

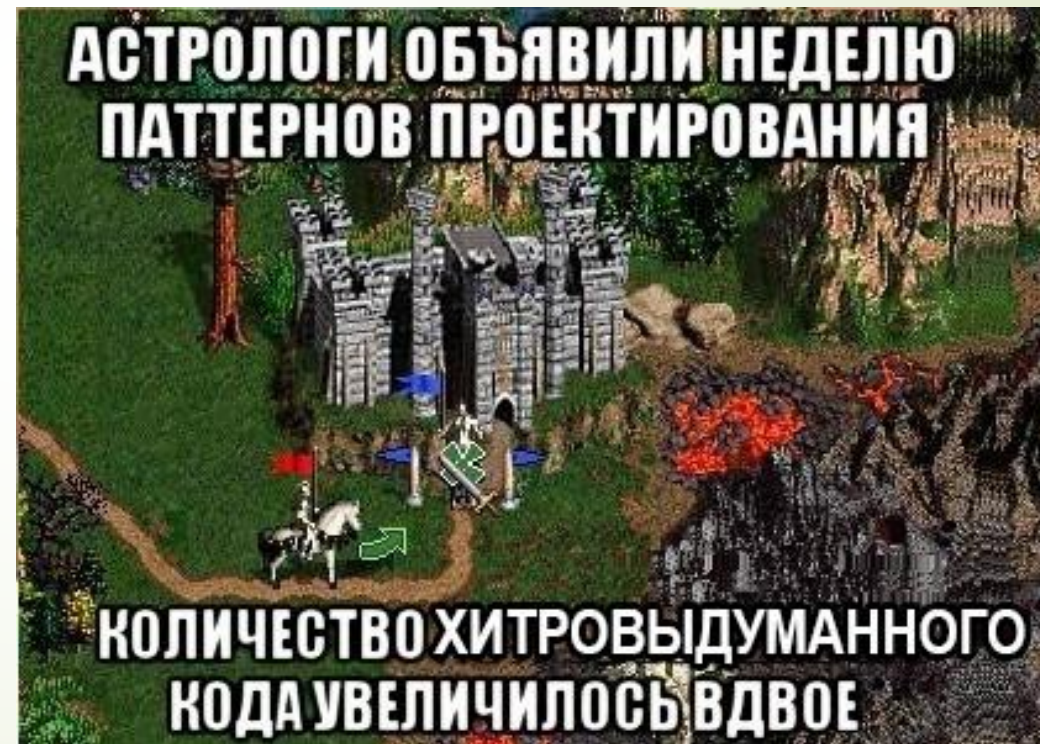


# Лекция 5

## Паттерны GoF. Часть 2

# В прошлой серии...

- Порождающие паттерны GoF
  - Singleton
  - Factory Method
  - Abstract Factory
  - Builder
  - Prototype



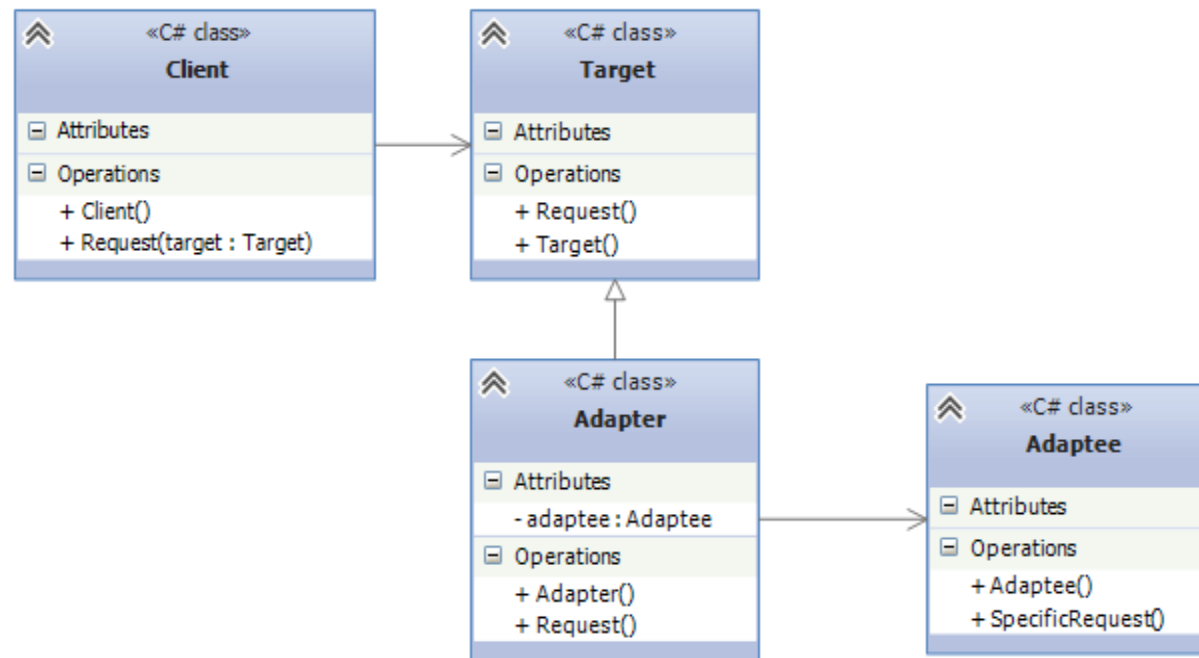


# Структурные паттерны

- Adapter (Адаптер)
- Bridge (Мост)
- Composite (Компоновщик)
- Decorator (Декоратор)
- Facade (Фасад)
- Flyweight (Приспособленец)
- Proxy (Заместитель)

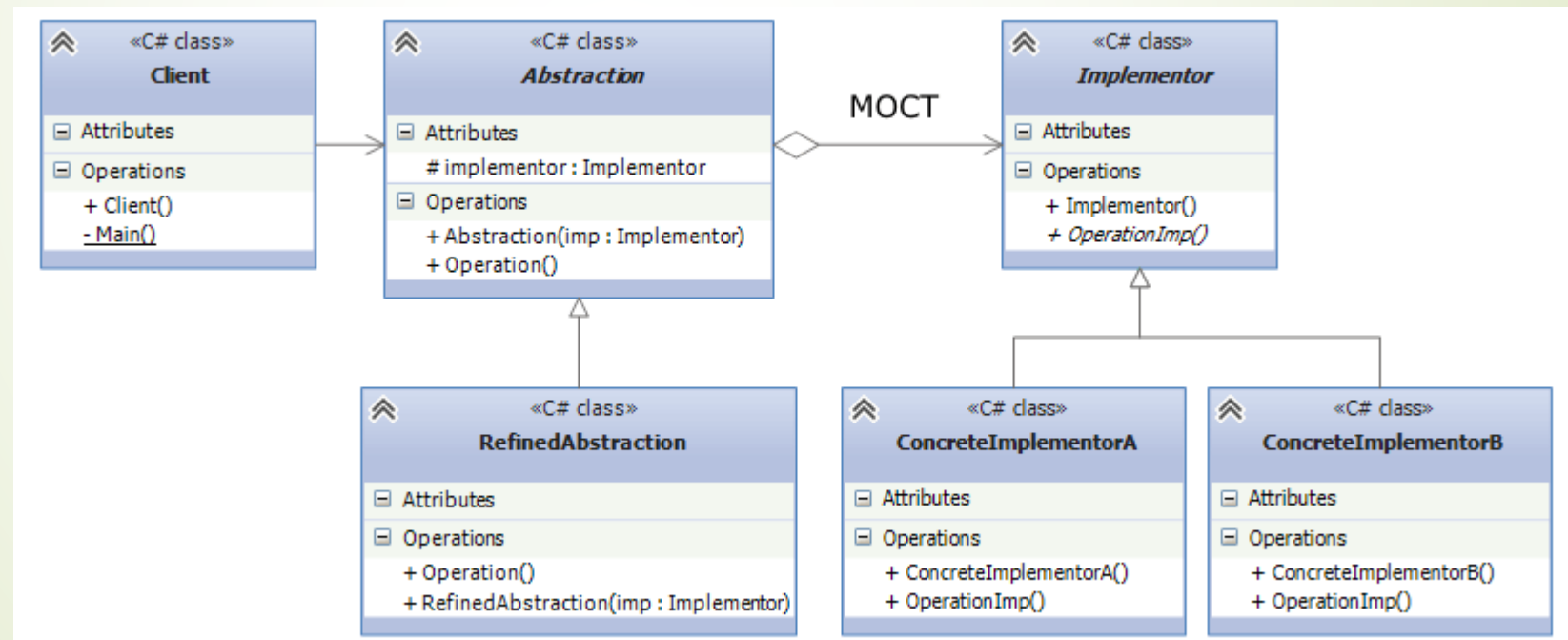
# Adapter

- Адаптер преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер обеспечивает совместную работу классов с несовместимыми интерфейсами, которая без него была бы невозможна (пример: создание текстового элемента в графическом редакторе)

