонентом в другом адресном пространстве.
* **Виртуальный заместитель (virtual proxies)**—​ может кэшировать дополнительную информацию о реальном компоненте, чтобы отложить его создание.
* **Защищающий заместитель (protection proxies)**— проверяет, имеет ли вызывающий объект необходимые для выполнения запроса права.​

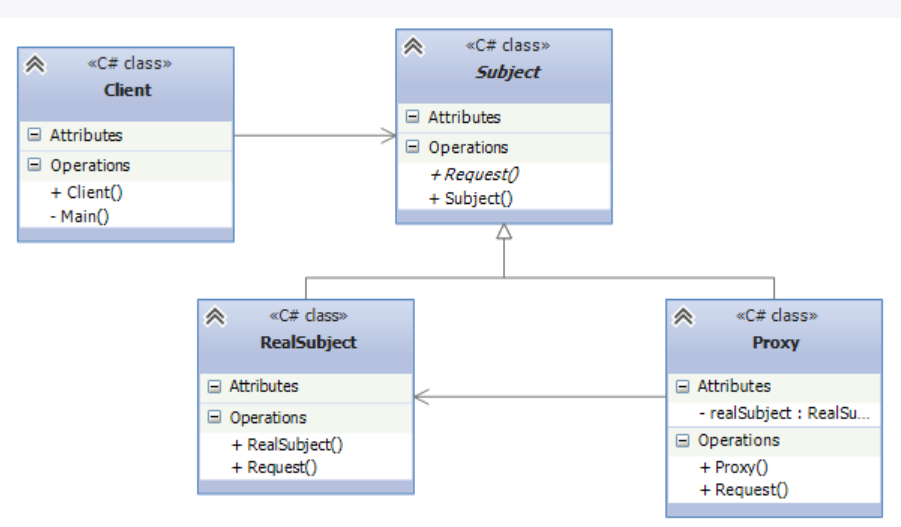
Другими словами:

Когда надо осуществлять взаимодействие по сети, а объект-проси должен имитировать поведения объекта в другом адресном пространстве. Использование прокси позволяет снизить накладные издержки при передачи данных через сеть. Подобная ситуация еще называется **удалённый заместитель (remote proxies)**

Когда нужно управлять доступом к ресурсу, создание которого требует больших затрат. Реальный объект создается только тогда, когда он действительно может понадобится, а до этого все запросы к нему обрабатывает прокси-объект. Подобная ситуация еще называется **виртуальный заместитель (virtual proxies)**

Когда необходимо разграничить доступ к вызываемому объекту в зависимости от прав вызывающего объекта. Подобная ситуация еще называется **защищающий заместитель (protection proxies)**

Когда нужно вести подсчет ссылок на объект или обеспечить потокобезопасную работу с реальным объектом. Подобная ситуация называется **"умные ссылки" (smart reference)**



6. В чем разница между паттернами Decorator и Adapter?

Adapter – используем два разных интерфейса, но в итоге работаем с этими двумя классами, как с одним.

Decorator – добавляет дополнительные характеристики при необходимости динамически.

Совсем разные паттерны)

Или же **Адаптор**- частный случай декоратора. Отличие скорее в том, что **декоратор**может быть масштабнее (содержать в себе много сущностей, в то время как **адаптор**, как правило, реализует интерфейс для определенного типа).

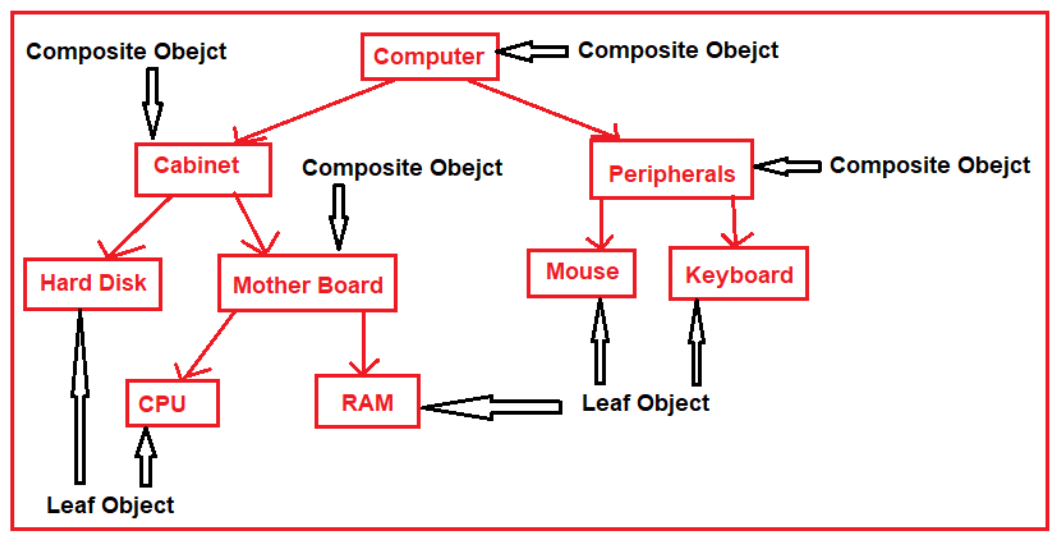
7. В чем суть паттерна Composite? Поясните в каких случаях его надо применять?

Паттерн Компоновщик (Composite) объединяет группы объектов в древовидную структуру по принципу "часть-целое и позволяет клиенту одинаково работать как с отдельными объектами, так и с группой объектов.

Образно реализацию паттерна можно представить в виде меню, которое имеет различные пункты. Эти пункты могут содержать подменю, в которых, в свою очередь, также имеются пункты. То есть пункт меню служит с одной стороны частью меню, а с другой стороны еще одним меню. В итоге мы однообразно можем работать как с пунктом меню, так и со всем меню в целом.

