Паттерны проектирования

Что такое Паттерн?

Паттерн проектирования — это часто встречающееся 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 какой-то конкретный код, а общую концепцию решения той или иной проблемы, которую нужно будет ещё подстроить под нужды вашей программы.
- Паттерны часто путают с алгоритмами, ведь оба понятия описывают типовые решения каких-то известных проблем. Но если алгоритм — это чёткий набор действий, то паттерн — это высокоуровневое описание решения, реализация которого может отличаться в двух разных программах.
- Если привести аналогии, то алгоритм — это кулинарный рецепт с чёткими шагами, а паттерн — инженерный чертёж, на котором нарисовано решение, но не конкретные шаги его реализации.

Из чего состоит паттерн?

Описания паттернов обычно очень формальны и чаще всего состоят из таких пунктов:

- проблема, которую решает паттерн;
- мотивации к решению проблемы способом, который предлагает паттерн;
- структуры классов, составляющих решение;
- примера на одном из языков программирования;
- особенностей реализации в различных контекстах;
- связей с другими паттернами.

Классификация паттернов

- Самые низкоуровневые и простые паттерны — *идиомы*. Они не универсальны, поскольку применимы только в рамках одного языка программирования.
- Самые универсальные — *архитектурные паттерны*, которые можно реализовать практически на любом языке. Они нужны для проектирования всей программы, а не отдельных её элементов.

Классификация паттернов

Паттерны также можно классифицировать по предназначению:

- **Порождающие (creational)** беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей.
- **Структурные (structural)** показывают различные способы построения связей между объектами.
- **Поведенческие (behavioral)** заботятся об эффективной коммуникации между объектами.

Порождающие паттерны

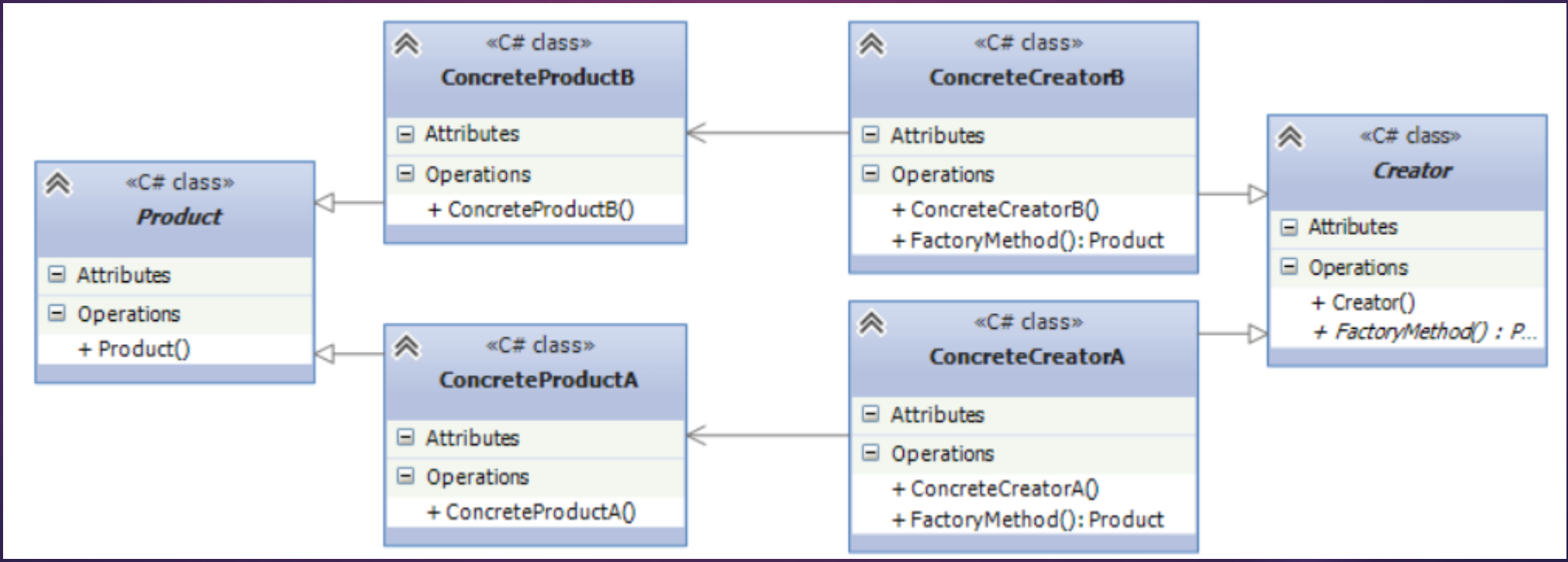
- Factory Method – Фабричный метод
- Abstract Factory – Абстрактная фабрика
- Builder – Строитель
- Prototype – Прототип
- Singleton – Одиночка

Эти паттерны отвечают за удобное и безопасное создание новых объектов или даже целых семейств объектов.

Фабричный метод

Фабричный метод — это порождающий паттерн проектирования, который определяет общий интерфейс для создания объектов в суперклассе, позволяя подклассам изменять тип создаваемых объектов.

На языке UML паттерн можно описать следующим образом:



Фабричный метод

Когда надо применять паттерн

- Когда заранее неизвестно, объекты каких типов необходимо создавать
- Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать.
- Когда создание новых объектов необходимо делегировать из базового класса классам наследникам

Фабричный метод

Преимущества данного паттерна проектирования:

- Избавляет класс от привязки к конкретным классам продуктов.
- Выделяет код производства продуктов в одно место, упрощая поддержку кода.
- Упрощает добавление новых продуктов в программу.
- Реализует *принцип открытости/закрытости*.