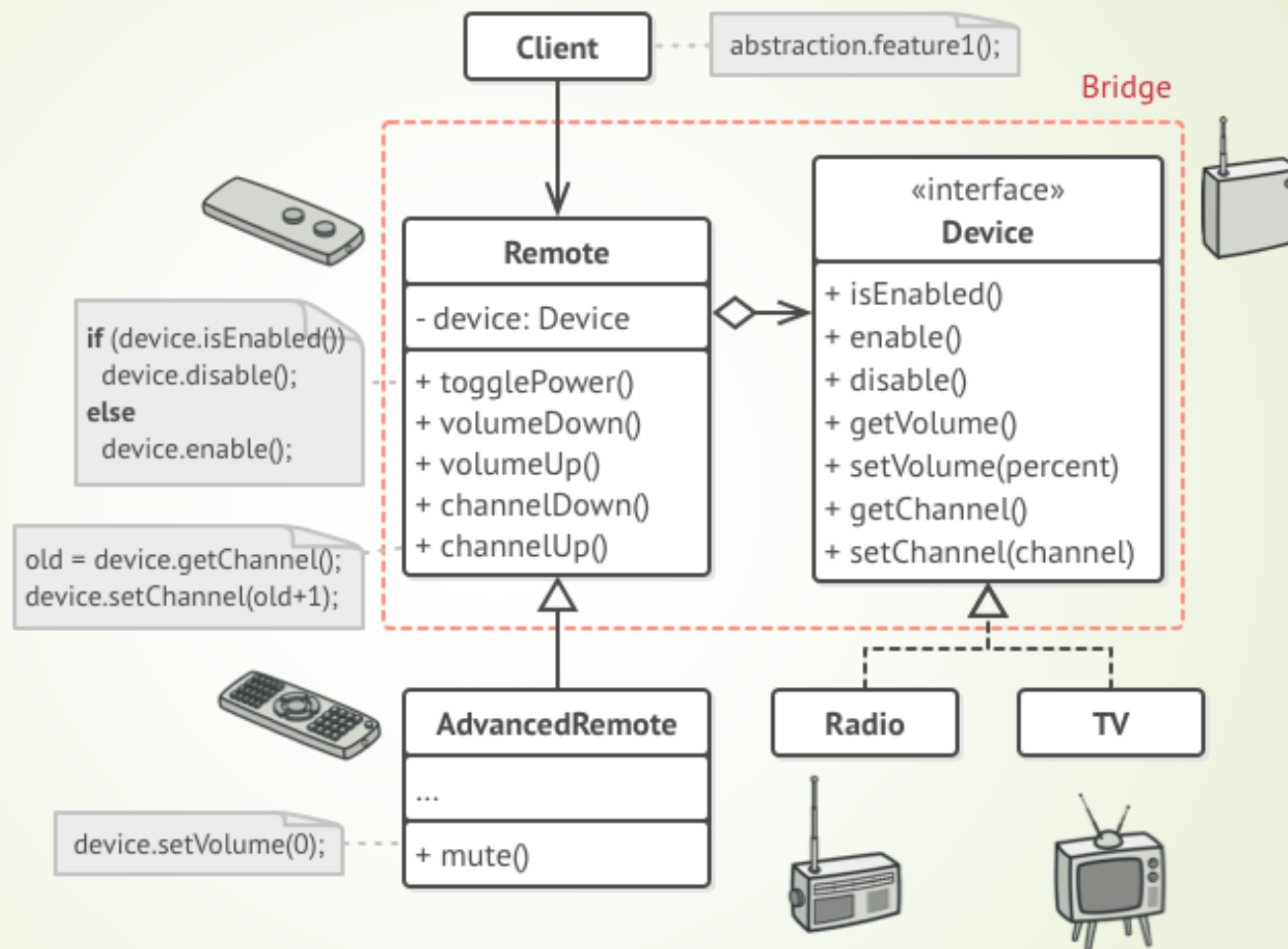


# Bridge

- Необходим для отделения абстракции от ее реализации так, чтобы то и другое можно было изменять независимо. Применяется, когда и абстракции, и реализации должны расширяться новыми подклассами. В таком случае паттерн мост позволяет комбинировать разные абстракции и реализации и изменять их независимо.








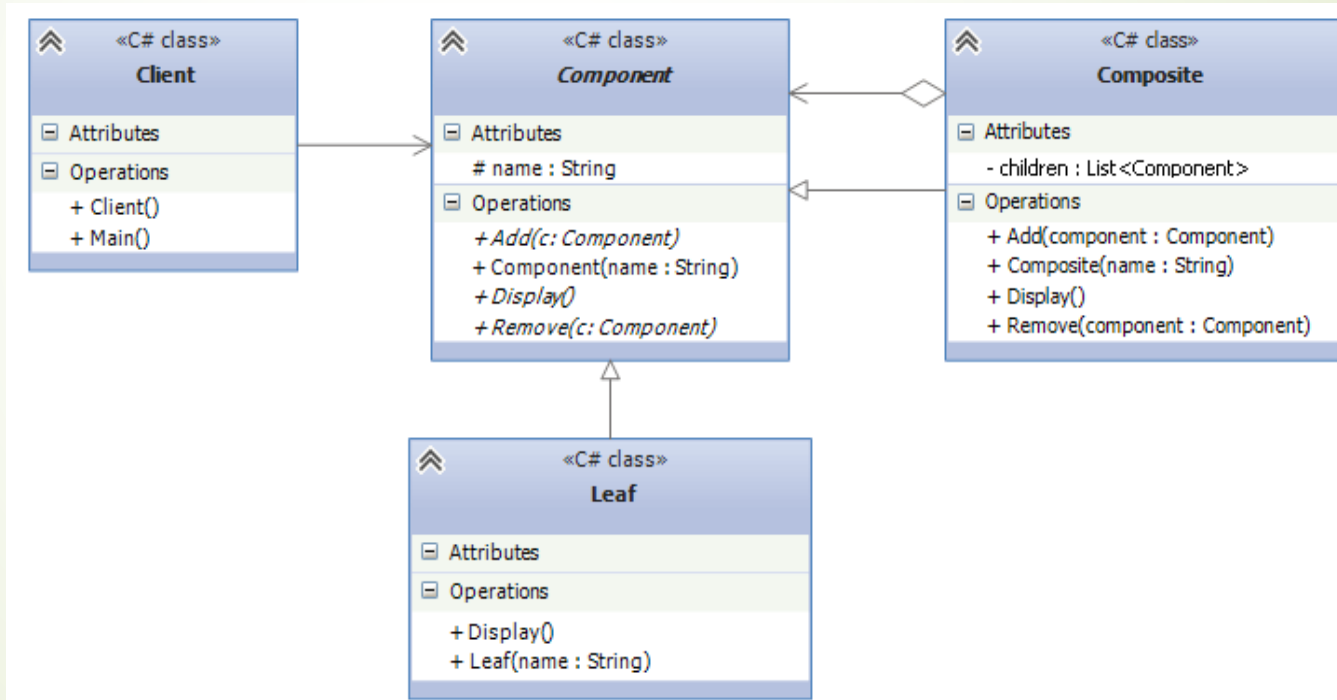


# Особенности

- отделение реализации от интерфейса. Реализация больше не имеет постоянной привязки к интерфейсу. Реализацию абстракции можно конфигурировать во время выполнения. Объект может даже динамически изменять свою реализацию
  - повышение степени расширяемости. Можно расширять независимо иерархии классов Abstraction и Implementor
- 

