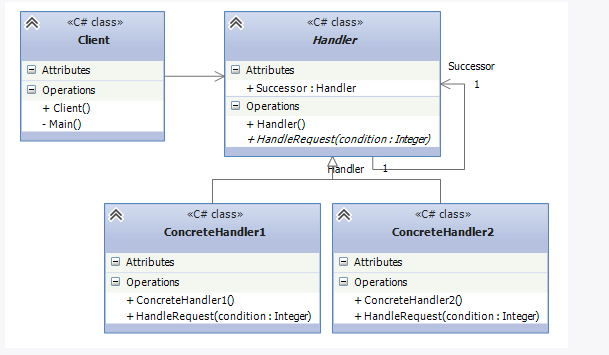
1. **Поведенческие паттерны** заботятся об эффективной коммуникации между объектами.

Решают задачи эффективного и безопасного взаимодействия между объектами программы.

2. Цепочка Обязанностей (Chain of responsibility) - поведенческий шаблон проектирования, который позволяет избежать жесткой привязки отправителя запроса к получателю, позволяя нескольким объектам обработать запрос. Все возможные обработчики запроса образуют цепочку, а сам запрос перемещается по этой цепочке, пока один из ее объектов не обработает запрос. Каждый объект при получении запроса выбирает, либо обработать запрос, либо передать выполнение запроса следующему по цепочке.

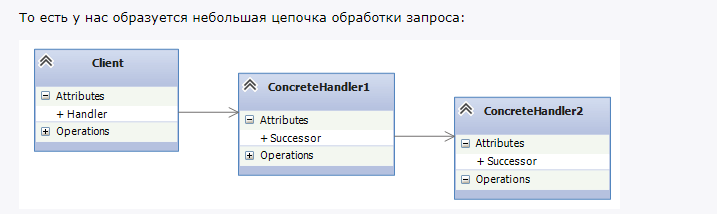
* Когда имеется более одного объекта, который может обработать определенный запрос
* Когда надо передать запрос на выполнение одному из нескольких объект, точно не определяя, какому именно объекту
* Когда набор объектов задается динамически



* **Handler**: определяет интерфейс для обработки запроса. Также может определять ссылку на следующий обработчик запроса
* **ConcreteHandler1** и **ConcreteHandler2**: конкретные обработчики, которые реализуют функционал для обработки запроса. При невозможности обработки и наличия ссылки на следующий обработчик, передают запрос этому обработчику

В данном случае для простоты примера в качестве параметра передается некоторое число, и в зависимости от значения данного числа обработчики и принимают решения об обработке запроса.

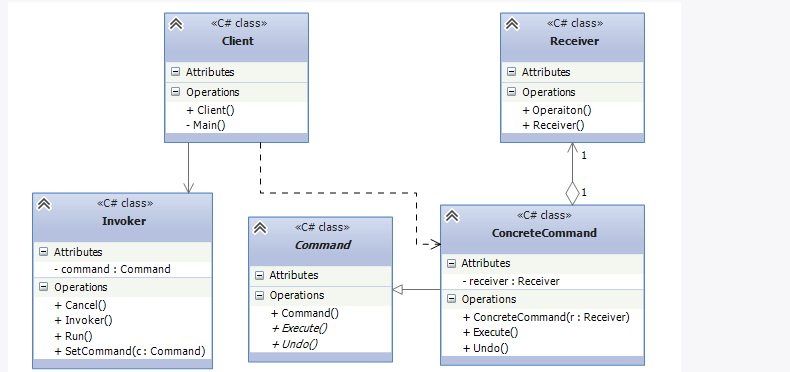
* **Client**: отправляет запрос объекту Handler



3. 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.