### Когда использовать компоновщик?

* Когда объекты должны быть реализованы в виде иерархической древовидной структуры
* Когда клиенты единообразно должны управлять как целыми объектами, так и их составными частями. То есть целое и его части должны реализовать один и тот же интерфейс



8. Расскажите о паттерне Façade?

## Назначение:

предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

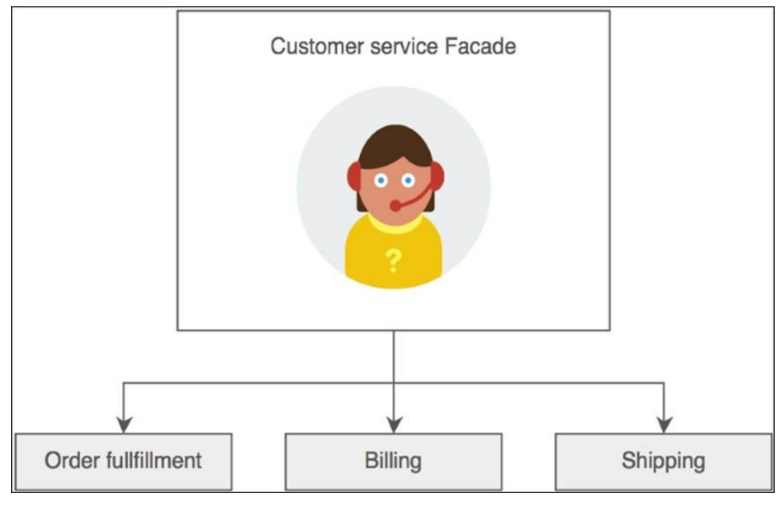
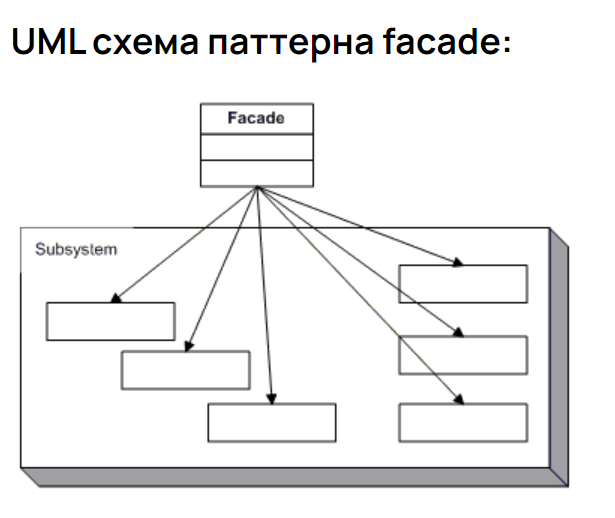
Шаблон Фасад объединяет группу объектов в рамках одного специализированного интерфейса и переадресует вызовы его методов к этим объектам.

## Когда использовать фасад?

* Когда имеется сложная система, и необходимо упростить с ней работу. Фасад позволит определить одну точку взаимодействия между клиентом и системой.
* Когда надо уменьшить количество зависимостей между клиентом и сложной системой. Фасадные объекты позволяют отделить, изолировать компоненты системы от клиента и развивать и работать с ними независимо.
* Когда нужно определить подсистемы компонентов в сложной системе. Создание фасадов для компонентов каждой отдельной подсистемы позволит упростить взаимодействие между ними и повысить их независимость друг от друга.

## Схема примера использования паттерна "Facade" из реальной жизни:

Фасад определяет унифицированный интерфейс более высокого уровня для подсистемы, которая упрощает ее использование. Потребители сталкиваются с фасадом при заказе из каталога. Потребитель звонит на один номер и разговаривает с представителем службы поддержки клиентов. Представитель службы поддержки клиентов выступает в качестве фасада, предоставляя интерфейс отделу исполнения заказов, отделу биллинга и отделу доставки.

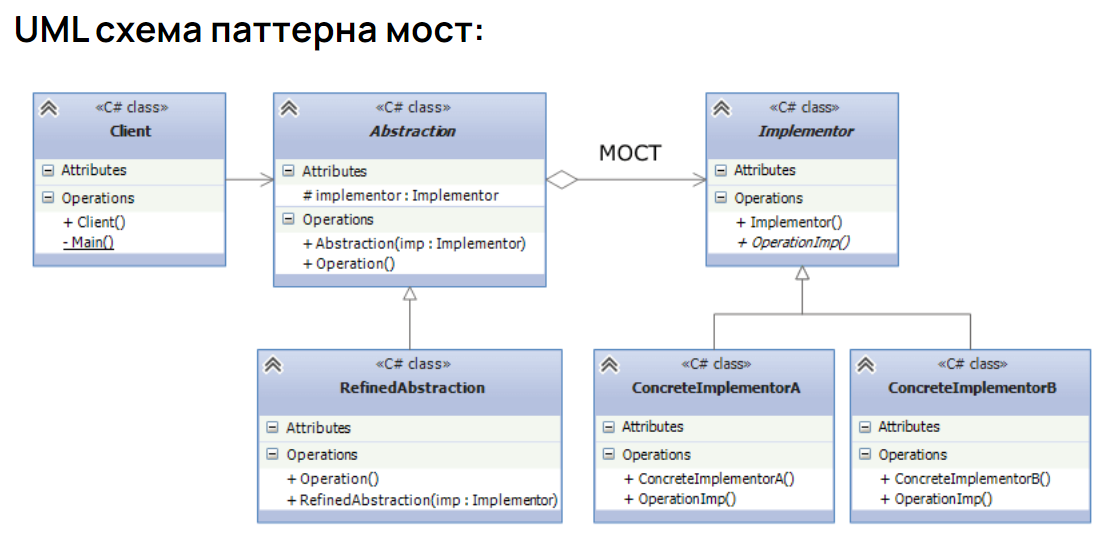
9. В каких случаях надо применять Bridge? Поясните на диаграмме классов как он применятся

**Мост** — это структурный паттерн, который разделяет бизнес-логику или большой класс на несколько отдельных иерархий, которые потом можно развивать отдельно друг от друга.