# Composite

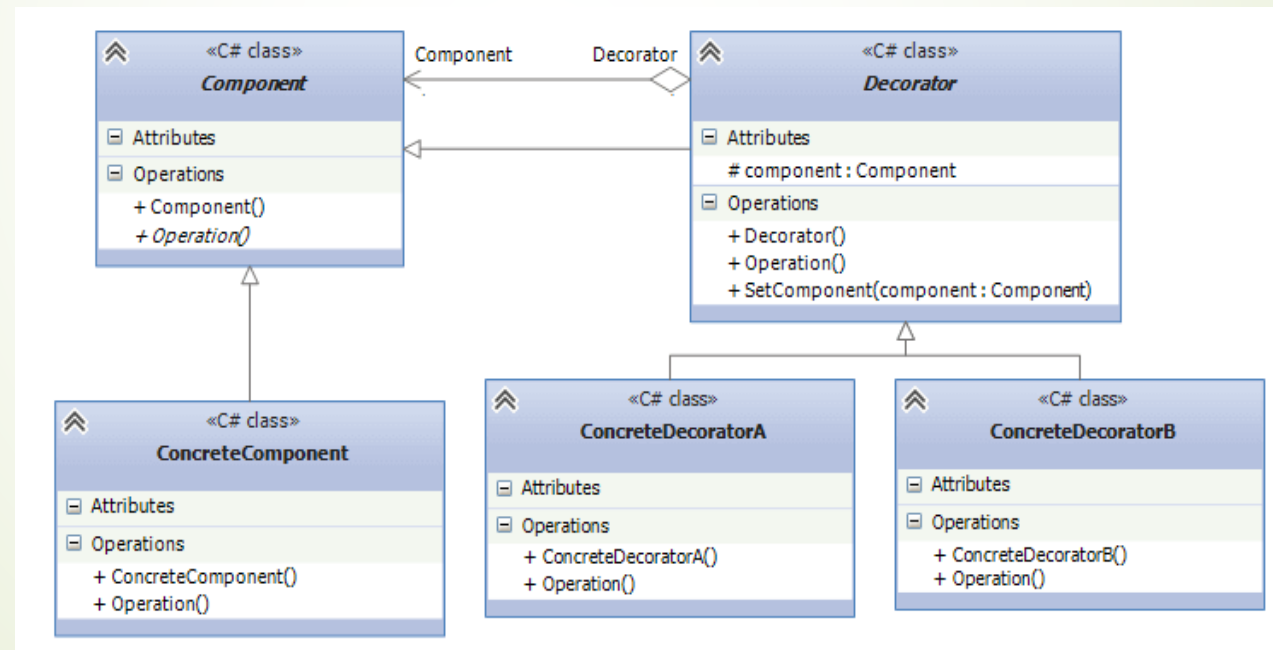
- Компонует объекты в древовидные структуры для представления иерархий часть-целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты





# Decorator

- Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.





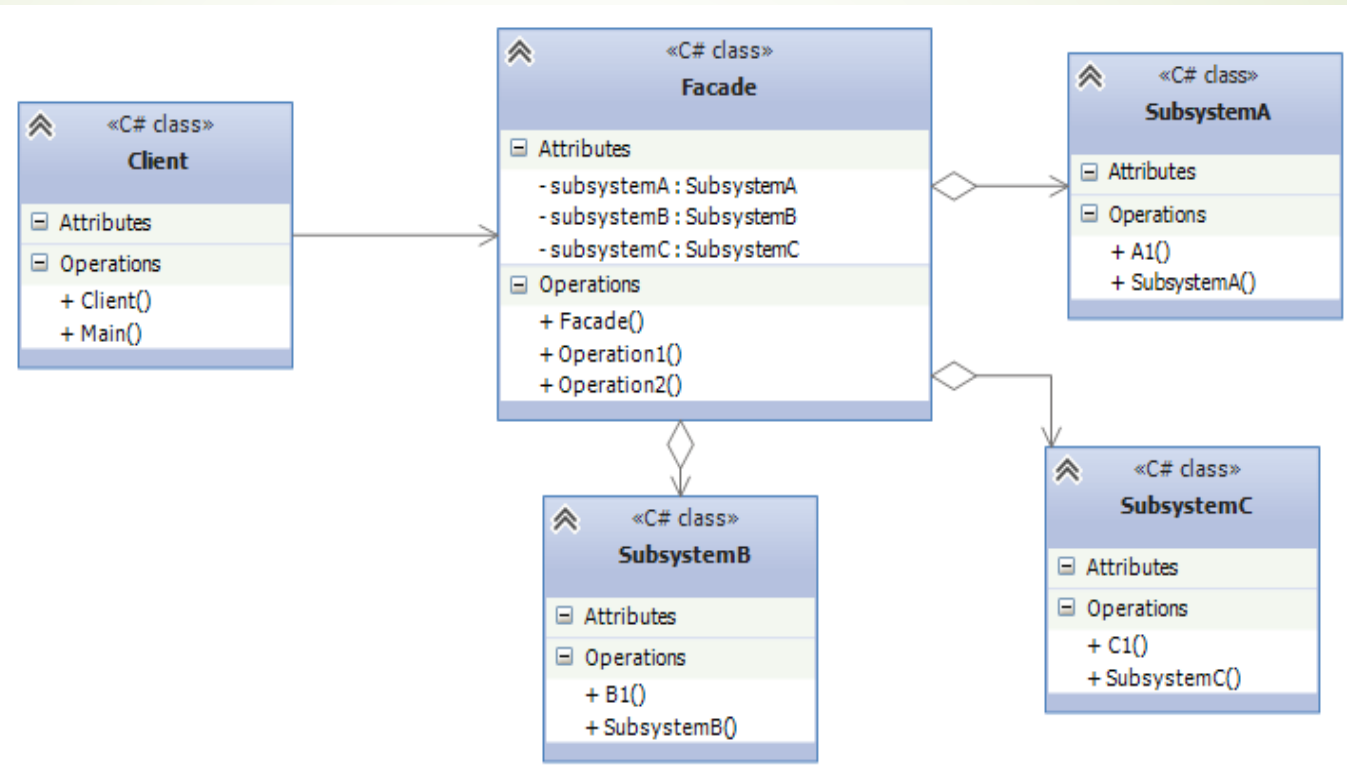
# Особенности

- ▶ паттерн декоратор позволяет более гибко добавлять объекту новые обязанности, чем было бы возможно в случае статического (множественного) наследования
- ▶ позволяет избежать перегруженных функциями классов на верхних уровнях иерархии. Декоратор разрешает добавлять новые обязанности по мере необходимости вместо того чтобы пытаться поддержать все мыслимые возможности в одном сложном, допускающем разностороннюю настройку классе
- ▶ декоратор действует как прозрачное обрамление. Но декорированный компонент все же не идентичен исходному.



# Facade

- Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.
- Применяется для упрощения работы со сложной системы. Фасад позволяет определить одну точку взаимодействия между клиентом и системой, чем уменьшает количество зависимостей.

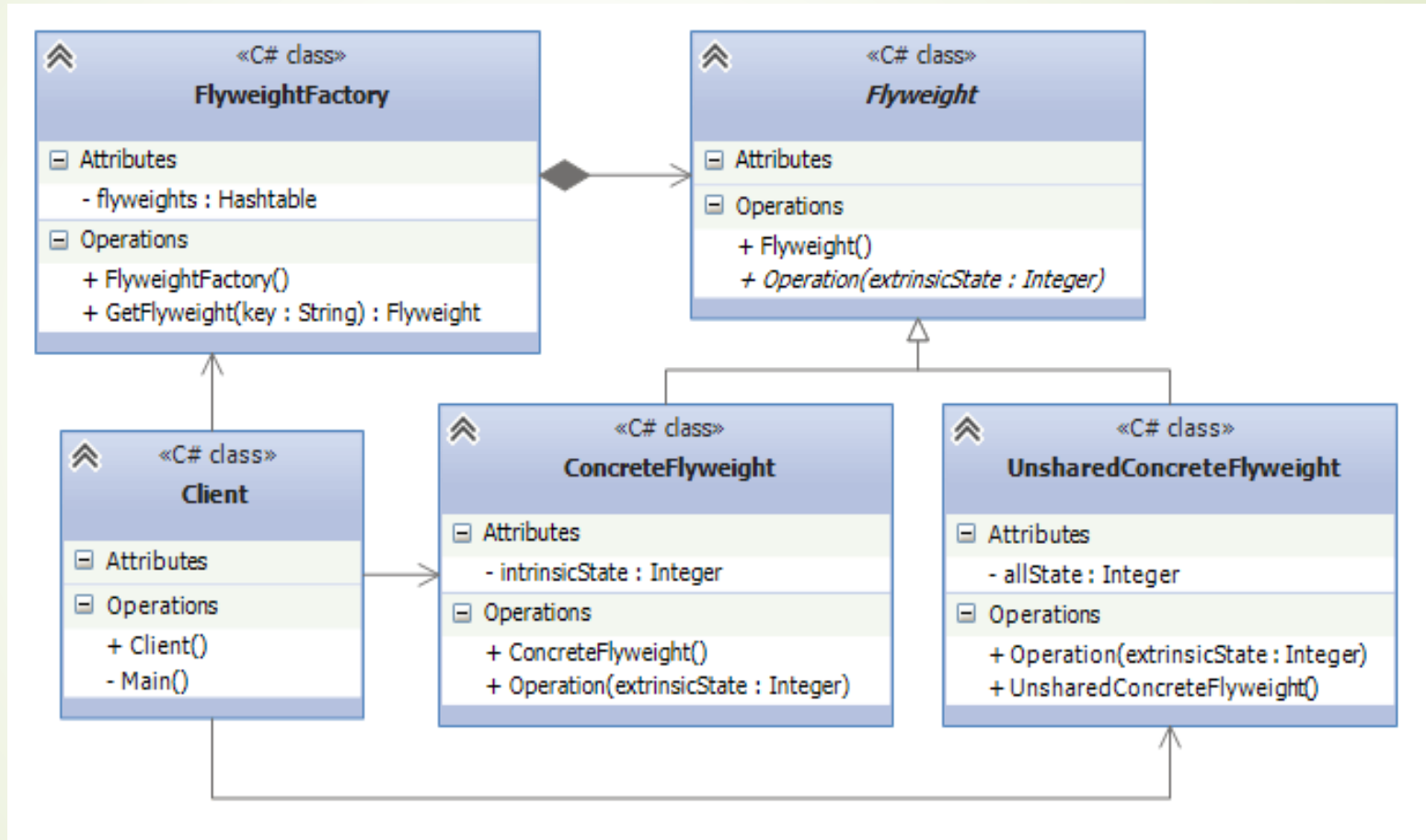




# Flyweight


- Шаблон проектирования, который позволяет использовать разделяемые объекты сразу в нескольких контекстах. Приспособленцы не могут делать предположений о контексте, в котором работают.
- Применим когда:
  - в приложении используется большое число очень схожих экземпляров заданного класса;
  - часть состояния объекта является контекстной и может быть легко вынесена во внешние структуры;
  - после вынесения части состояния, все экземпляры становятся одинаковыми и это дает возможность заменить их одним;
  - приложение не проверяет идентичность объектов, т.к. в этом случае все якобы самостоятельные экземпляры являются одним объектом.