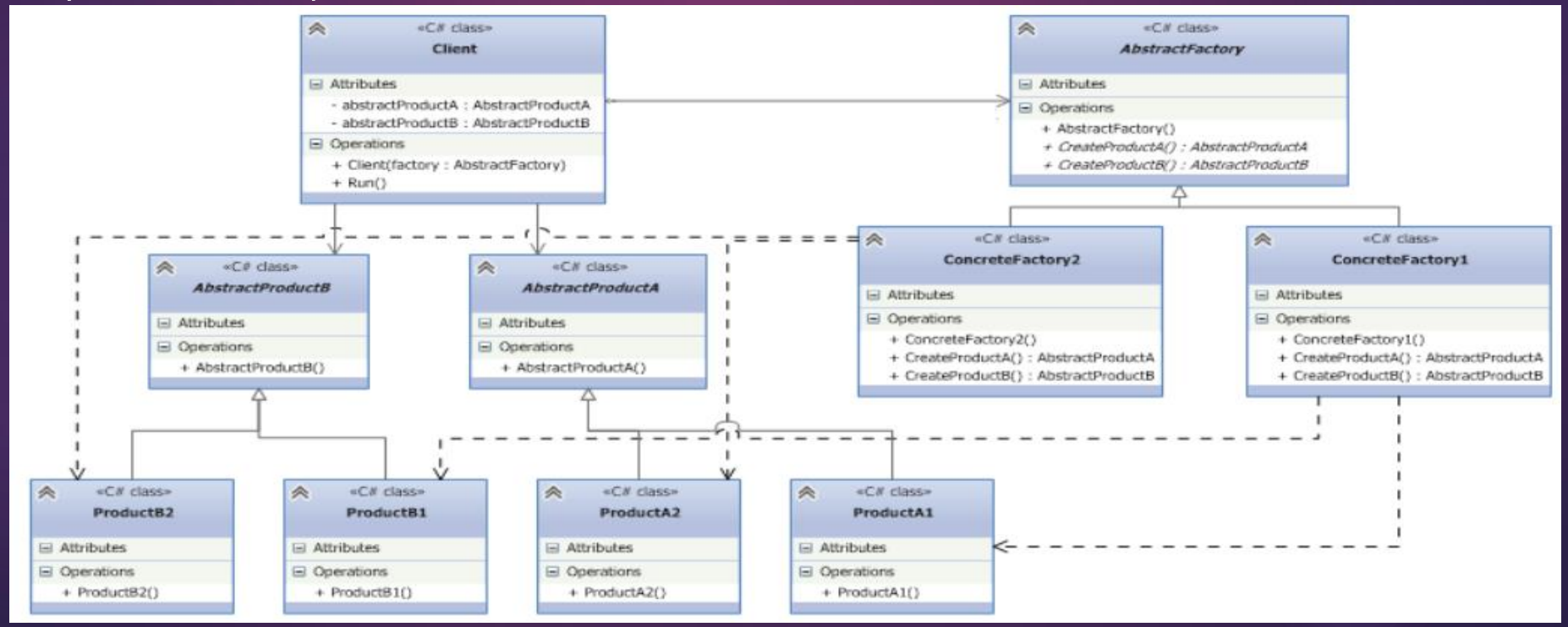
Из недостатков можно выделить:

- Может привести к созданию больших параллельных иерархий классов, так как для каждого класса продукта надо создать свой подкласс создателя.

Абстрактная фабрика

Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

С помощью UML абстрактную фабрику можно представить следующим образом:



Абстрактная фабрика

Когда использовать абстрактную фабрику

- Когда система не должна зависеть от способа создания и компоновки новых объектов
- Когда создаваемые объекты должны использоваться вместе и являются взаимосвязанными

Абстрактная фабрика

Преимущества:

- Гарантирует сочетаемость создаваемых продуктов.
- Избавляет клиентский код от привязки к конкретным классам продуктов.
- Выделяет код производства продуктов в одно место, упрощая поддержку кода.
- Упрощает добавление новых продуктов в программу.
- Реализует *принцип открытости/закрытости*.

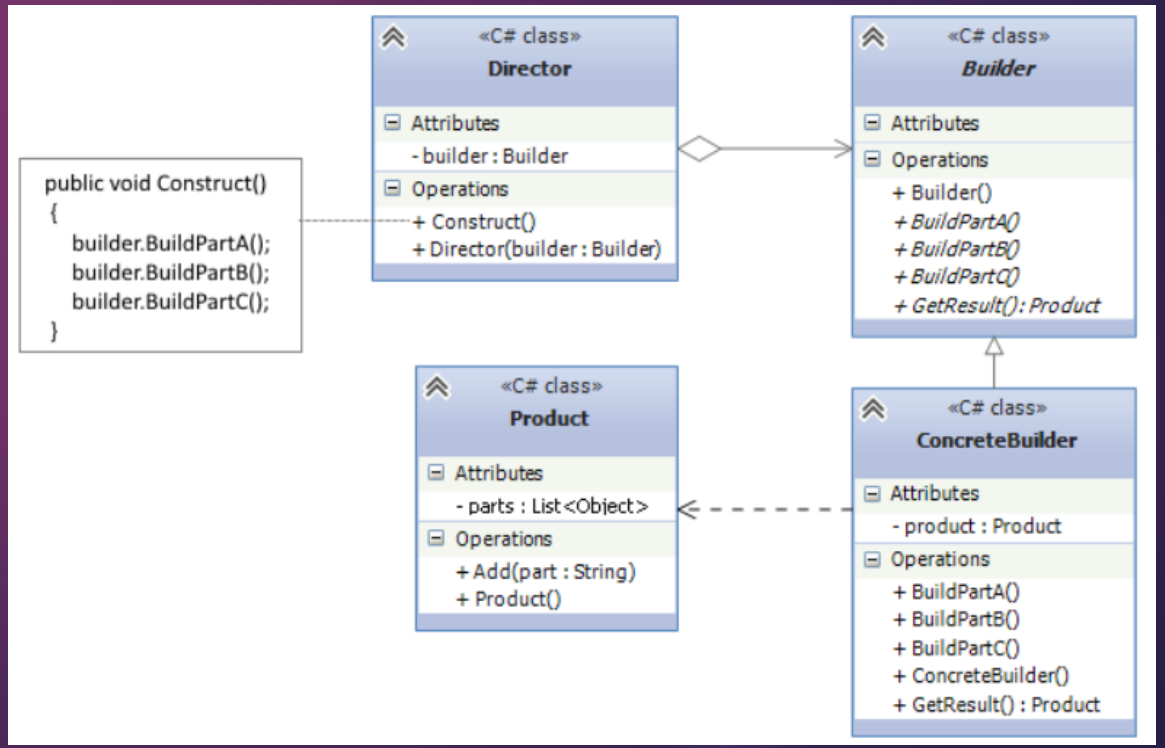
Недостатки:

- Усложняет код программы из-за введения множества дополнительных классов.
- Требует наличия всех типов продуктов в каждой вариации.

