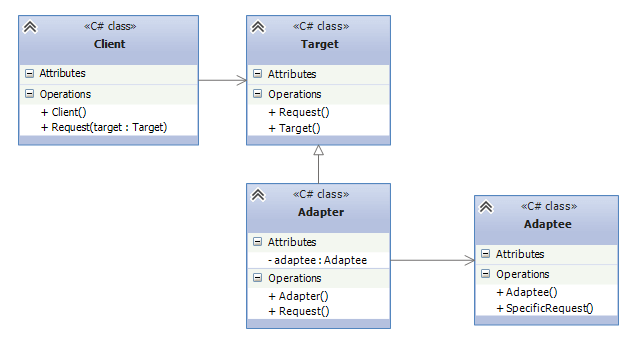
№ 19-20 Применение структурных и паттернов поведения

1. Назначение структурных паттернов?

Структурные паттерны помогают решать задачи с тем, как совмещать и сочетать сущности вместе. Они заботятся о том, как сущности могут использовать друг друга. Простыми словами — они отвечают на вопрос «Как составить программный компонент так, чтобы его можно было компоновать с другими?».

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

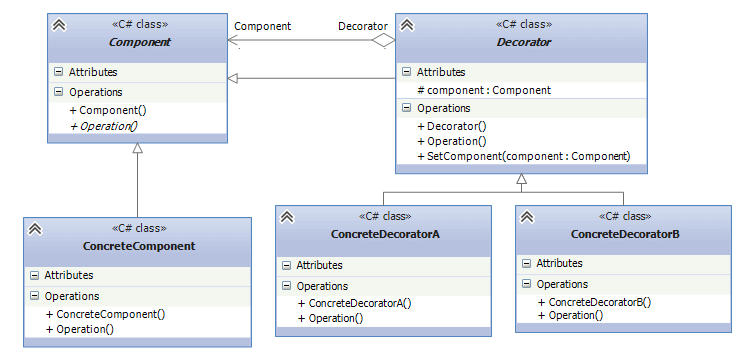
Паттерн Адаптер (Adapter) предназначен для преобразования интерфейса одного класса в интерфейс другого. Благодаря реализации данного паттерна мы можем использовать вместе классы с несовместимыми интерфейсами.



1. В чем разница между адаптером класса и адаптером объекта?
2. 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.



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

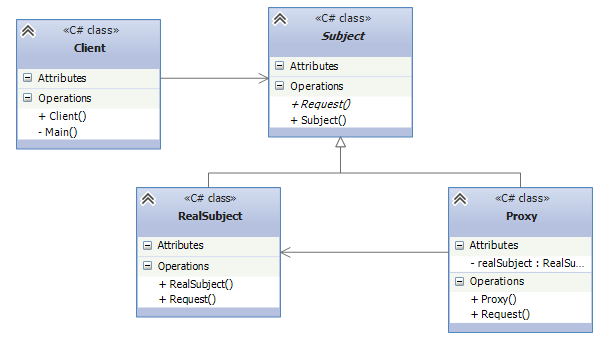
Паттерн Заместитель (Proxy) предоставляет объект-заместитель, который управляет доступом к другому объекту. То есть создается объект-суррогат, который может выступать в роли другого объекта и замещать его.

Разновидности  
- Удаленный прокси (Remote proxy) – обеспечивает взаимодействие с объектом в другом адресном пространстве.

- Виртуальный прокси (Virtual proxy) – выполняет оптимизацию доступа к объекту, может самостоятельно обрабатывать некоторые запросы.

- Защищающий прокси (Protection proxy) – управляет объектом, разграничивая права доступа различных клиентов.

- Умная ссылка (Smart reference) – обеспечивает выполнение дополнительных действий при вызове методов объекта. Примерами могут служить подсчет ссылок или обеспечение потокобезопасности работы с объектом.



1. В чем разница между паттернами Decorator и Adapter?