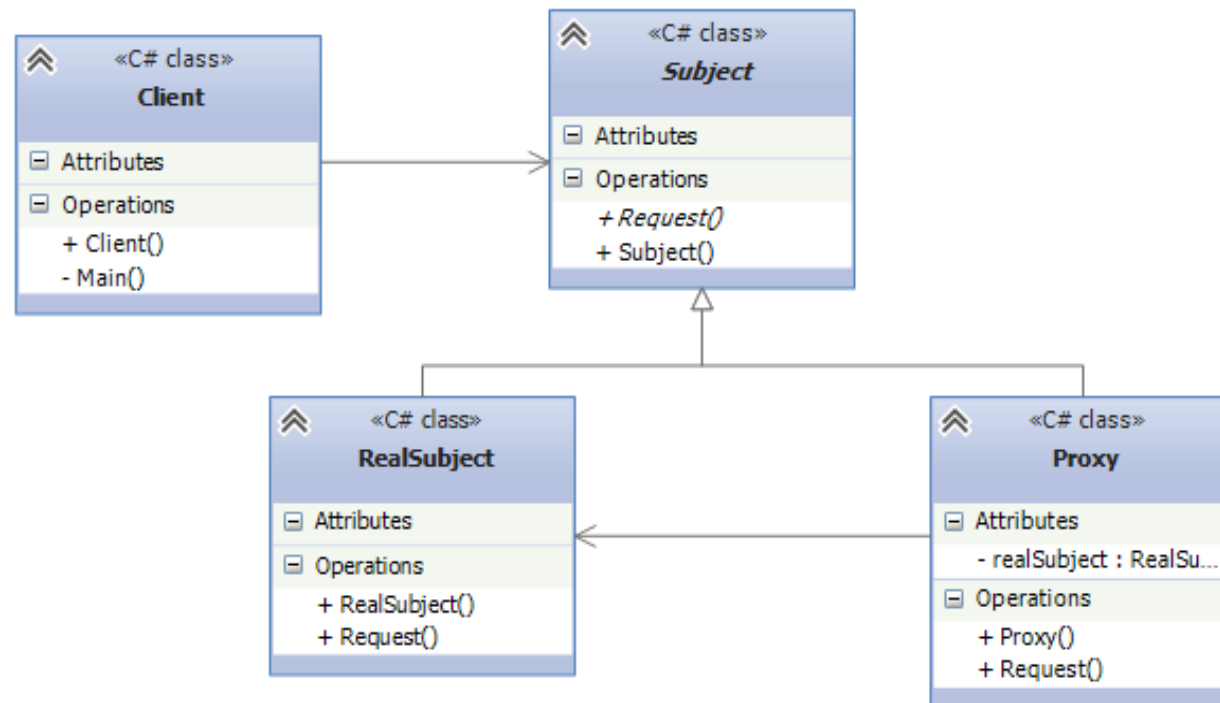


# Особенности

- При использовании приспособленцев не исключены затраты на передачу, поиск или вычисление внутреннего состояния, особенно если раньше оно хранилось как внутреннее. Однако такие расходы с лихвой компенсируются экономией памяти за счет разделения объектов-приспособленцев
  - Паттерн приспособленец часто применяется вместе с компоновщиком для представления иерархической структуры в виде графа с разделяемыми листовыми узлами.
- 

# Proxy

- Прокси является суррогатом другого объекта и контролирует доступ к нему.



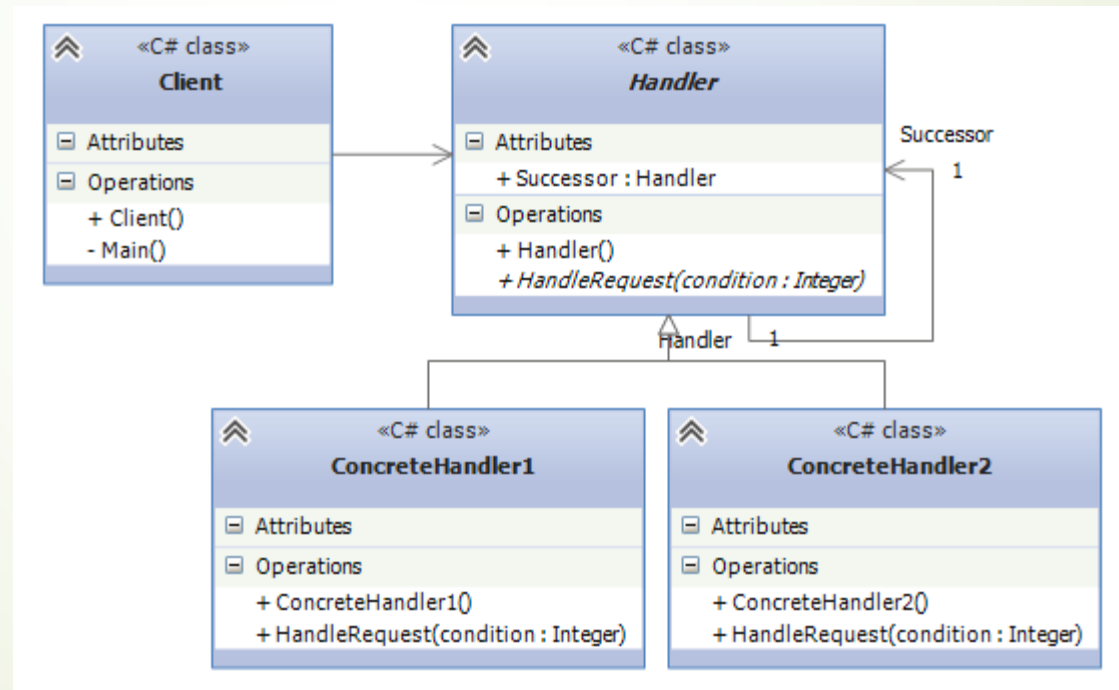


# Паттерны поведения

- Chain of responsibility (Цепочка обязанностей )
- Command (Команда)
- Interpreter (Интерпретатор)
- Iterator (Итератор)
- Mediator (Посредник)
- Memento (Хранитель)
- Observer (Наблюдатель)
- State (Состояние)
- Strategy (Стратегия)
- Template method (Шаблонный метод)
- Visitor (Посетитель)