Строитель

Строитель — это порождающий паттерн проектирования, который позволяет создавать сложные объекты пошагово. Строитель даёт возможность использовать один и тот же код строительства для получения разных представлений объектов.

Формально в UML паттерн мог бы выглядеть следующим образом:



Строитель

Когда использовать паттерн Строитель?

- Когда процесс создания нового объекта не должен зависеть от того, из каких частей этот объект состоит и как эти части связаны между собой
- Когда необходимо обеспечить получение различных вариаций объекта в процессе его создания

Строитель

Преимущества:

- Позволяет создавать продукты пошагово.
- Позволяет использовать один и тот же код для создания различных продуктов.
- Изолирует сложный код сборки продукта от его основной бизнес-логики.

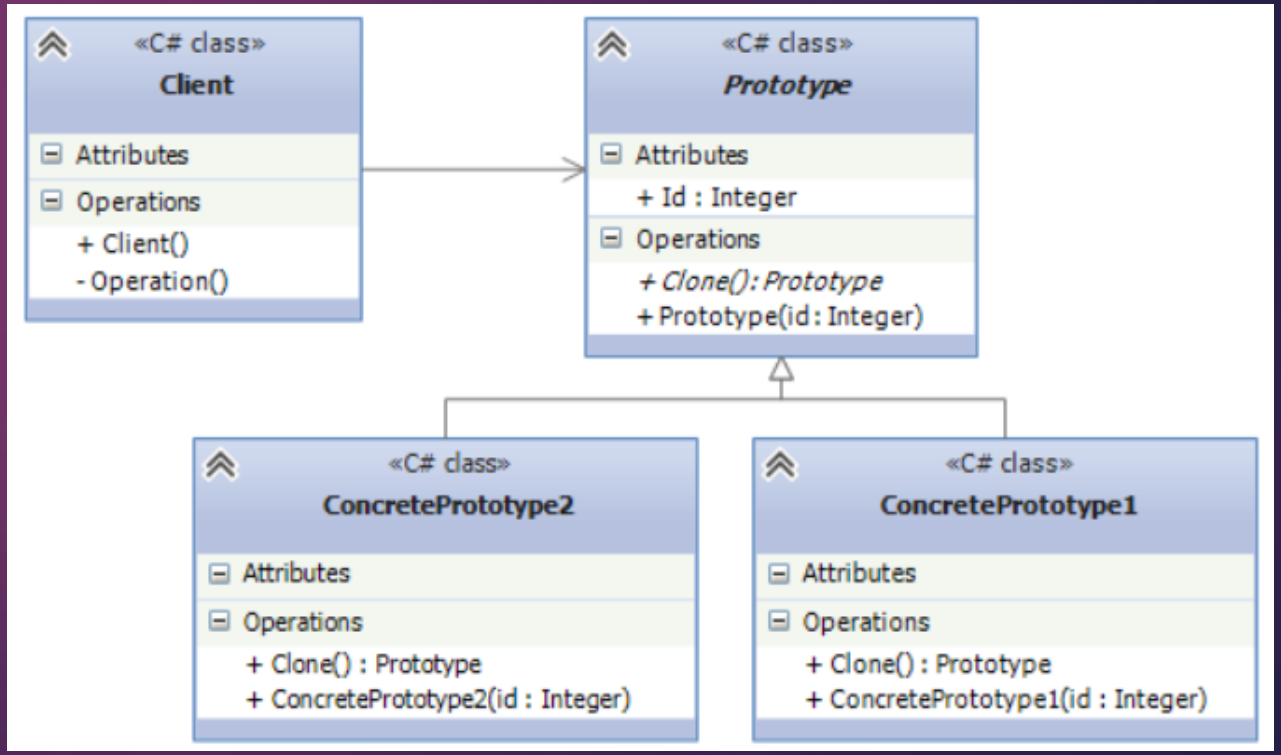
Недостатки:

- Усложняет код программы из-за введения дополнительных классов.
- Клиент будет привязан к конкретным классам строителей, так как в интерфейсе строителя может не быть метода получения результата.

Прототип

Прототип — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

На языке UML отношения между классами при применении данного паттерна можно описать следующим образом:



Прототип

Когда использовать Прототип?

- Когда конкретный тип создаваемого объекта должен определяться динамически во время выполнения
- Когда нежелательно создание отдельной иерархии классов фабрик для создания объектов-продуктов из параллельной иерархии классов (как это делается, например, при использовании паттерна Абстрактная фабрика)
- Когда клонирование объекта является более предпочтительным вариантом нежели его создание и инициализация с помощью конструктора. Особенно когда известно, что объект может принимать небольшое ограниченное число возможных состояний.

Прототип

Преимущества:

- Позволяет клонировать объекты, не привязываясь к их конкретным классам.
- Меньше повторяющегося кода инициализации объектов.
- Ускоряет создание объектов.
- Альтернатива созданию подклассов для конструирования сложных объектов.

Из недостатков можно выделить:

- Сложно клонировать составные объекты, имеющие ссылки на другие объекты.

Одиночка

Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа.

Когда надо использовать Singleton? Когда необходимо, чтобы для класса существовал только один экземпляр

Singleton позволяет создать объект только при его необходимости. Если объект не нужен, то он не будет создан. В этом отличие синглтона от глобальных переменных.

Одиночка

Преимущества:

- Гарантирует наличие единственного экземпляра класса.
- Предоставляет к нему глобальную точку доступа.
- Реализует отложенную инициализацию объекта-одиночки.

Недостатки:

- Нарушает *принцип единственной ответственности класса*.
- Маскирует плохой дизайн.
- Проблемы мультипоточности.
- Требует постоянного создания Mock-объектов при юнит-тестировании.