Отличие декоратора от адаптера в том, что адаптер имеет внешний интерфейс, отличный от интерфейса оборачиваемого объекта, и используется именно для стыковки разных интерфейсов. Декоратор же имеет точно такой же интерфейс, и используется для добавления функциональности.

1. 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

1. Расскажите о паттерне Façade?

Фасад (Facade) представляет шаблон проектирования, который позволяет скрыть сложность системы с помощью предоставления упрощенного интерфейса для взаимодействия с ней.

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.

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

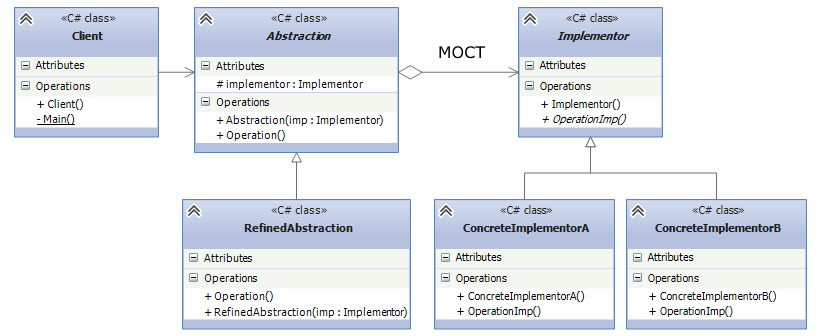
Мост (Bridge) - структурный шаблон проектирования, который позволяет отделить абстракцию от реализации таким образом, чтобы и абстракцию, и реализацию можно было изменять независимо друг от друга.

Даже если мы отделим абстракцию от конкретных реализаций, то у нас все равно все наследуемые классы будут жестко привязаны к интерфейсу, определяемому в базовом абстрактном классе. Для преодоления жестких связей и служит паттерн Мост.

Когда использовать данный паттерн?

Когда надо избежать постоянной привязки абстракции к реализации

Когда наряду с реализацией надо изменять и абстракцию независимо друг от друга. То есть изменения в абстракции не должно привести к изменениям в реализации



1. Назначение паттернов поведения?

Позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации. Создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах. Позволяет объектам менять поведение в зависимости от своего состояния.

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

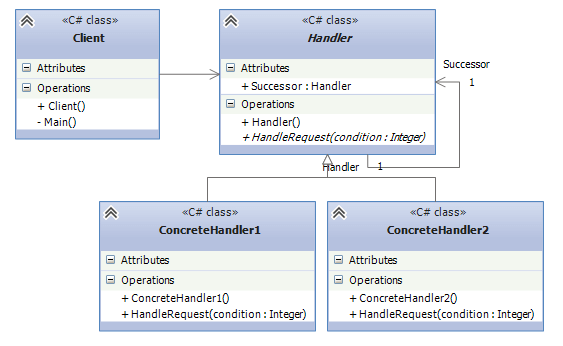
Цепочка Обязанностей (Chain of responsibility) - поведенческий шаблон проектирования, который позволяет избежать жесткой привязки отправителя запроса к получателю. Все возможные обработчики запроса образуют цепочку, а сам запрос перемещается по этой цепочке. Каждый объект в этой цепочке при получении запроса выбирает, либо закончить обработку запроса, либо передать запрос на обработку следующему по цепочке объекту.

Когда применяется цепочка обязанностей?

Когда имеется более одного объекта, который может обработать определенный запрос

Когда надо передать запрос на выполнение одному из нескольких объект, точно не определяя, какому именно объекту

Когда набор объектов задается динамически



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

Паттерн "Команда" (Command) позволяет инкапсулировать запрос на выполнение определенного действия в виде отдельного объекта. Этот объект запроса на действие и называется командой. При этом объекты, инициирующие запросы на выполнение действия, отделяются от объектов, которые выполняют это действие.