# Chain of responsibility

- Паттерн 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





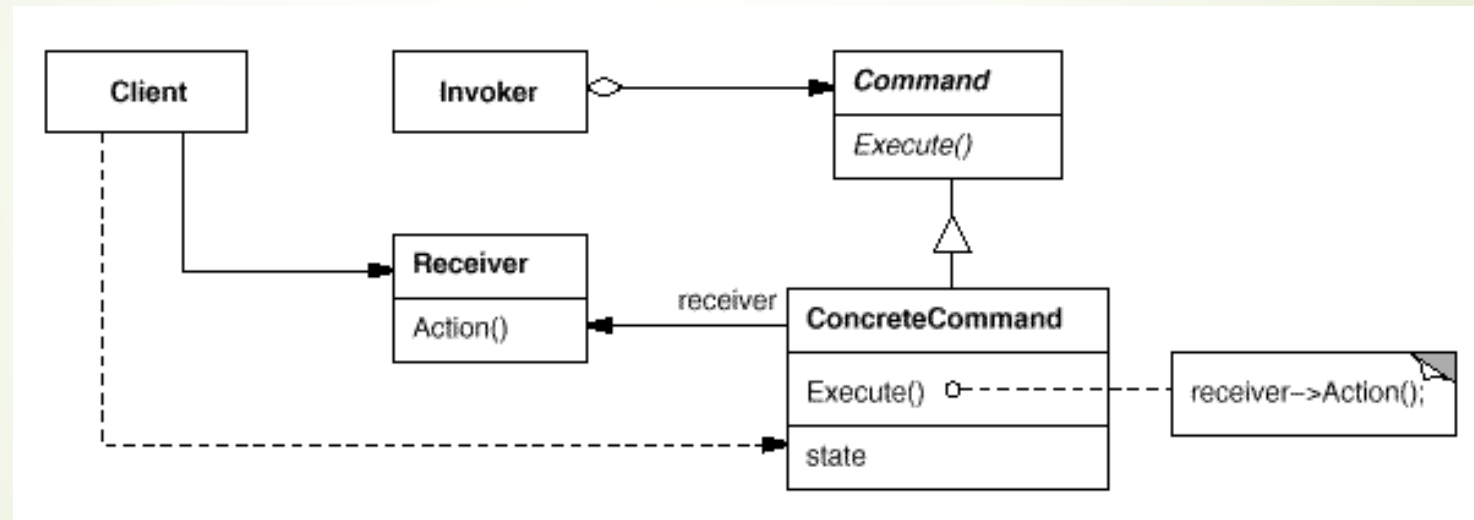


# Особенности

- ослабление связанности. Этот паттерн освобождает объект от необходимости знать, кто конкретно обрабатывает его запрос
- дополнительная гибкость при распределении обязанностей между объектами. Цепочка обязанностей позволяет повисить гибкость распределения обязанностей между объектами. Добавить или изменить обязанности по обработке запроса можно, включив в цепочку новых участников или изменив ее каким-то другим образом.
- получение не гарантировано

# Command

- Инкапсулирует запрос как объект, позволяя тем самым задавать параметры клиентов для обработки соответствующих запросов, ставить запросы в очередь или протоколировать их, а также поддерживать отмену операций



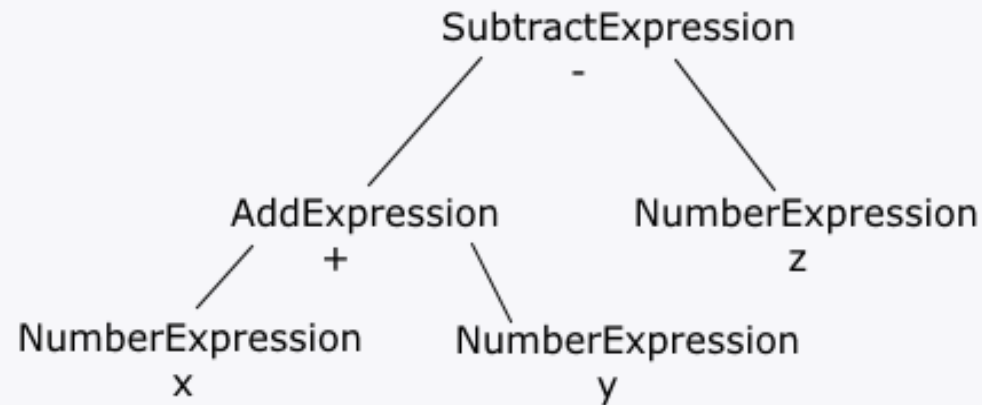


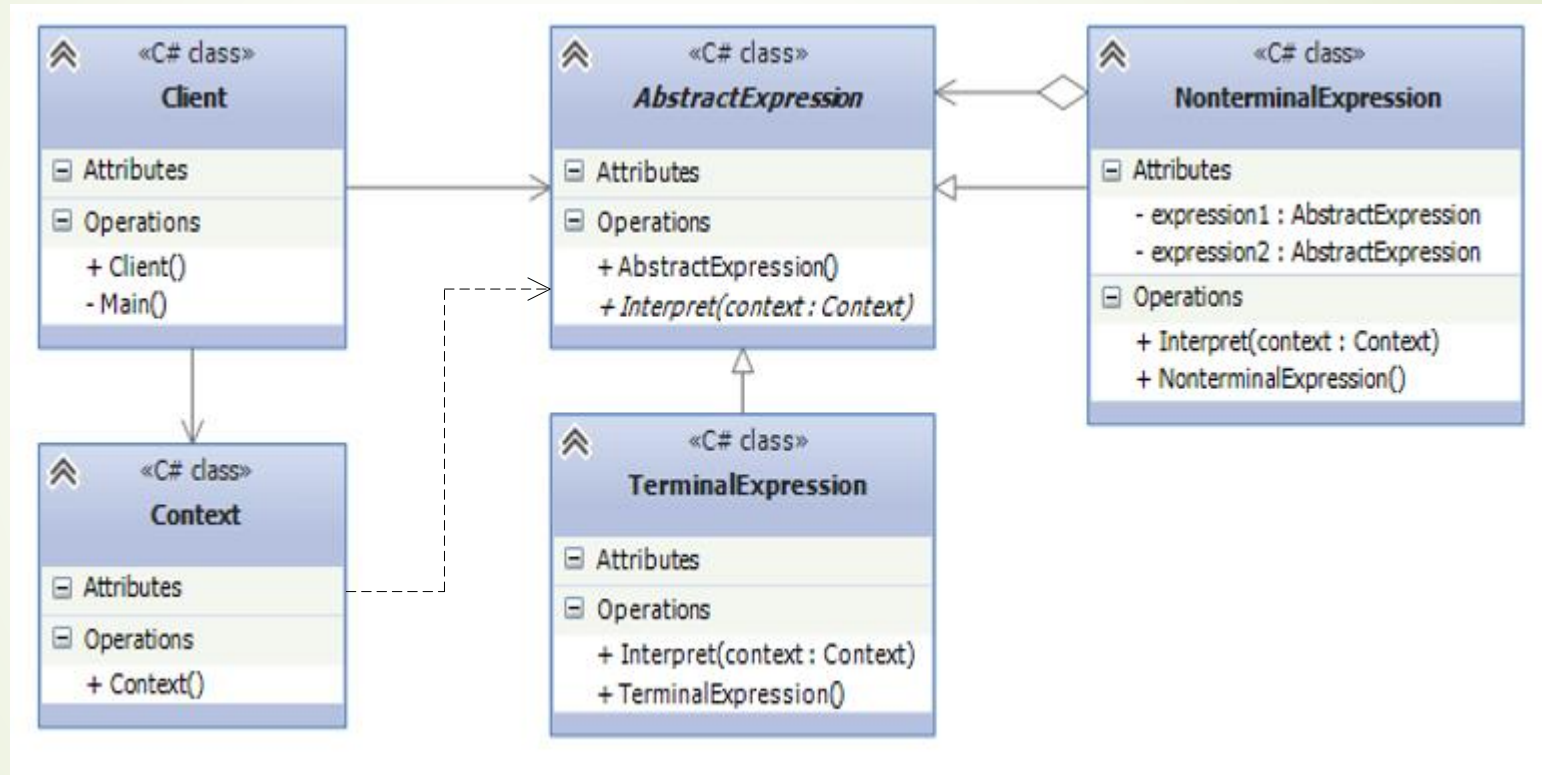
# Особенности

- команда разрывает связь между объектом, инициирующим операцию, и объектом, имеющим информацию о том, как ее выполнить
- команды - это самые настоящие объекты. Допускается манипулировать ими и расширять их точно так же, как в случае с любыми другими объектами
- Из простых команд можно собирать составные (например макросы в текстовых редакторах). В общем случае составные команды описываются паттерном компоновщик
- добавлять новые команды легко, поскольку никакие существующие классы изменять не нужно

# Interpreter

- Паттерн определяет представление грамматики для заданного языка и интерпретатор предложений этого языка. Как правило, данный шаблон проектирования применяется для часто повторяющихся операций.