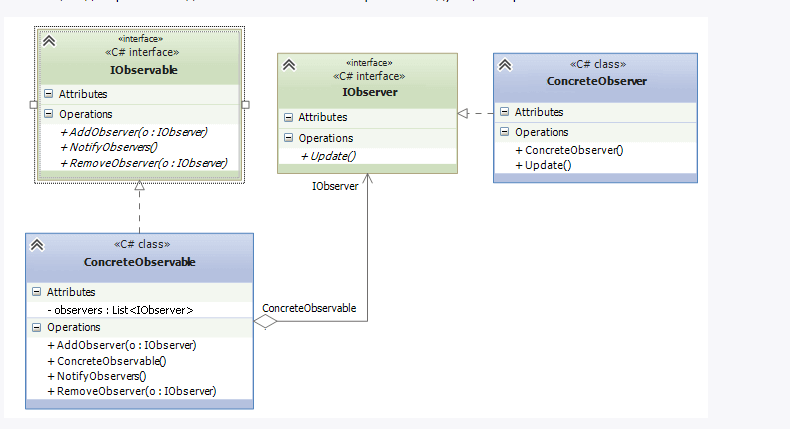
* Когда надо передавать в качестве параметров определенные действия, вызываемые в ответ на другие действия. То есть когда необходимы функции обратного действия в ответ на определенные действия.
* Когда необходимо обеспечить выполнение очереди запросов, а также их возможную отмену.
* Когда надо поддерживать логгирование изменений в результате запросов. Использование логов может помочь восстановить состояние системы - для этого необходимо будет использовать последовательность запротоколированных команд.



* **Command**: интерфейс, представляющий команду. Обычно определяет метод Execute() для выполнения действия, а также нередко включает метод Undo(), реализация которого должна заключаться в отмене действия команды
* **ConcreteCommand**: конкретная реализация команды, реализует метод Execute(), в котором вызывается определенный метод, определенный в классе Receiver
* **Receiver**: получатель команды. Определяет действия, которые должны выполняться в результате запроса.
* **Invoker**: инициатор команды - вызывает команду для выполнения определенного запроса
* **Client**: клиент - создает команду и устанавливает ее получателя с помощью метода SetCommand()

4. 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.

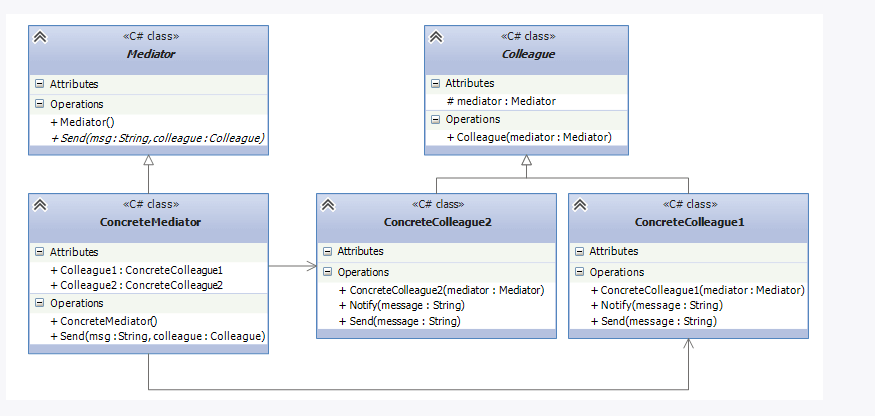
* Когда система состоит из множества классов, объекты которых должны находиться в согласованных состояниях
* Когда общая схема взаимодействия объектов предполагает две стороны: одна рассылает сообщения и является главным, другая получает сообщения и реагирует на них. Отделение логики обеих сторон позволяет 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.



* **IObservable**: представляет наблюдаемый объект. Определяет три метода: AddObserver() (для добавления наблюдателя), RemoveObserver() (удаление набюдателя) и NotifyObservers() (уведомление наблюдателей)
* **ConcreteObservable**: конкретная реализация интерфейса IObservable. Определяет коллекцию объектов наблюдателей.
* **IObserver**: представляет наблюдателя, который подписывается на все уведомления наблюдаемого объекта. Определяет метод Update(), который вызывается наблюдаемым объектом для уведомления наблюдателя.
* **ConcreteObserver**: конкретная реализация интерфейса IObserver.

5. 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 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.