Структурные паттерны

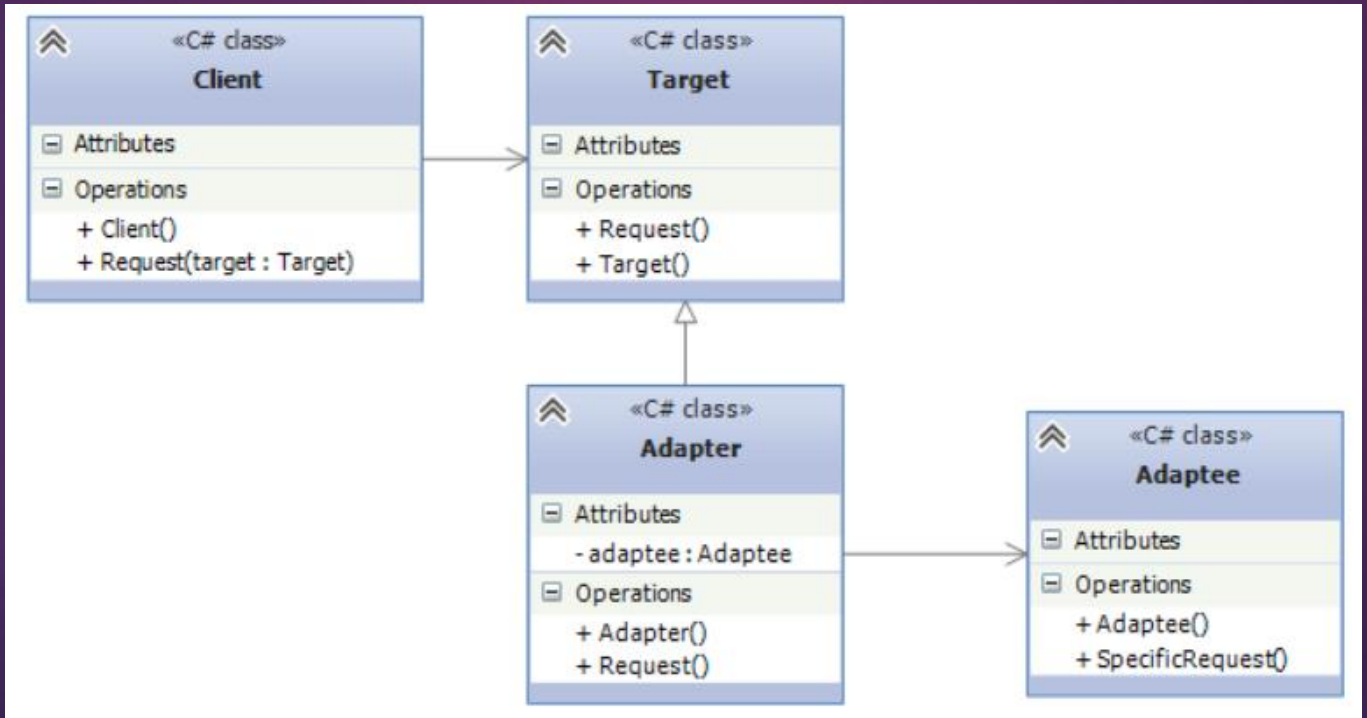
- Adapter – Адаптер
- Composite – Компоновщик
- Proxy – Заместитель
- Flyweight – Легковес
- Facade – Фасад
- Bridge – Мост
- Decorator – Декоратор

Эти паттерны отвечают за построение удобных в поддержке иерархий классов.

Адаптер

Адаптер — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Формальное определение паттерна на UML выглядит следующим образом:



Адаптер

Когда надо использовать Адаптер?

- Когда необходимо использовать имеющийся класс, но его интерфейс не соответствует потребностям
- Когда надо использовать уже существующий класс совместно с другими классами, интерфейсы которых не совместимы

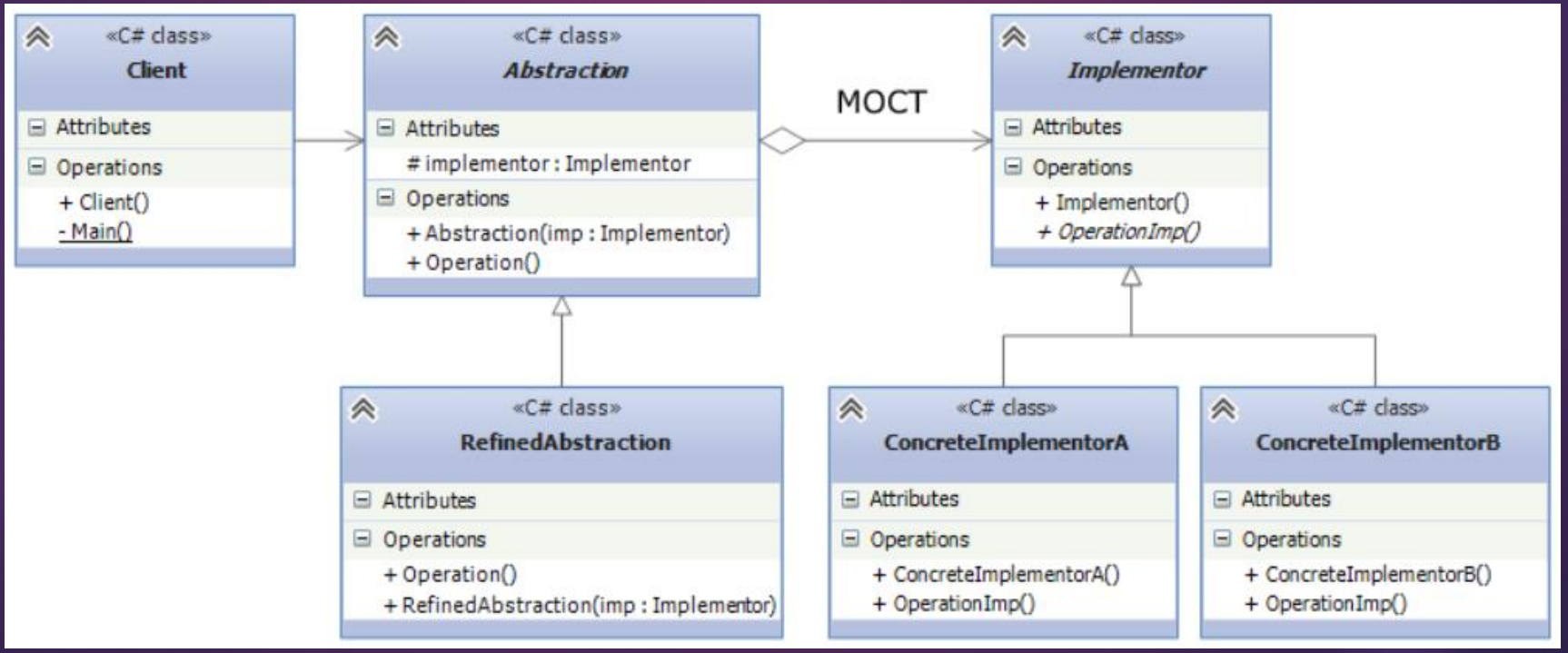
Преимущества и недостатки:

- Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.
- Усложняет код программы из-за введения дополнительных классов.