Одна из этих иерархий (абстракция) получит ссылку на объекты другой иерархии (реализация) и будет делегировать им основную работу. Благодаря тому, что все реализации будут следовать общему интерфейсу, их можно будет взаимозаменять внутри абстракции.

**Применимость:** Паттерн Мост особенно полезен когда вам приходится делать кросс-платформенные приложения, поддерживать несколько типов баз данных или работать с разными поставщиками похожего API (например, cloud-сервисы, социальные сети и т. д.)

**Признаки применения паттерна:** Если в программе чётко выделены классы «управления» и несколько видов классов «платформ», причём управляющие объекты делегируют выполнение платформам, то можно сказать, что у вас используется Мост.



10. Назначение паттернов поведения?

Поведенческие шаблоны (англ. behavioral patterns) — шаблоны проектирования, определяющие алгоритмы и способы реализации взаимодействия различных объектов и классов.

В поведенческих шаблонах уровня класса используется наследование, чтобы определить поведение для различных классов. В поведенческих шаблонах уровня объекта используется композиция. Некоторые из них описывают, как с помощью кооперации несколько равноправных объектов работают над заданием, которое они не могут выполнить по отдельности. Здесь важно то, как объекты получают информацию о существовании друг друга. Объекты-коллеги могут хранить ссылки друг на друга, но это усиливает степень связанности системы. При высокой связанности каждому объекту пришлось бы иметь информацию обо всех остальных.

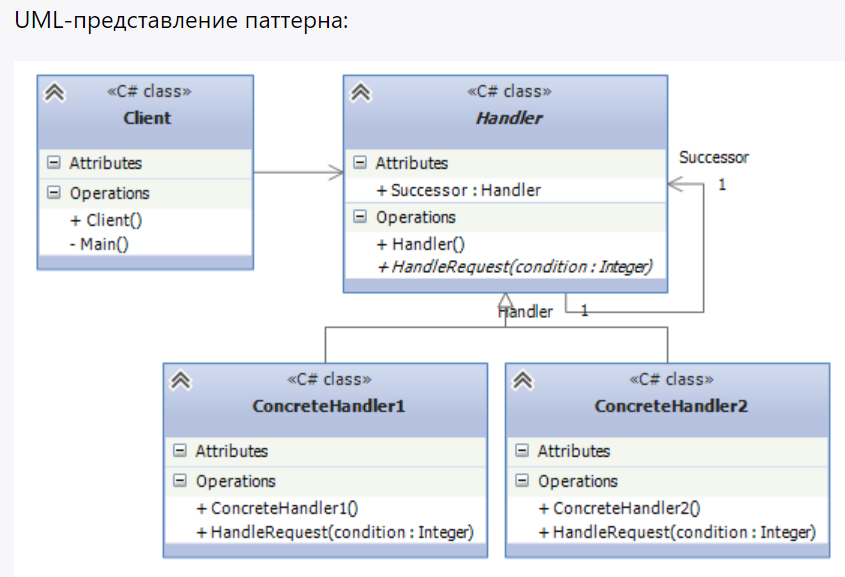
11. Нарисуете диаграмму классов и поясните принцип работы паттерна Chain of responsibility. В каких случаях надо его применять?

## Назначение:

Позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Связывает объекты-получатели в цепочку и передает запрос вдоль этой цепочки, пока его не обработают. «Цепочка обязанностей» является довольно распространенным паттерном в .NET Framework, хотя не все знают, что часто пользуются им. Цепочка обязанностей — это любое событие, аргументы которого позволяют уведомить инициатора, что событие обработано с помощью метода Handle() или путем установки свойства Handled в True.

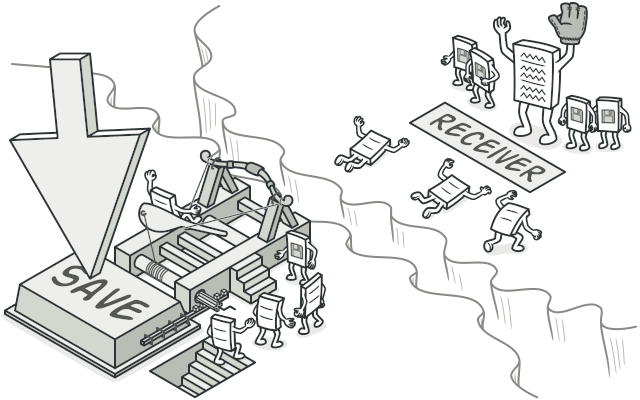
## Когда использовать Паттерн Chain of responsibility:

1. Когда имеется более одного объекта, который может обработать определенный запрос;
2. Когда надо передать запрос на выполнение одному из нескольких объект, точно не определяя, какому именно объекту;
3. Когда набор объектов задается динамически.



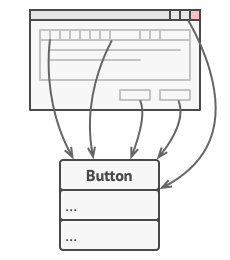
12. Назначение и принцип организации паттерна Command. Поясните как он связан с конечными автоматами

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



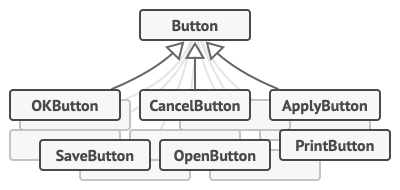
## *Проблема*

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



*Все кнопки приложения унаследованы от одного класса.*

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



*Множество подклассов кнопок.*

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



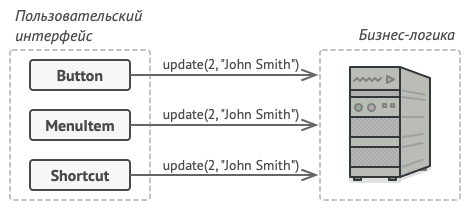
*Несколько классов дублируют одну и ту же функциональность.*

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

## Решение

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

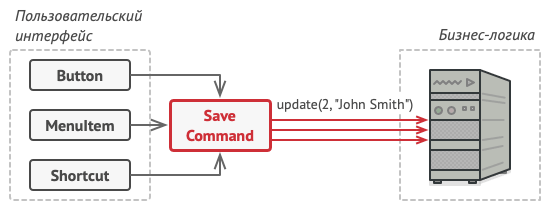
В реальности это выглядит так: один из объектов интерфейса напрямую вызывает метод одного из объектов бизнес-логики, передавая в него какие-то параметры.