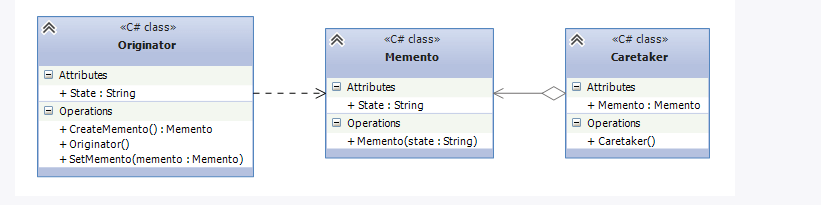
* **Mediator**: представляет интерфейс для взаимодействия с объектами Colleague
* **Colleague**: представляет интерфейс для взаимодействия с объектом Mediator
* **ConcreteColleague1** и **ConcreteColleague2**: конкретные классы коллег, которые обмениваются друг с другом через объект Mediator
* **ConcreteMediator**: конкретный посредник, реализующий интерфейс типа Mediator

6. Фасад создаёт упрощённый интерфейс к подсистеме, классы подсистемы общаются друг с другом напрямую.

Посредник централизует общение между компонентами системы.

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

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

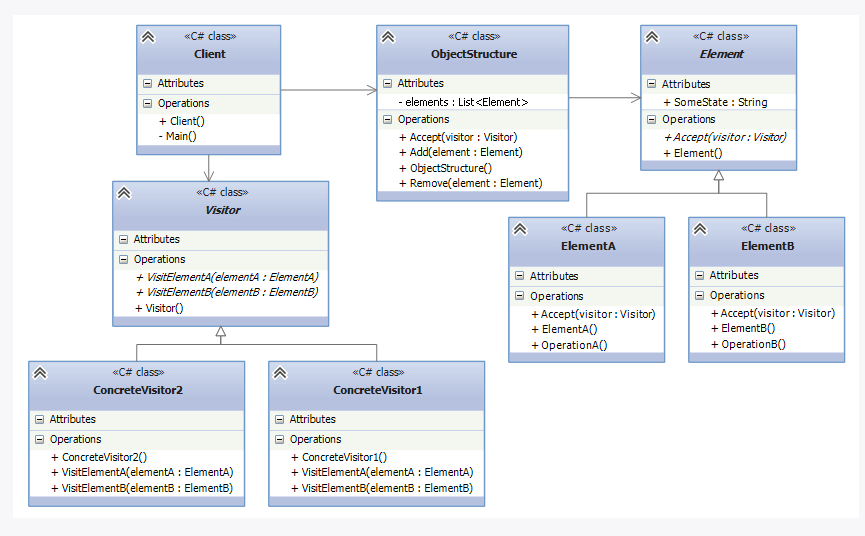


* **Memento**: хранитель, который сохраняет состояние объекта Originator и предоставляет полный доступ только этому объекту Originator
* **Originator**: создает объект хранителя для сохранения своего состояния
* **Caretaker**: выполняет только функцию хранения объекта Memento, в то же время у него нет полного доступа к хранителю и никаких других операций над хранителем, кроме собственно сохранения, он не производит

8. 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.

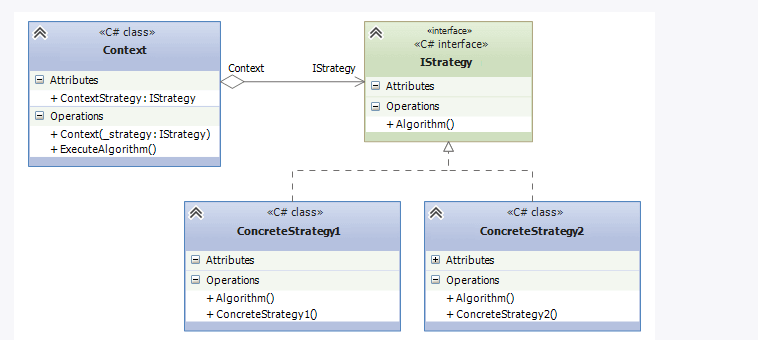
* Когда имеется много объектов разнородных классов с разными интерфейсами, и требуется выполнить ряд операций над каждым из этих объектов
* Когда классам необходимо добавить одинаковый набор операций без изменения этих классов
* Когда часто добавляются новые операции к классам, при этом общая структура классов стабильна и практически не изменяется