Команды могут использовать параметры, которые передают ассоциированную с командой информацию. Кроме того, команды могут ставиться в очередь и также могут быть отменены.

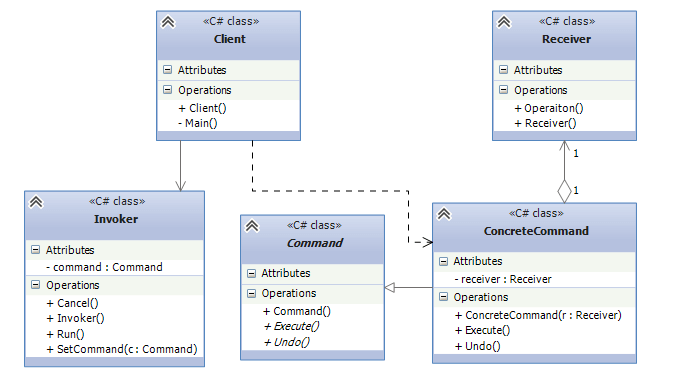
Когда использовать команды?

Когда надо передавать в качестве параметров определенные действия, вызываемые в ответ на другие действия. То есть когда необходимы функции обратного действия в ответ на определенные действия.

Когда необходимо обеспечить выполнение очереди запросов, а также их возможную отмену.

Когда надо поддерживать логгирование изменений в результате запросов. Использование логов может помочь восстановить состояние системы - для этого необходимо будет использовать последовательность запротоколированных команд.

Схематично в UML паттерн Команда представляется следующим образом:



1. Как реализовать паттерн Observer?

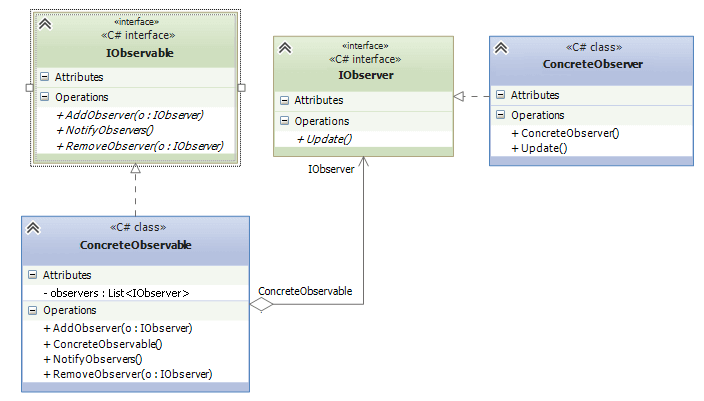
Паттерн "Наблюдатель" (Observer) представляет поведенческий шаблон проектирования, который использует отношение "один ко многим". В этом отношении есть один наблюдаемый объект и множество наблюдателей. И при изменении наблюдаемого объекта автоматически происходит оповещение всех наблюдателей.

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.

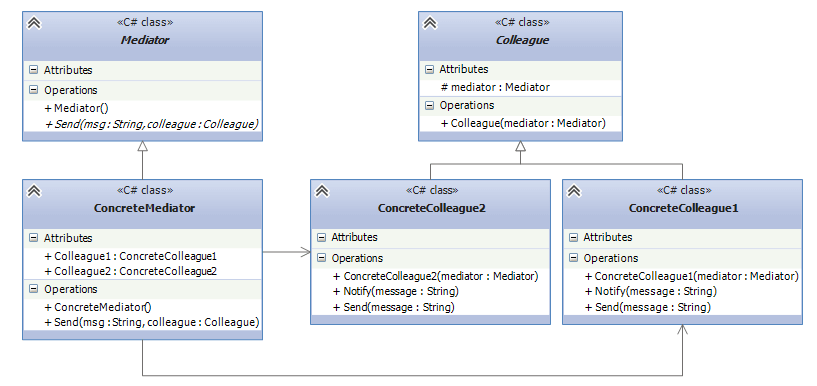
1. Нарисуйте диаграмму классов для паттерна Mediator. Поясните его назначение.