*Прямой доступ из UI в бизнес-логику.*

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

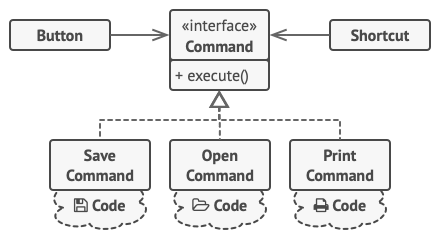
К объекту интерфейса можно будет привязать объект команды, который знает, кому и в каком виде следует отправлять запросы. Когда объект интерфейса будет готов передать запрос, он вызовет метод команды, а та — позаботится обо всём остальном.



*Доступ из UI в бизнес-логику через команду.*

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

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



*Классы UI делегируют работу командам.*

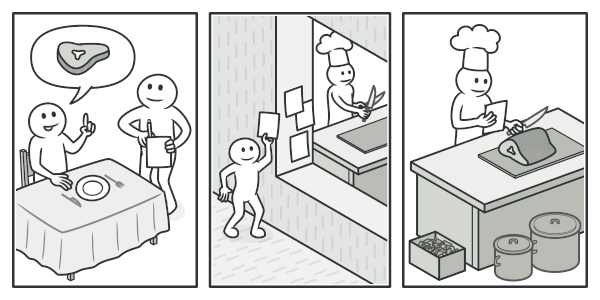
После применения Команды в нашем примере с текстовым редактором вам больше не потребуется создавать уйму подклассов кнопок под разные действия. Будет достаточно единственного класса с полем для хранения объекта команды.

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

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

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

## Аналогия из жизни

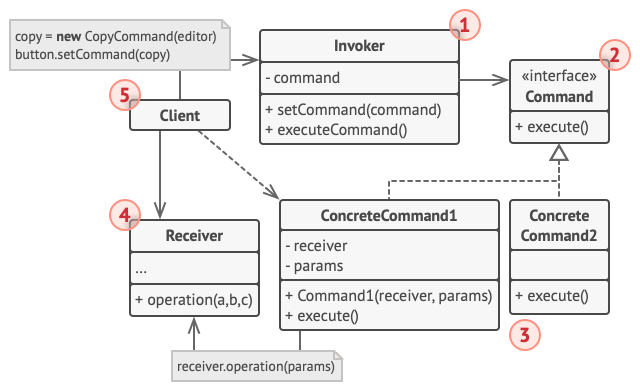


*Пример заказа в ресторане.*

Вы заходите в ресторан и садитесь у окна. К вам подходит вежливый официант и принимает заказ, записывая все пожелания в блокнот. Откланявшись, он уходит на кухню, где вырывает лист из блокнота и клеит на стену. Далее лист оказывается в руках повара, который читает содержание заказа и готовит заказанные блюда.

В этом примере вы являетесь отправителем, официант с блокнотом — командой, а повар — получателем. Как и в паттерне, вы не соприкасаетесь напрямую с поваром. Вместо этого вы отправляете заказ с официантом, который самостоятельно «настраивает» повара на работу. С другой стороны, повар не знает, кто конкретно послал ему заказ. Но это ему безразлично, так как вся необходимая информация есть в листе заказа.

## Структура



13. Как реализовать паттерн Observer?

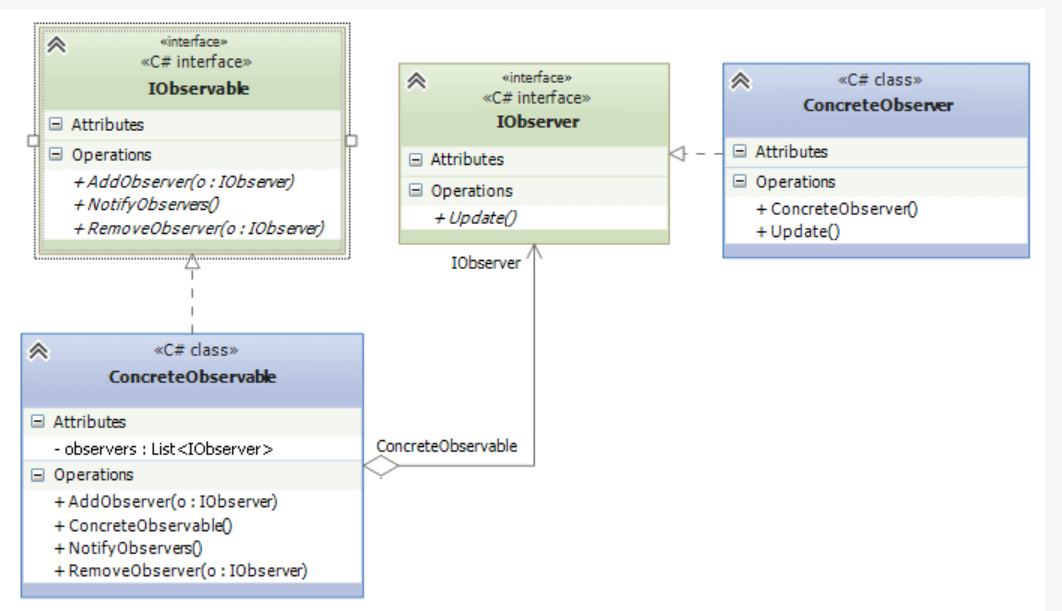
Паттерн "Наблюдатель" (Observer) представляет поведенческий шаблон проектирования, который использует отношение "один ко многим". В этом отношении есть один наблюдаемый объект и множество наблюдателей. И при изменении наблюдаемого объекта автоматически происходит оповещение всех наблюдателей.