Мост

Мост — это структурный паттерн проектирования, который разделяет один или несколько классов на две отдельные иерархии — абстракцию и реализацию, позволяя изменять их независимо друг от друга.

Представление паттерна с помощью UML:



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

Мост

Преимущества:

- Позволяет строить платформо-независимые программы.
- Скрывает лишние или опасные детали реализации от клиентского кода.
- Реализует *принцип открытости/закрытости*.

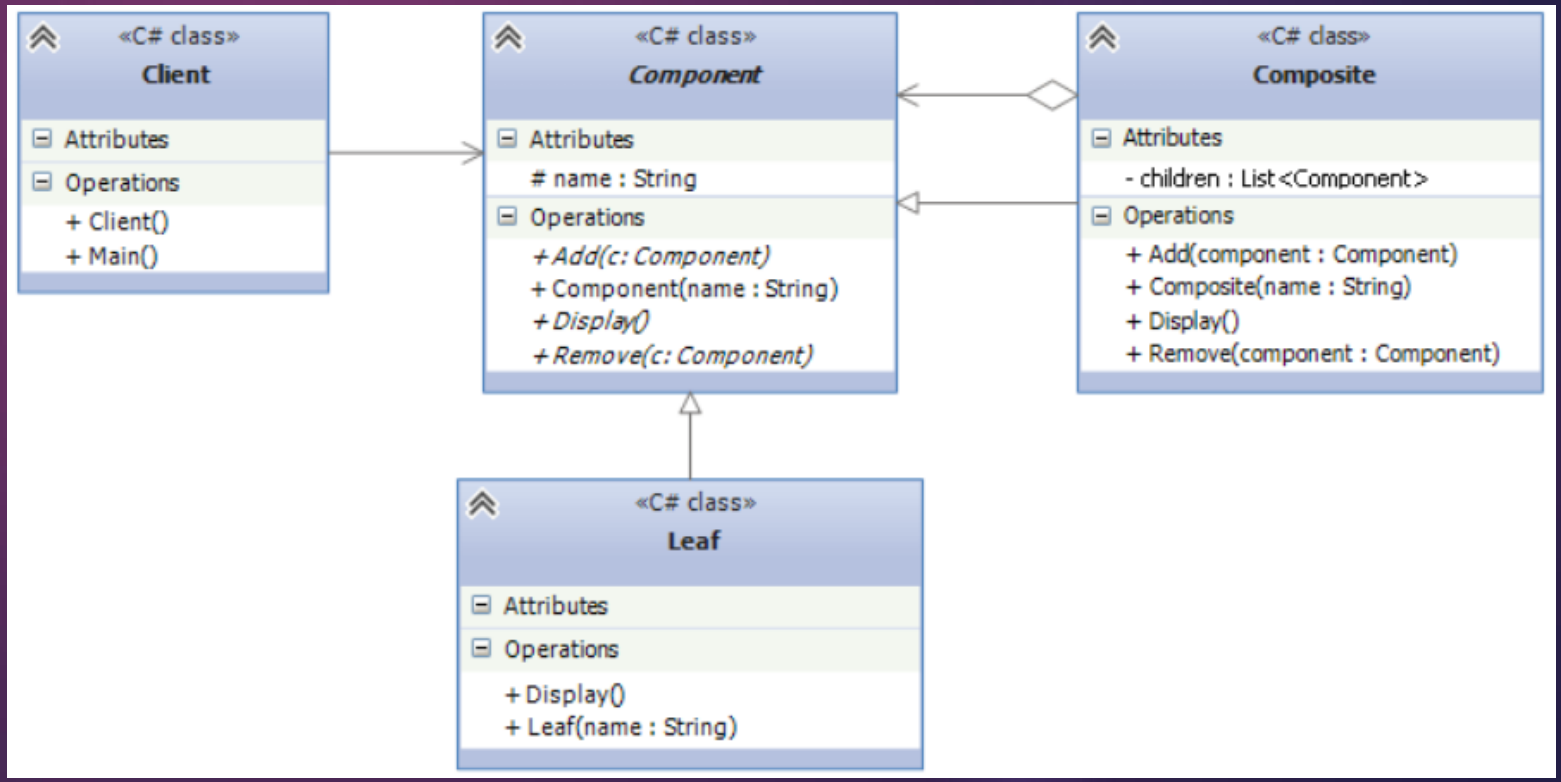
Из недостатков можно выделить только:

- Усложняет код программы из-за введения дополнительных классов.

Компоновщик

Компоновщик — это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.

С помощью UML шаблон компоновщик можно представить следующим образом:



Компоновщик

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

Компоновщик

Преимущества:

- Упрощает архитектуру клиента при работе со сложным деревом компонентов.
- Облегчает добавление новых видов компонентов.