Паттерн Посредник (Mediator) представляет такой шаблон проектирования, который обеспечивает взаимодействие множества объектов без необходимости ссылаться друг на друга. Тем самым достигается слабосвязанность взаимодействующих объектов.

Когда используется паттерн Посредник?

Когда имеется множество взаимосвязаных объектов, связи между которыми сложны и запутаны.

Когда необходимо повторно использовать объект, однако повторное использование затруднено в силу сильных связей с другими объектами.



1. В чем разница между паттернами Mediator и Facade?

Принципиальное различие между этими двумя паттернами заключается в том, что фасад, как структурный паттерн, всего лишь передает существующую функциональность в медиатор, в то время как медиатор, как поведенческий паттерн, может эту функциональность расширять.

1. 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.

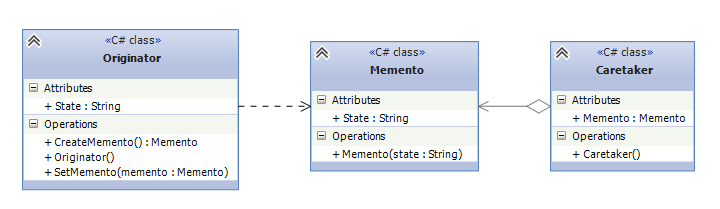
Когда использовать Memento?

Когда нужно сохранить состояние объекта для возможного последующего восстановления

Когда сохранение состояния должно проходить без нарушения принципа инкапсуляции

То есть ключевыми понятиями для данного паттерна являются сохранение внутреннего состояния и инкапсуляция, и важно соблюсти баланс между ними. Ведь, как правило, если мы не нарушаем инкапсуляцию, то состояние объекта хранится в объекте в приватных переменных. И не всегда для доступа к этим переменным есть методы или свойства с сеттерами и геттерами. Например, в игре происходит управление героем, все состояние которого заключено в нем самом - оружие героя, показатель жизней, силы, какие-то другие показатели. И нередко может возникнуть ситуация, сохранить все эти показатели во вне, чтобы в будущем можно было откатиться к предыдущему уровню и начать игру заново. В этом случае как раз и может помочь паттерн Хранитель.

С помощью диаграмм структуру паттерна можно изобразить следующим образом:



1. Расскажите о паттерне Visitor?

Паттерн Посетитель (Visitor) позволяет определить операцию для объектов других классов без изменения этих классов.

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.

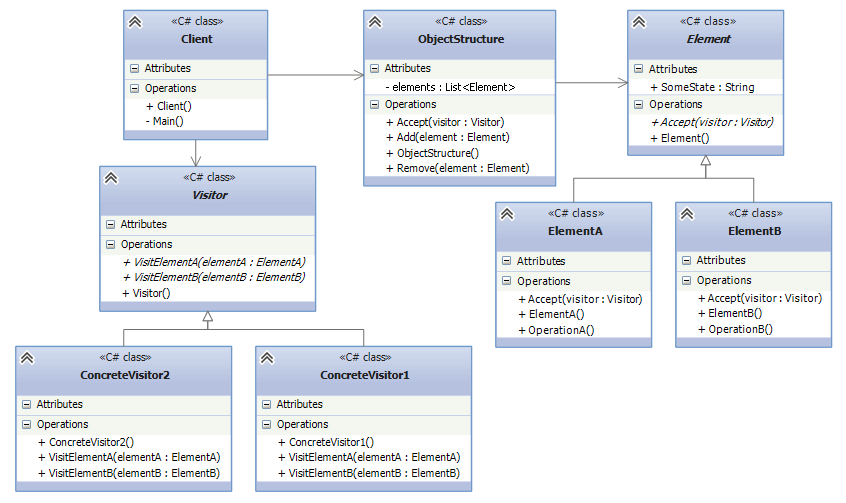
Когда использовать данный паттерн?