Данный паттерн еще называют Publisher-Subscriber (издатель-подписчик), поскольку отношения издателя и подписчиков характеризуют действие данного паттерна: подписчики подписываются email-рассылку определенного сайта. Сайт-издатель с помощью email-рассылки уведомляет всех подписчиков о изменениях. А подписчики получают изменения и производят определенные действия: могут зайти на сайт, могут проигнорировать уведомления и т.д.

### Когда использовать паттерн Наблюдатель?

* Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях
* Когда общая схема взаимодействия объектов предполагает две стороны: одна рассылает сообщения и является главным, другая получает сообщения и реагирует на них. Отделение логики обеих сторон позволяет их рассматривать независимо и использовать отдельно друга от друга.
* Когда существует один объект, рассылающий сообщения, и множество подписчиков, которые получают сообщения. При этом точное число подписчиков заранее неизвестно и процессе работы программы может изменяться.

С помощью диаграмм UML данный шаблон можно выразить следующим образом:

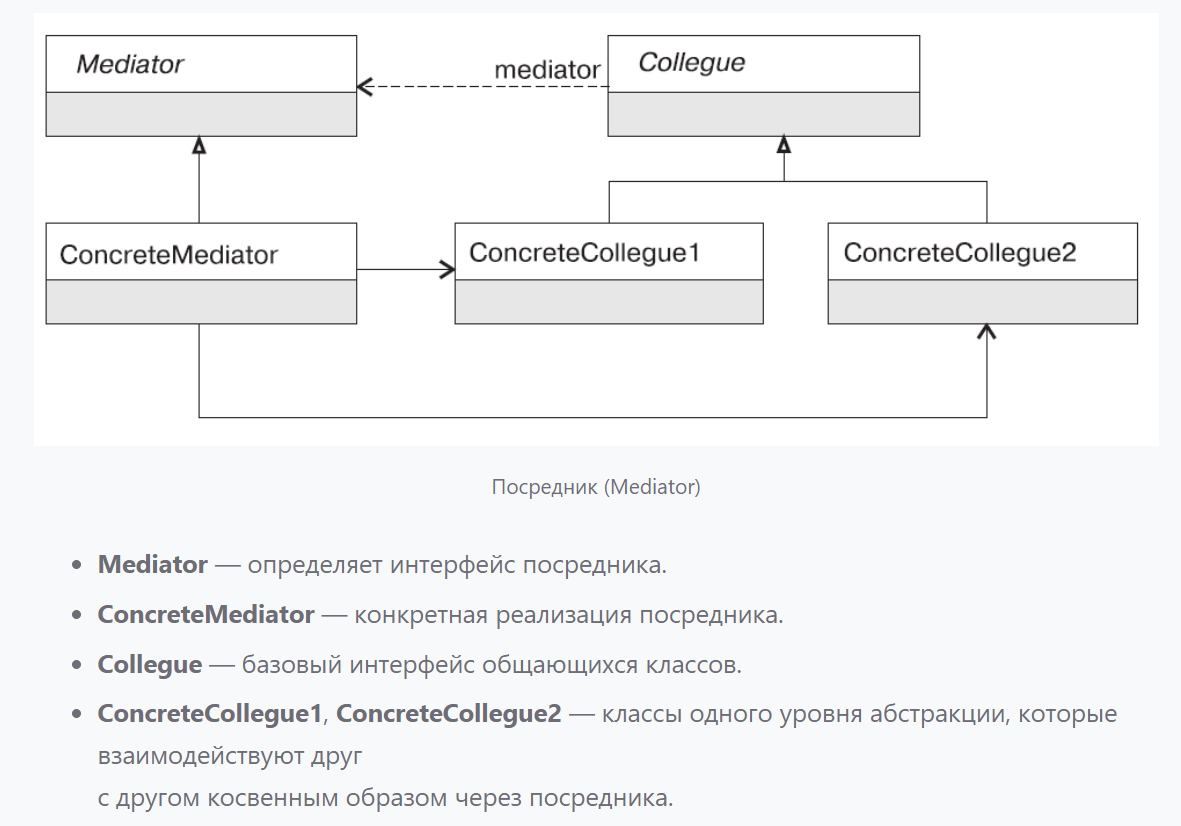


14. Нарисуйте диаграмму классов для паттерна Mediator. Поясните его назначение.