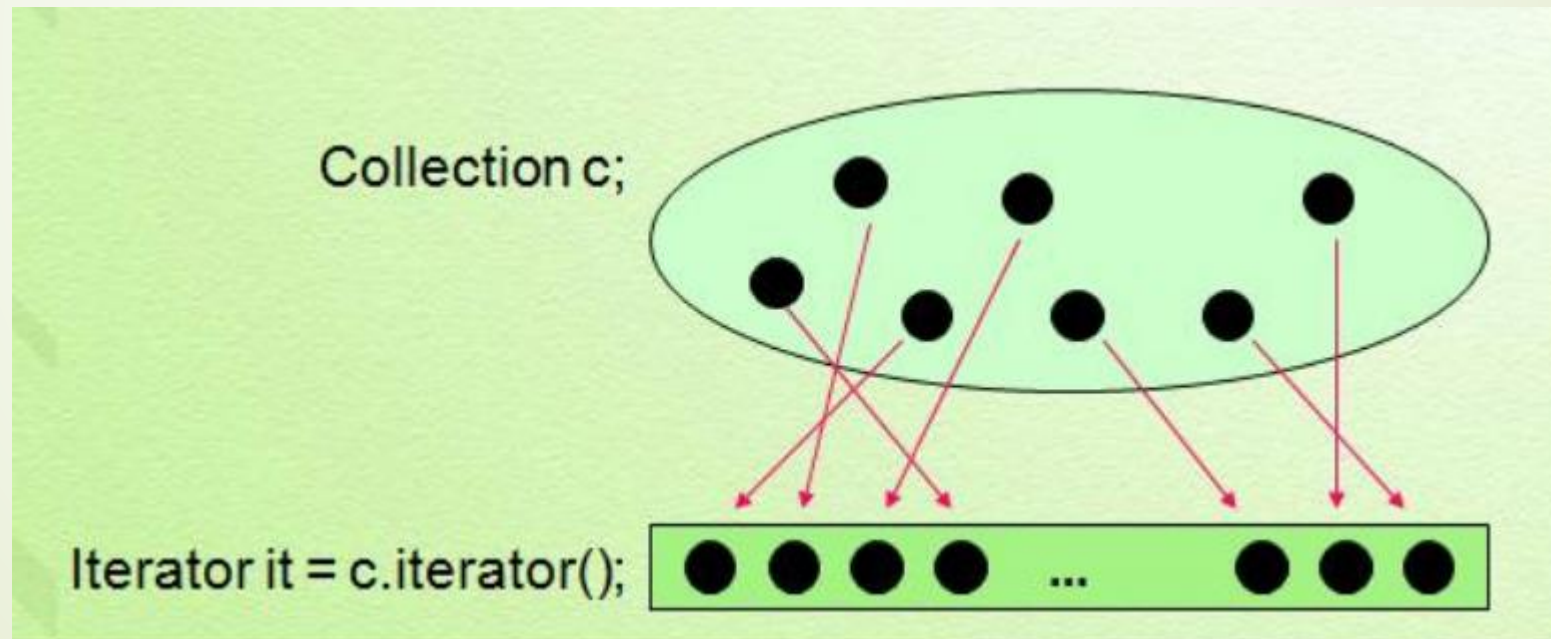


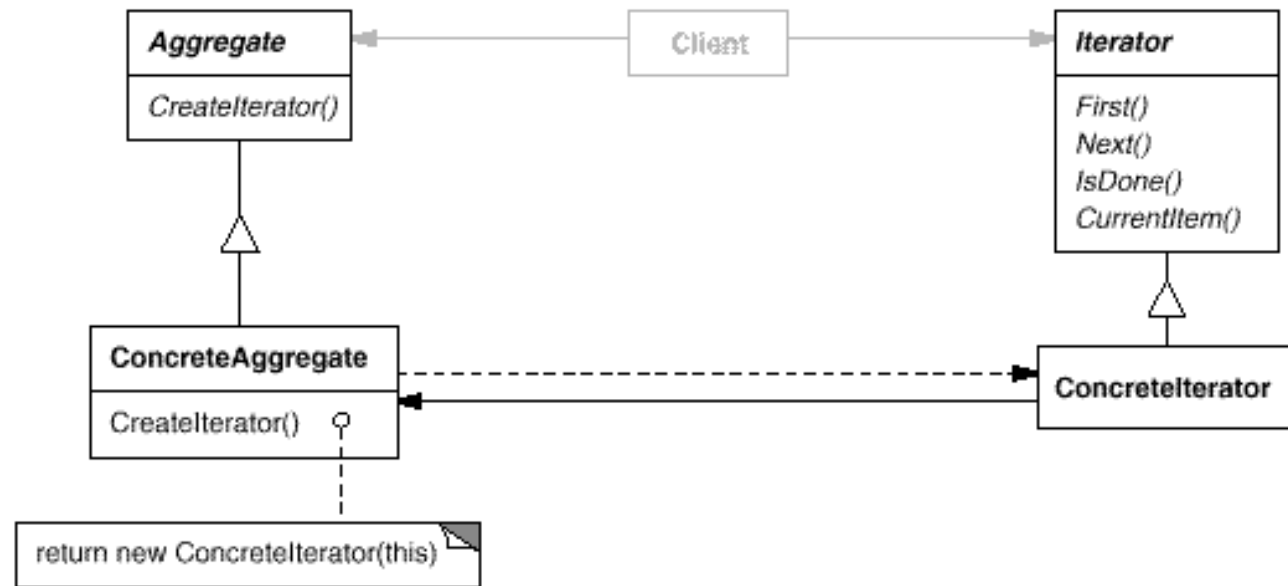
# Особенности

- грамматику легко изменять и расширять
- сложные грамматики трудно сопровождать
- добавление новых способов интерпретации выражений (например, вывод в строковом виде)

# Iterator

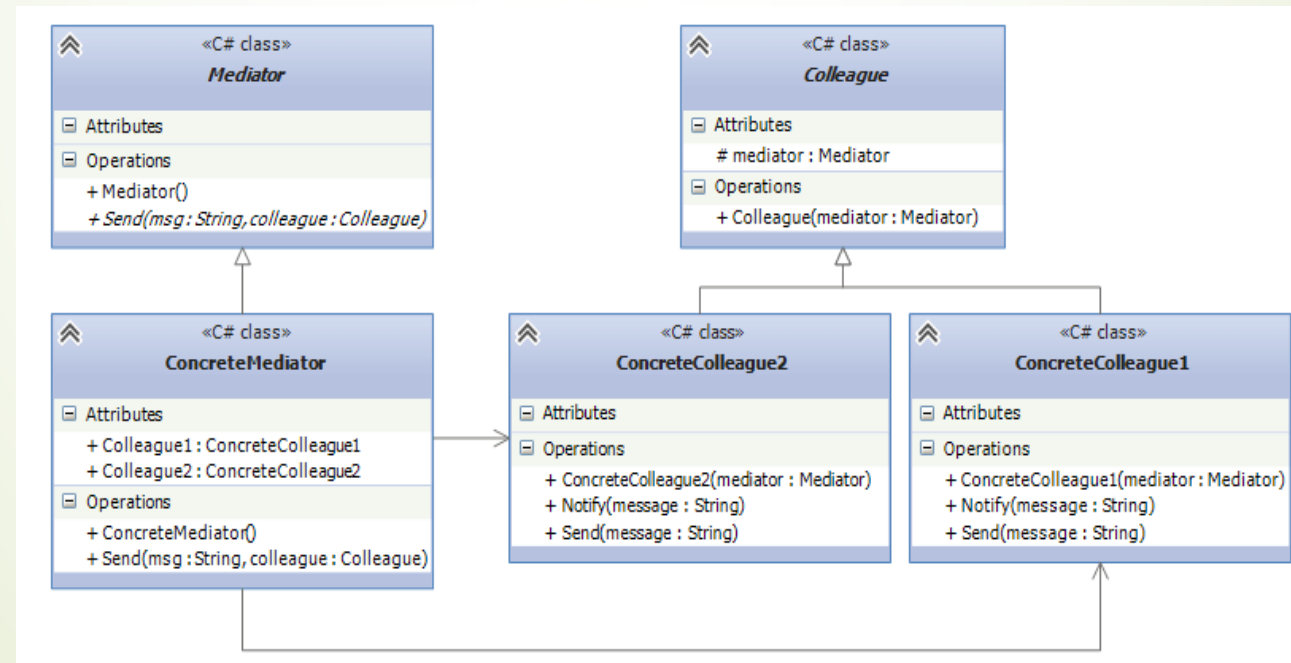
- Предоставляет абстрактный интерфейс для последовательного доступа ко всем элементам составного объекта без раскрытия его внутренней структуры





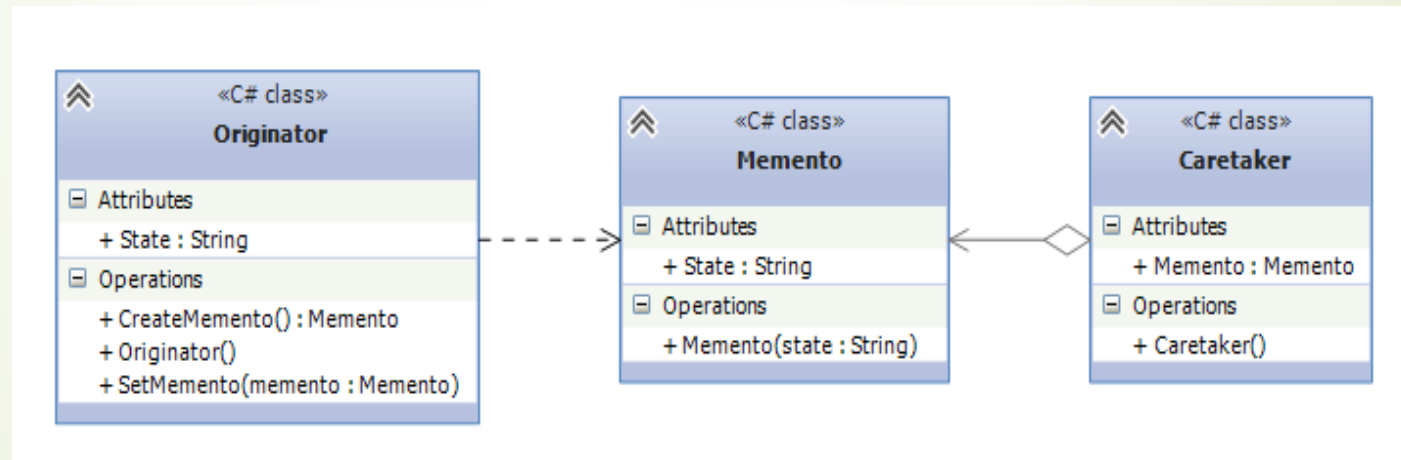
# Mediator

- Определяет объект, инкапсулирующий способ взаимодействия множества объектов. Посредник обеспечивает слабую связанность системы, избавляя объекты от необходимости явно ссылаться друг на друга и позволяя тем самым независимо изменять взаимодействия между ними