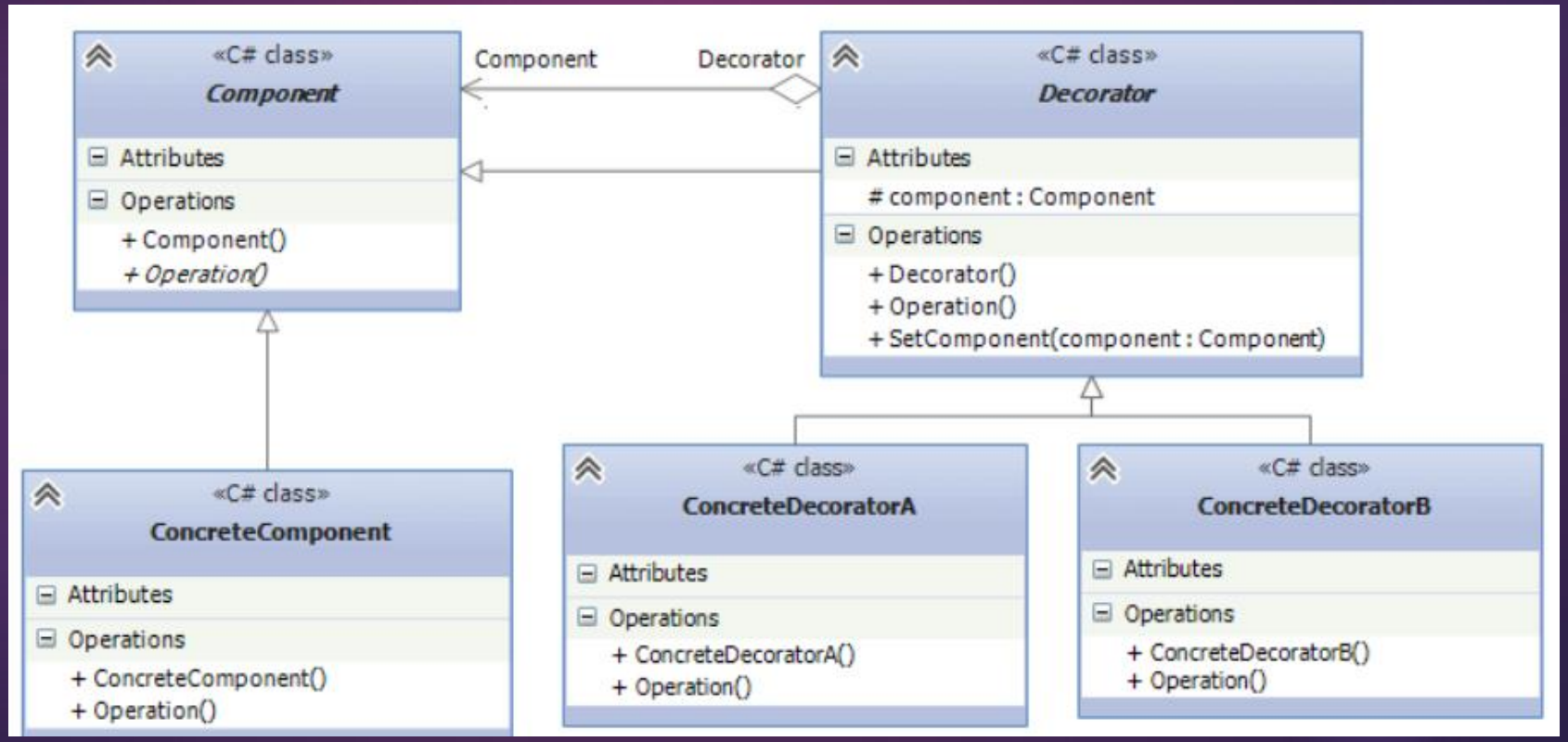
Недостаток:

- Создаёт слишком общий дизайн классов.

Декоратор

Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».

Схематически шаблон "Декоратор" можно выразить следующим образом:



Декоратор

Когда следует использовать декораторы?

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

Преимущества:

- Большая гибкость, чем у наследования.
- Позволяет добавлять обязанности на лету.
- Можно добавлять несколько новых обязанностей сразу.
- Позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.

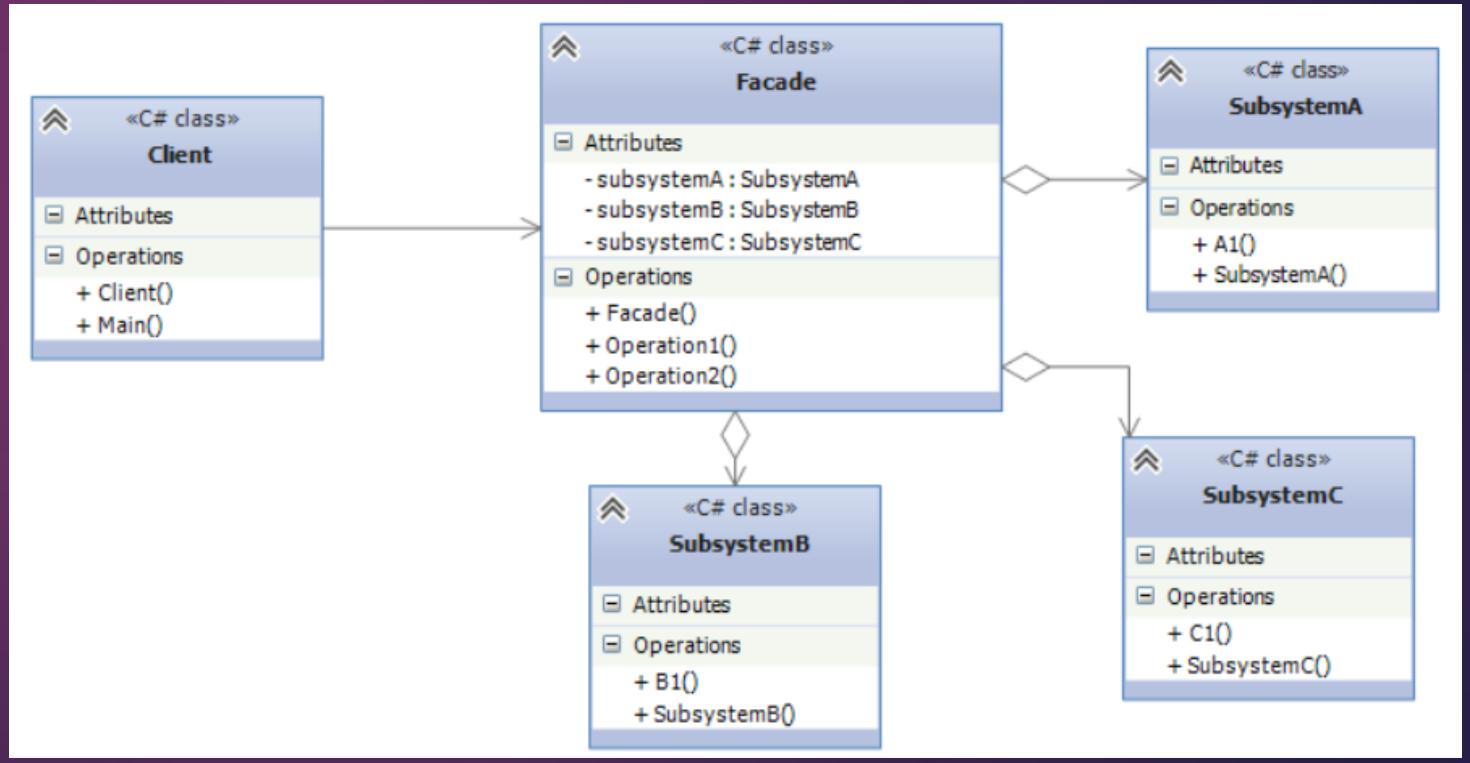
Недостатки:

- Трудно конфигурировать многократно обёрнутые объекты.
- Обилие крошечных классов.