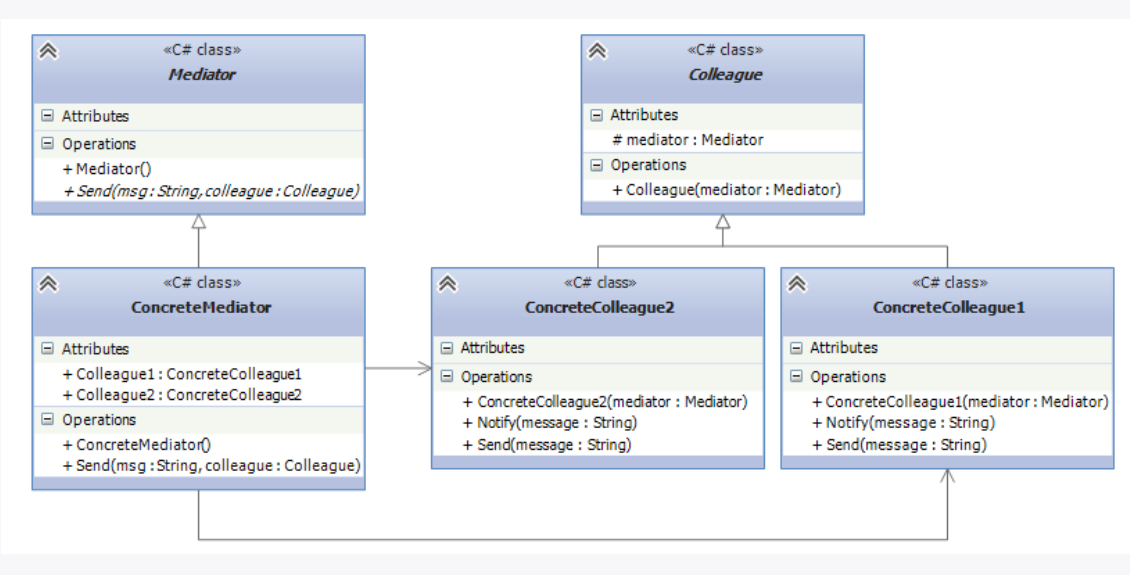
**Посредник (Mediator)** — это поведенческий паттерн, который позволяет организовать работу множества слабо связанных объектов без непосредственного общения между ними. То есть, Посредник выступает промежуточным звеном между объектами, принимая и перенаправляя сообщения.

## Логика работы Посредника

Рассмотрим основную логику работы паттерна Посредник. Давайте смоделируем процесс работы диспетчера полетов в аэропорту. Пилоты взлетающих или идущих на посадку самолетов в районе аэропорта общаются с диспетчером вместо непосредственного общения друг с другом. Диспетчер определяет, кто и в каком порядке будет садиться или взлетать. Данный механизм взаимодействия очень точно описывает логику работы паттерна Посредник.



или



15. В чем разница между паттернами Mediator и Facade?

**Фасад** только раскрывает существующую функциональность с другой точки зрения.

**Посредник** "добавляет" функциональность, поскольку он объединяет различные существующие функции для создания новой.

Фасад – просто связующее звено (интерфейс) для всех подклассов, в то время как посредник берет промежуточный результат, сохраняет, может поменять, а только затем передаёт его классу.

16. В чем суть паттерна Memento? Поясните на примере.

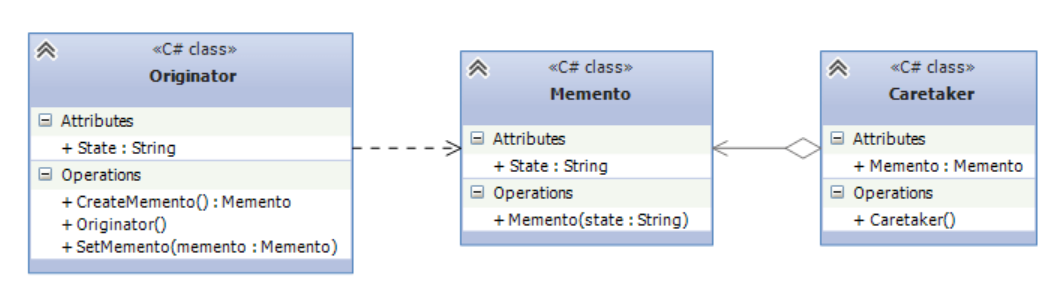
**Паттерн Хранитель (Memento) —** позволяет выносить внутреннее состояние объекта за его пределы для последующего возможного восстановления объекта без нарушения принципа инкапсуляции.

* Не нарушая инкапсуляции, паттерн Memento получает и сохраняет за пределами объекта его внутреннее состояние так, чтобы позже можно было восстановить объект в таком же состоянии.
* Является средством для инкапсуляции "контрольных точек" программы.
* Паттерн Memento придает операциям "Отмена" (undo) или "Откат" (rollback) статус "полноценного объекта".

## Когда использовать Memento?

* Когда нужно сохранить состояние объекта для возможного последующего восстановления;
* Когда сохранение состояния должно проходить без нарушения принципа инкапсуляции;

## UML схема паттерна Хранитель​:



\*В лабораторной работе я сохранял опыт работы водителя на конкретный период времени, затем он изменялся с течением времени, но в конечном итоге была возможность восстановить изначальное состояние данного поля.

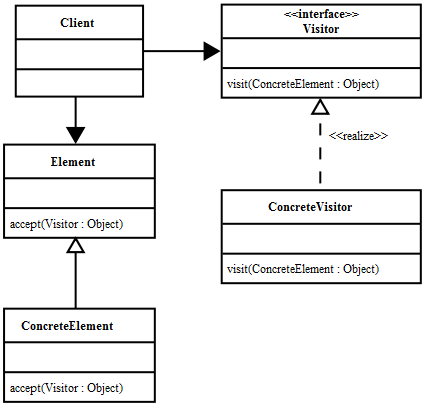
17. Расскажите о паттерне Visitor?

**Назначение:** описывает операцию, выполняемую с каждым объектом из некоторой иерархии классов. Паттерн «Посетитель» позволяет определить новую операцию, не изменяя классов этих объектов.