# Memento

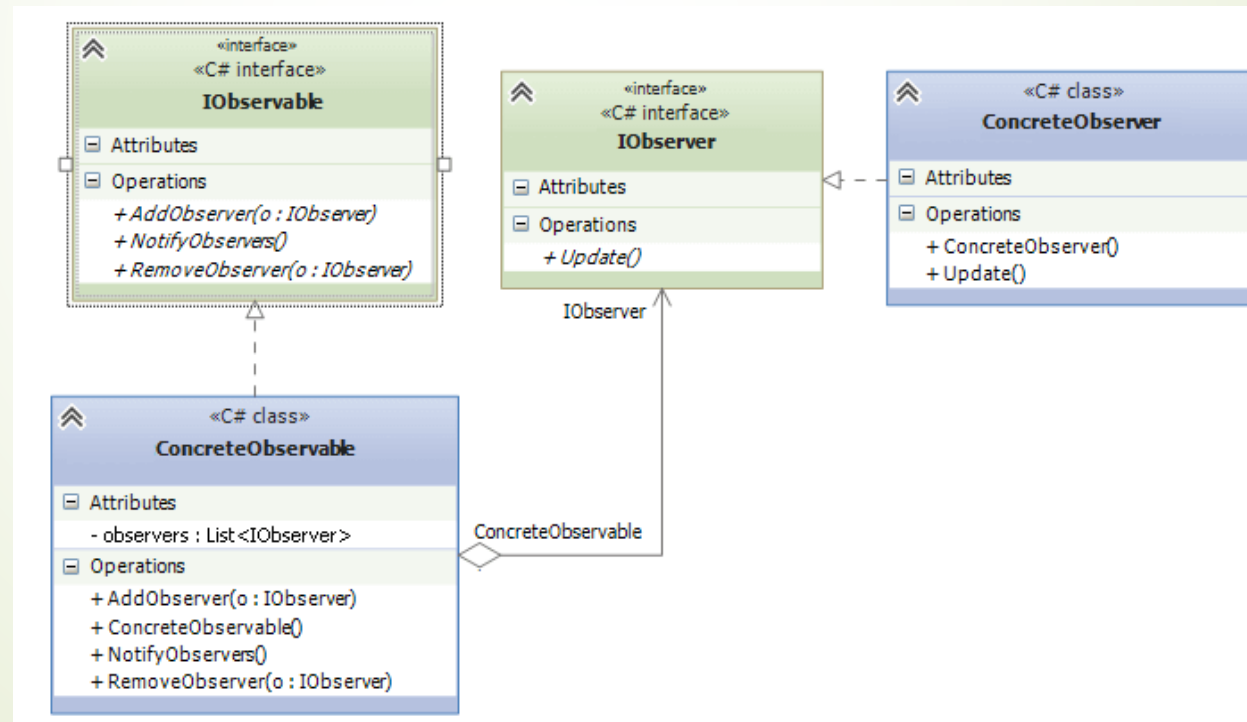
- Позволяет выносить внутреннее состояние объекта за его пределы для последующего возможного восстановления объекта без нарушения принципа инкапсуляции





# Observer

- Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом и автоматически обновляются



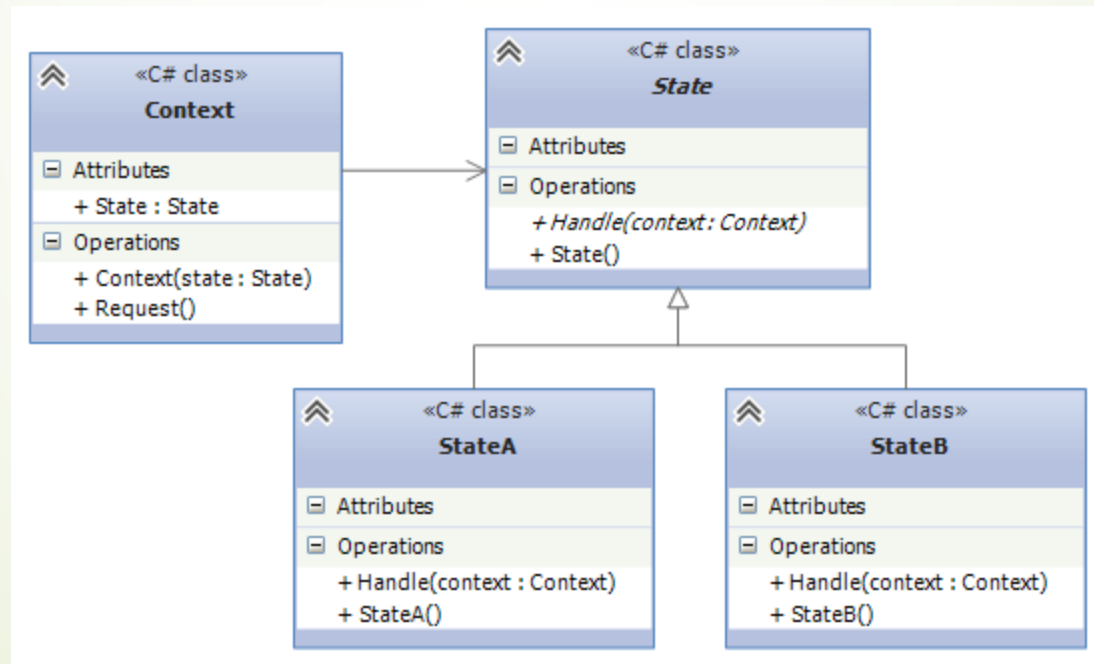


# Особенности

- абстрактная связанность субъекта и наблюдателя. Субъект имеет информацию лишь о том, что у него есть ряд наблюдателей, каждый из которых подчиняется простому интерфейсу абстрактного класса Observer. Субъекту неизвестны конкретные классы наблюдателей. Таким образом, связи между субъектами и наблюдателями носят абстрактный характер и сведены к минимуму.
- поддержка широковещательных коммуникаций. В отличие от обычного запроса для уведомления, посылаемого субъектом, не нужно задавать определенного получателя. Уведомление автоматически поступает всем подписавшимся на него объектам.
- Поскольку наблюдатели не располагают информацией друг о друге, им неизвестно и о том, во что обходится изменение субъекта. Безобидная, на первый взгляд, операция над субъектом может вызвать целый ряд обновлений наблюдателей и зависящих от них объектов.

# State

- позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния. Извне создается впечатление, что изменился класс объекта