Фасад

Фасад — это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку.

В UML общую схему фасада можно представить следующим образом:



Фасад

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.

Фасад

Преимущества и недостатки:

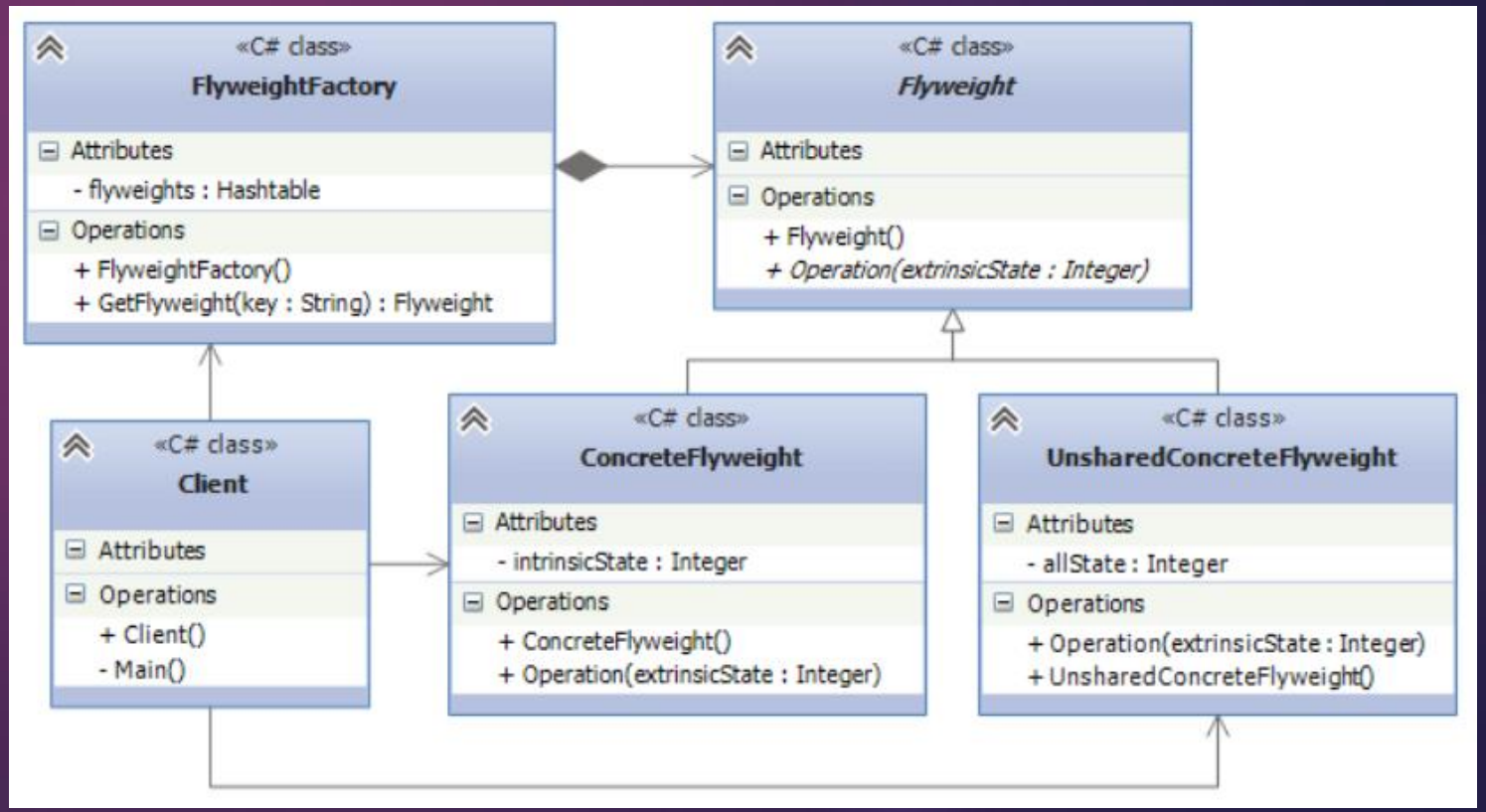
- Изолирует клиентов от компонентов сложной подсистемы.
- Фасад рискует стать **божественным объектом**, привязанным ко всем классам программы.

Божественный объект — антипаттерн объектно-ориентированного программирования, описывающий объект, который хранит в себе «слишком много» или делает «слишком много».

Легковес

Легковес — это структурный паттерн проектирования, который позволяет вместить большее количество объектов в отведённую оперативную память. Легковес экономит память, разделяя общее состояние объектов между собой, вместо хранения одинаковых данных в каждом объекте.

Отношения в данном паттерне можно описать следующей схемой:



Легковес

Паттерн Легковес следует применять при соблюдении всех следующих условий:

- Когда приложение использует большое количество однообразных объектов, из-за чего происходит выделение большого количества памяти
- Когда часть состояния объекта, которое является изменяемым, можно вынести во вне. Вынесение внешнего состояния позволяет заменить множество объектов небольшой группой общих разделяемых объектов.