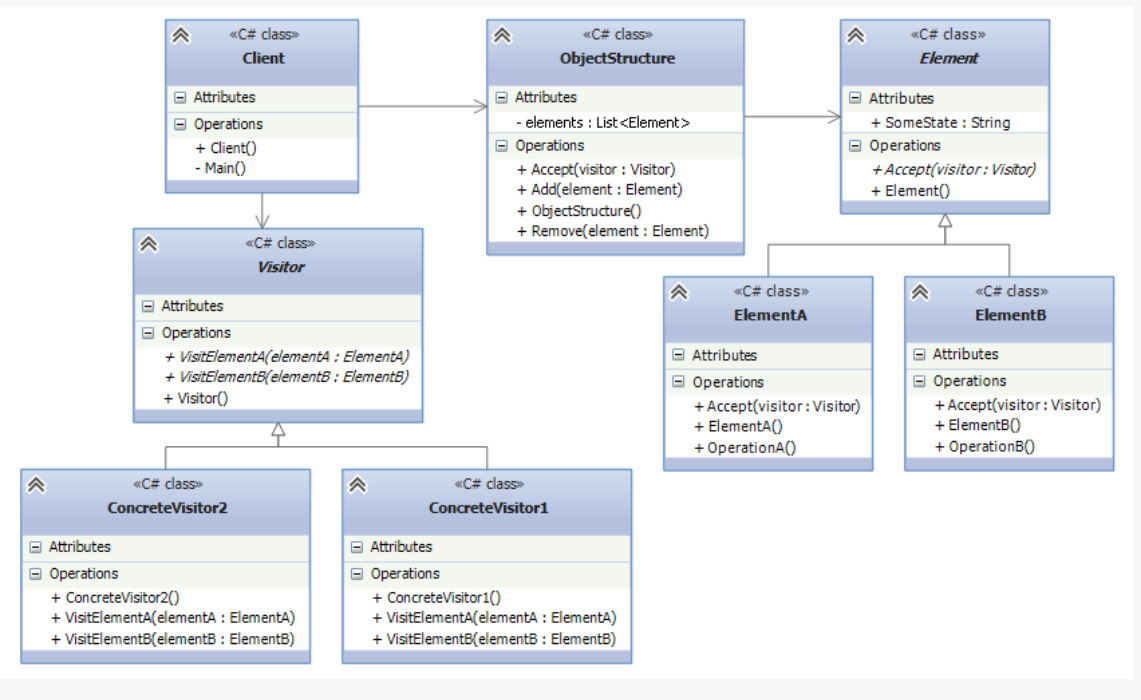
## Когда использовать паттерн Посетитель

* Паттерн Посетитель определяет операцию, выполняемую на каждом элементе из некоторой структуры. Позволяет, не изменяя классы этих объектов, добавлять в них новые операции.
* Является классической техникой для восстановления потерянной информации о типе.
* Паттерн Посетитель позволяет выполнить нужные действия в зависимости от типов двух объектов.
* Предоставляет механизм двойной диспетчеризации.

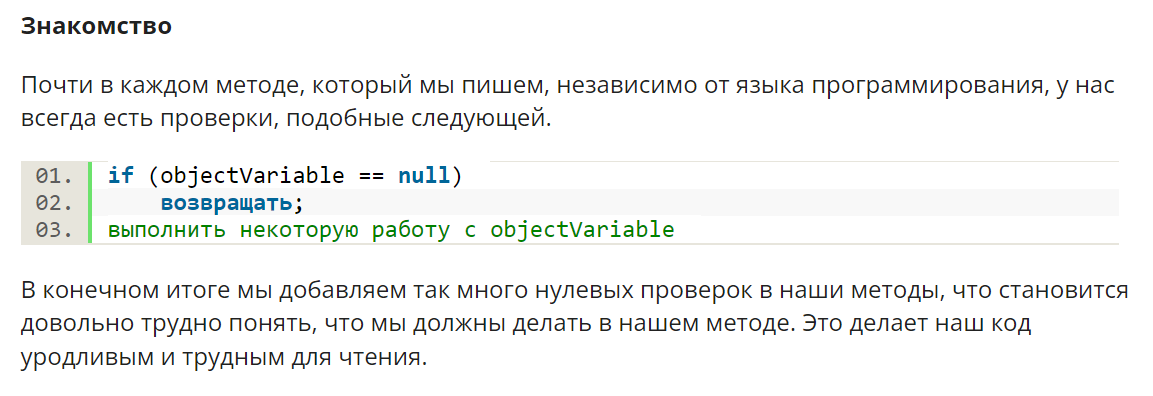
## UML схема паттерна Visitor



При использовании паттерна Посетитель определяются две иерархии классов: одна для элементов, для которых надо определить новую операцию, и вторая иерархия для посетителей, описывающих данную операцию.



18. В каких случаях надо применять Null object?



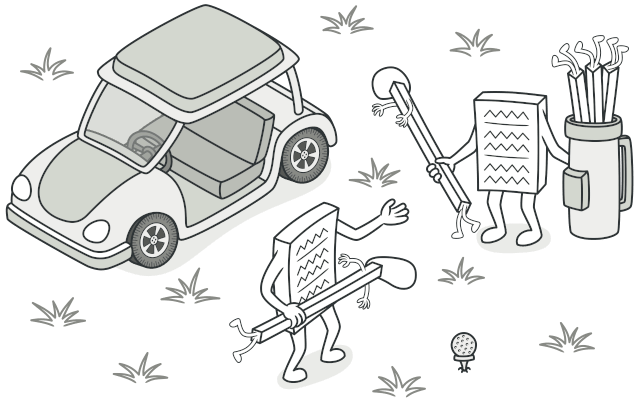
Пустой объект также называется Stub, Active Nothing или Active Null. Это помогает логике нашей программы избавиться от нулевых проверок, где это возможно. Мы все знаем, что мы не можем вызывать методы по нулевой ссылке, так как это приводит кисключению NullReferenceException. Шаблон нулевого объекта предоставляет*нефункциональный*объект вместо нулевой ссылки и, следовательно, позволяет вызывать методы для него.

Таким образом, для некоторых недопустимых входных данных в методGetMobileByName нашMobileRepositoryбудет возвращать экземпляр, но нулевой объект IMobile вместо нулевой ссылки. В отличие от любого другого объекта Mobile, когда мы выполняем какую-либо операцию над этим объектом, ничего не произойдет.