* **Visitor**: интерфейс посетителя, который определяет метод Visit() для каждого объекта Element
* **ConcreteVisitor1 / ConcreteVisitor2**: конкретные классы посетителей, реализуют интерфейс, определенный в Visitor.
* **Element**: определяет метод Accept(), в котором в качестве параметра принимается объект Visitor
* **ElementA / ElementB**: конкретные элементы, которые реализуют метод Accept()
* **ObjectStructure**: некоторая структура, которая хранит объекты Element и предоставляет к ним доступ. Это могут быть и простые списки, и сложные составные структуры в виде деревьев

9. Нулевой объект–это объект с заданным нейтральным поведением, использование которого позволяет заменить условные операторы (проверки на null)полиморфизмом

10. 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 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
* Интерфейс IStrategy, который определяет метод Algorithm(). Это общий интерфейс для всех реализующих его алгоритмов. Вместо интерфейса здесь также можно было бы использовать абстрактный класс.
* Классы ConcreteStrategy1 и ConcreteStrategy, которые реализуют интерфейс IStrategy, предоставляя свою версию метода Algorithm(). Подобных классов-реализаций может быть множество.
* Класс Context хранит ссылку на объект IStrategy и связан с интерфейсом IStrategy отношением агрегации.

11. Термин "SOLID" представляет собой акроним для набора практик проектирования программного кода и построения гибкой и адаптивной программы. Данный термин был введен 15 лет назад известным американским специалистом в области программирования Робертом Мартином (Robert Martin), более известным как "дядюшка Боб" или Uncle Bob (Bob - сокращение от имени Robert).

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** (Single Responsibility Principle) можно сформулировать так:

**У класса должна быть только одна причина для изменения**

Под обязанностью здесь понимается набор функций, которые выполняют единую задачу. Суть этого принципа заключается в том, что класс должен выполнять одну единственную задачу. Весь функционал класса должен быть целостным, обладать высокой связностью (high cohesion).

**Принцип открытости/закрытости** (Open/Closed Principle) можно сформулировать так:

**Сущности программы должны быть открыты для расширения, но закрыты для изменения**.

Суть этого принципа состоит в том, что система должна быть построена таким образом, что все ее последующие изменения должны быть реализованы с помощью добавления нового кода, а не изменения уже существующего.

**Принцип подстановки Лисков** (Liskov Substitution Principle) представляет собой некоторое руководство по созданию иерархий наследования. Изначальное определение данного принципа, которое было дано Барбарой Лисков в 1988 году, выглядело следующим образом:

Если для каждого объекта o1 типа S существует объект o2 типа T, такой, что для любой программы P, определенной в терминах T, поведение P не изменяется при замене o2 на o1, то S является подтипом T.

То есть иными словами класс S может считаться подклассом T, если замена объектов T на объекты S не приведет к изменению работы программы.

В общем случае данный принцип можно сформулировать так:

**Должна быть возможность вместо базового типа подставить любой его подтип**.

Фактически принцип подстановки Лисков помогает четче сформулировать иерархию классов, определить функционал для базовых и производных классов и избежать возможных проблем при применении полиморфизма.

**Принцип разделения интерфейсов** (Interface Segregation Principle) относится к тем случаям, когда классы имеют "жирный интерфейс", то есть слишком раздутый интерфейс, не все методы и свойства которого используются и могут быть востребованы. Таким образом, интерфейс получатся слишком избыточен или "жирным".

Принцип разделения интерфейсов можно сформулировать так:

**Клиенты не должны вынужденно зависеть от методов, которыми не пользуются.**

**Принцип инверсии зависимостей** (Dependency Inversion Principle) служит для создания слабосвязанных сущностей, которые легко тестировать, модифицировать и обновлять. Этот принцип можно сформулировать следующим образом:

**Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те и другие должны зависеть от абстракций.**

**Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.**