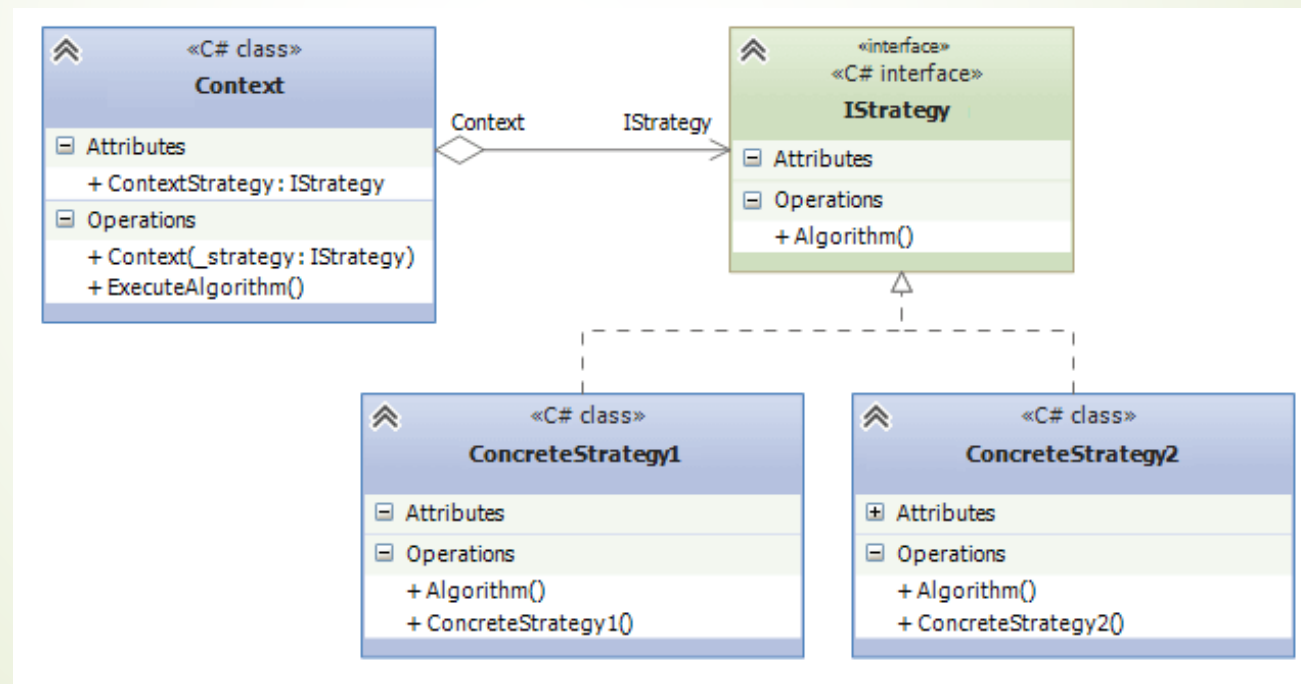


# Особенности

- локализует зависящее от состояния поведение и делит его на части, соответствующие состояниям. Паттерн состояние помещает все поведение, ассоциированное с конкретным состоянием, в отдельный объект. Поскольку зависящий от состояния код целиком находится в одном из подклассов класса *State*, то добавлять новые состояния и переходы можно просто путем порождения новых подклассов
- делает явными переходы между состояниями. Если объект определяет свое текущее состояние исключительно в терминах внутренних данных, то переходы между состояниями не имеют явного представления; они проявляются лишь как присваивания некоторым переменным. Ввод отдельных объектов для различных состояний делает переходы более явными.
- объекты состояния можно разделять. Если в объекте состояния *State* отсутствуют переменные экземпляра, то есть представляемое им состояние кодируется исключительно самим типом, то разные контексты могут разделять один и тот же объект *State*.

# Strategy

- Определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются.





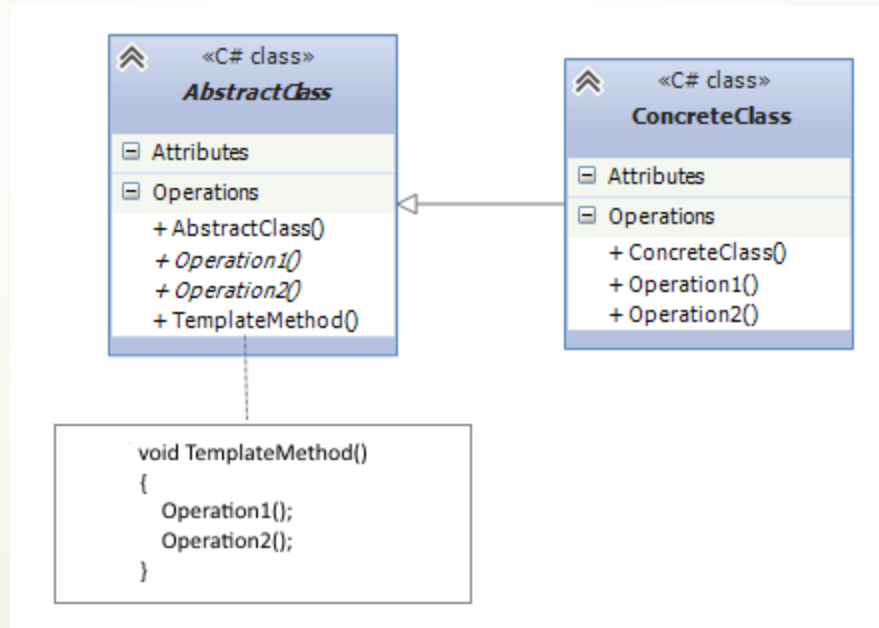
# Особенности

- альтернатива порождению подклассов. При наследовании реализации алгоритма и контекста смешиваются, что затрудняет понимание, сопровождение и расширение контекста. Кроме того, заменить алгоритм динамически уже не удастся.
- с помощью стратегий можно избавиться от условных операторов
- клиенты должны знать о различных стратегиях. Т.е. паттерн стоит применять лишь тогда, когда различия в поведении имеют значение для клиента
- в отдельных случаях контекст создаст и проинициализирует параметры, которые не нужны более простым алгоритмам



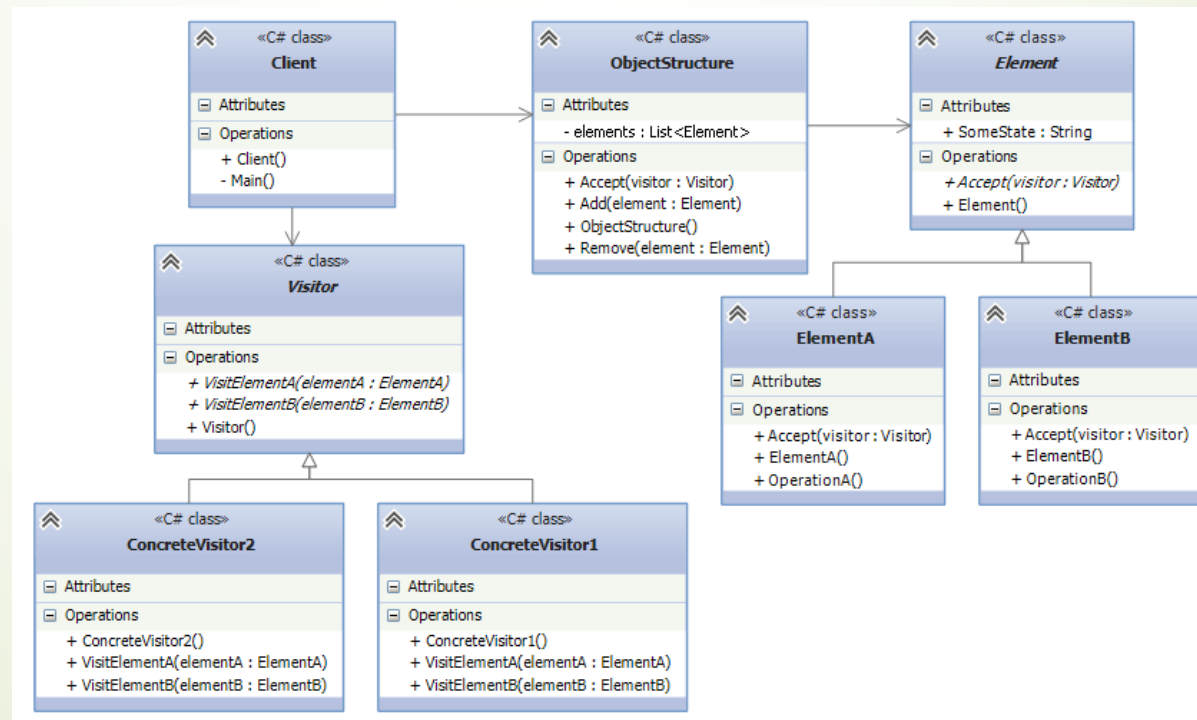
# Template Method

- шаблонный метод определяет основу алгоритма и позволяет подклассам переопределить некоторые шаги алгоритма, не изменяя его структуру в целом



# Visitor

- Описывает операцию, выполняемую с каждым объектом из некоторой структуры. Паттерн посетитель позволяет определить новую операцию, не изменяя классы этих объектов.





# Особенности

- упрощает добавление новых операций. С помощью посетителей легко добавлять операции, зависящие от компонентов сложных объектов. Для определения новой операции над структурой объектов достаточно просто ввести нового посетителя. Напротив, если функциональность распределена по нескольким классам, то для определения новой операции придется изменить каждый класс
- объединяет родственные операции и отсекает те, которые не имеют к ним отношения. Родственное поведение не разносится по всем классам, присутствующим в структуре объектов, оно локализовано в посетителе
- добавление новых классов ConcreteElement затруднено
- применение посетителей подразумевает, что у класса ConcreteElement достаточно развитый интерфейс для того, чтобы посетители могли справиться со своей работой. Поэтому при использовании данного паттерна приходится предоставлять открытые операции для доступа к внутреннему состоянию элементов, что ставит под угрозу инкапсуляцию