1. **public** **interface** IMobile
2. {
3. **void** TurnOn();
4. **void** TurnOff();
5. }
7. //mobile type implementing IMobile interface
8. **public** **class** SamsungGalaxy : IMobile
9. {
10. **public** **void** TurnOff()
11. {
12. Console.WriteLine("\nSamsung Galaxy Turned OFF!");
13. }
15. **public** **void** TurnOn()
16. {
17. Console.WriteLine("\nSamsung Galaxy Turned ON!");
18. }
19. }
21. //our null object class implementing IMobile interface as a singleton
22. **public** **class** NullMobile : IMobile
23. {
24. **private** **static** NullMobile \_instance;
25. **private** NullMobile()
26. { }
28. **public** **static** NullMobile Instance
29. {
30. **get** {
31. **if** (\_instance == **null**)
32. **return** **new** NullMobile();
33. **return** \_instance;
34. }
35. }
37. //do nothing methods
38. **public** **void** TurnOff()
39. { }
41. **public** **void** TurnOn()
42. { }
43. }

19. Поясните на диаграмме классов как реализовать Strategy.

**Стратегия** – это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

[](https://radioprog.ru/uploads/media/articles/0001/06/f7427d4eb07cda2764cc3270046c5b8391a73ffa.png)

## Проблема

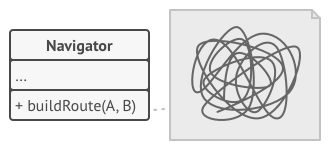
Вы решили написать приложение-навигатор для путешественников. Оно должно показывать красивую и удобную карту, позволяющую с лёгкостью ориентироваться в незнакомом городе.

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

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

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

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

[](https://radioprog.ru/uploads/media/articles/0001/06/115acc27ea2e19c419ebe697c869a78320d595f1.png)*Код навигатора становится слишком раздутым*

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

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

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

## Решение

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

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

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

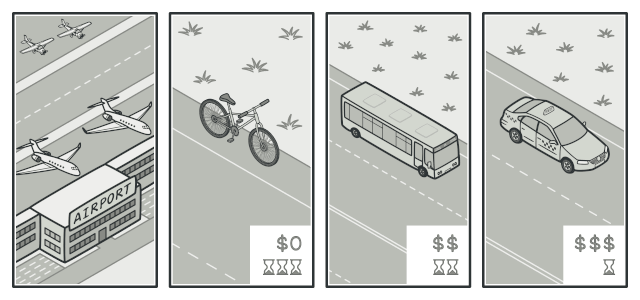
[](https://radioprog.ru/uploads/media/articles/0001/06/8bb9f83b31e61fb2a3965c2f478aa755c22d5c7e.png)*Стратегии построения пути*

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

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

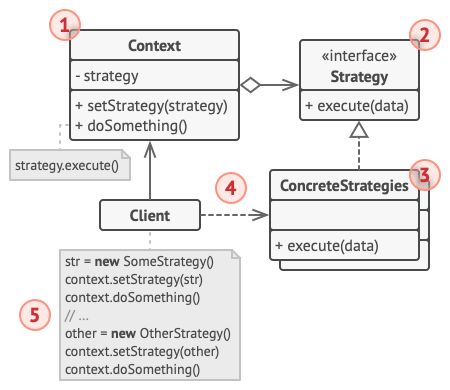
Класс навигатора будет иметь метод для установки стратегии, позволяя изменять стратегию поиска пути на лету. Такой метод пригодится клиентскому коду навигатора, например, переключателям типов маршрутов в пользовательском интерфейсе.

## Аналогия из жизни

[](https://radioprog.ru/uploads/media/articles/0001/06/48130d43ff60587ce6c7c0f55adb4f90a6b9fdb5.png)*Различные стратегии попадания в аэропорт*

Вам нужно добраться до аэропорта. Можно доехать на автобусе, такси или велосипеде. Здесь вид транспорта является стратегией. Вы выбираете конкретную стратегию в зависимости от контекста – наличия денег или времени до отлёта.

## Структура

[](https://radioprog.ru/uploads/media/articles/0001/06/269e58713229ff1142175c049acc355e52eb1901.png)*Структура классов паттерна Стратегия*

20. Перечислите и поясните принципы проектирования SOLID.

Термин "SOLID" представляет собой акроним для набора практик проектирования программного кода и построения гибкой и адаптивной программы. Данный термин был введен известным американским специалистом в области программирования Робертом Мартином (Robert Martin), более известным как "дядюшка Боб" или Uncle Bob.

Сам акроним образован по первым буквам названий SOLID-принципов:

* **S**ingle Responsibility Principle (Принцип единственной обязанности)
* **O**pen/Closed Principle (Принцип открытости/закрытости)
* **L**iskov Substitution Principle (Принцип подстановки Лисков)
* **I**nterface Segregation Principle (Принцип разделения интерфейсов)
* **D**ependency Inversion Principle (Принцип инверсии зависимостей)

Принципы SOLID - это не паттерны, их нельзя назвать какими-то определенными догмами, которые надо обязательно применять при разработке, однако их использование позволит улучшить код программы, упростить возможные его изменения и поддержку.

## Принцип единственной обязанности

**Принцип единственной обязанности** (Single Responsibility Principle) можно сформулировать так:

**Каждый компонент должен иметь одну и только одну причину для изменения.**.

В C# в качестве компонента может выступать класс, структура, метод. А под обязанностью здесь понимается набор действий, которые выполняют единую задачу. То есть суть принципа заключается в том, что класс/структура/метод должны выполнять одну единственную задачу. Весь функционал компонента должен быть целостным, обладать высокой связностью (high cohesion).

Конкретное применение принципа зависит от контекста. В данном случае важно понимать, как изменяется компонент. Если он выполняет несколько различных действий, и они изменяются по отдельности, то это как раз тот случай, когда можно применить принцип единственной обязанности. То есть иными словами, у компонента несколько причин для изменения.