Когда имеется много объектов разнородных классов с разными интерфейсами, и требуется выполнить ряд операций над каждым из этих объектов

Когда классам необходимо добавить одинаковый набор операций без изменения этих классов

Когда часто добавляются новые операции к классам, при этом общая структура классов стабильна и практически не изменяется

Общая структура классов паттерна, описанная с помощью диаграмм UML:



1. В каких случаях надо применять Null object?
2. Поясните на диаграмме классов как реализовать Strategy.

Паттерн Стратегия (Strategy) представляет шаблон проектирования, который определяет набор алгоритмов, инкапсулирует каждый из них и обеспечивает их взаимозаменяемость. В зависимости от ситуации мы можем легко заменить один используемый алгоритм другим. При этом замена алгоритма происходит независимо от объекта, который использует данный алгоритм.

Когда использовать стратегию?

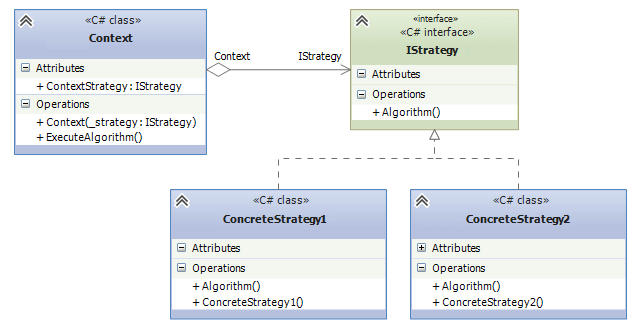
Когда есть несколько родственных классов, которые отличаются поведением. Можно задать один основной класс, а разные варианты поведения вынести в отдельные классы и при необходимости их применять

Когда необходимо обеспечить выбор из нескольких вариантов алгоритмов, которые можно легко менять в зависимости от условий

Когда необходимо менять поведение объектов на стадии выполнения программы

Когда класс, применяющий определенную функциональность, ничего не должен знать о ее реализации

Формально паттерн Стратегия можно выразить следующей схемой UML:



1. Перечислите и поясните принципы проектирования SOLID

****Расшифровка:****

* Single responsibility — принцип единственной ответственности
* Open-closed — принцип открытости / закрытости
* Liskov substitution — принцип подстановки Барбары Лисков
* Interface segregation — принцип разделения интерфейса
* Dependency inversion — принцип инверсии зависимостей

****Принцип единственной обязанности / ответственности**** (single responsibility principle / SRP) обозначает, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс. Все его сервисы должны быть направлены исключительно на обеспечение этой обязанности. [Подробнее про SRP...](https://web-creator.ru/articles/solid_the_single_responsibility_principle)

****Принцип открытости / закрытости**** (open-closed principle / OCP) декларирует, что программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода. [Подробнее про OCP...](https://web-creator.ru/articles/solid_the_open_closed_principle)

****Принцип подстановки Барбары Лисков**** (Liskov substitution principle / LSP) в формулировке Роберта Мартина: «функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом». [Подробнее про LSP...](https://web-creator.ru/articles/solid_the_liskov_substitution_principle)

****Принцип разделения интерфейса**** (interface segregation principle / ISP) в формулировке Роберта Мартина: «клиенты не должны зависеть от методов, которые они не используют». Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют. [Подробнее про ISP...](https://web-creator.ru/articles/solid_the_interface_segregation_principle)

****Принцип инверсии зависимостей**** (dependency inversion principle / DIP) — модули верхних уровней не должны зависеть от модулей нижних уровней, а оба типа модулей должны зависеть от абстракций; сами абстракции не должны зависеть от деталей, а вот детали должны зависеть от абстракций. [Подробнее про DIP...](https://web-creator.ru/articles/solid_the_dependency_inversion_principle)