Легковес

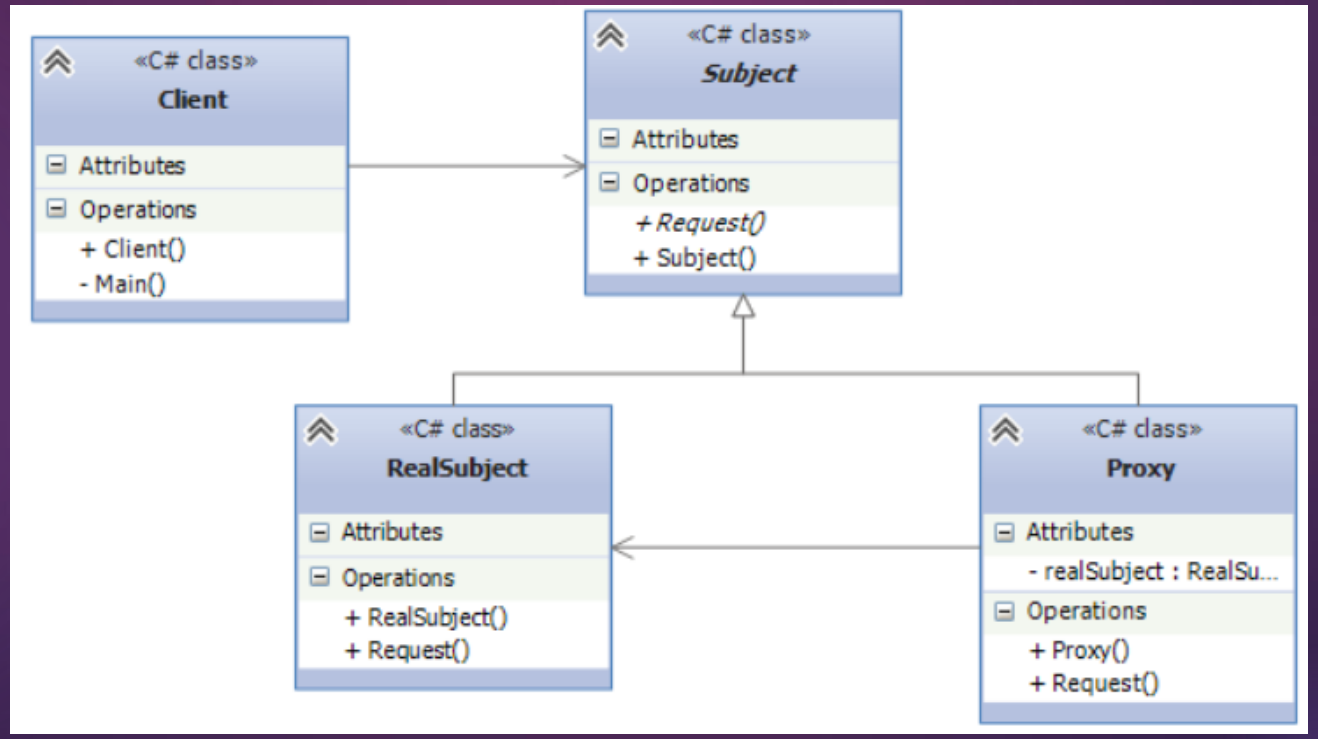
Преимущества и недостатки

- Экономит оперативную память.
- Расходует процессорное время на поиск/вычисление контекста.
- Усложняет код программы из-за введения множества дополнительных классов.

Заместитель

Заместитель — это структурный паттерн проектирования, который позволяет подставлять вместо реальных объектов специальные объекты-заменители. Эти объекты перехватывают вызовы к оригинальному объекту, позволяя сделать что-то *до* или *после* передачи вызова оригиналу.

С помощью UML паттерн может быть описан так:



Заместитель

- **Когда использовать прокси?**
- 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)**

Заместитель

Преимущества:

- Позволяет контролировать сервисный объект незаметно для клиента.
- Может работать, даже если сервисный объект ещё не создан.
- Может контролировать жизненный цикл служебного объекта.

Недостатки:

- Усложняет код программы из-за введения дополнительных классов.
- Увеличивает время отклика от сервиса.

Поведенческие паттерны

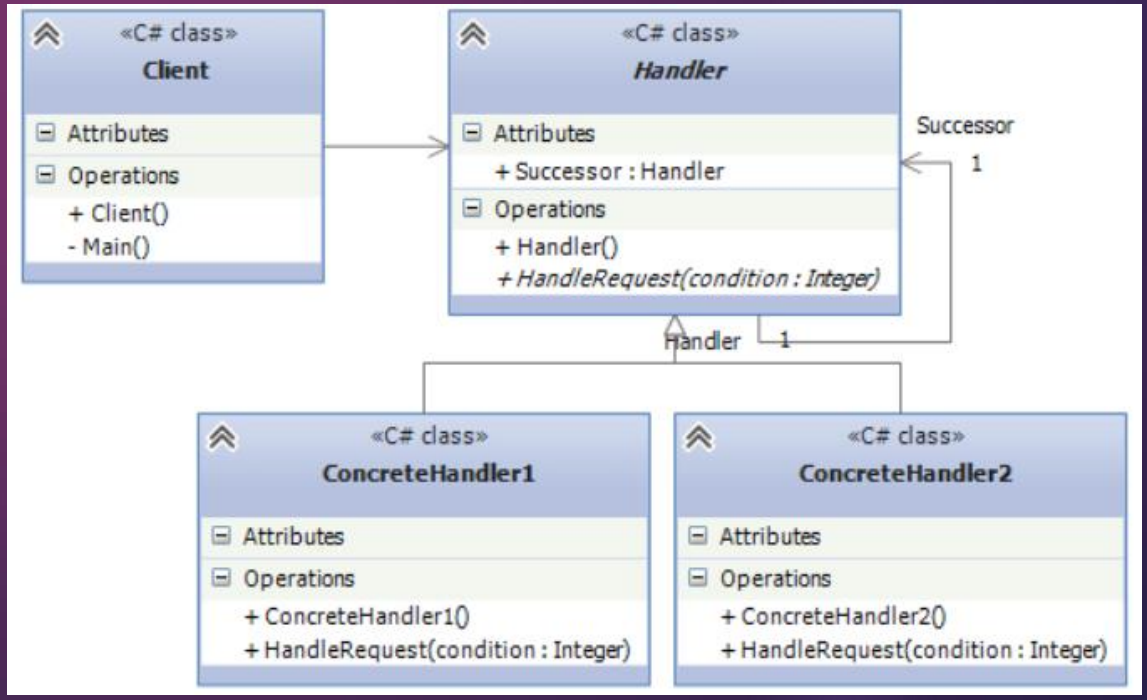
- Template Method – Шаблонный метод
- Mediator – Посредник
- Chain of Responsibility – Цепочка обязанностей
- Observer – Наблюдатель
- Strategy – Стратегия
- Command – Команда
- State – Состояние
- Visitor – Посетитель
- Interpreter – Интерпретатор
- Iterator – Итератор
- Memento – Хранитель

Эти паттерны решают задачи эффективного и безопасного взаимодействия между объектами программы.

Цепочка обязанностей

Цепочка обязанностей — это поведенческий паттерн проектирования, который 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.

UML-представление паттерна:



Цепочка обязанностей

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

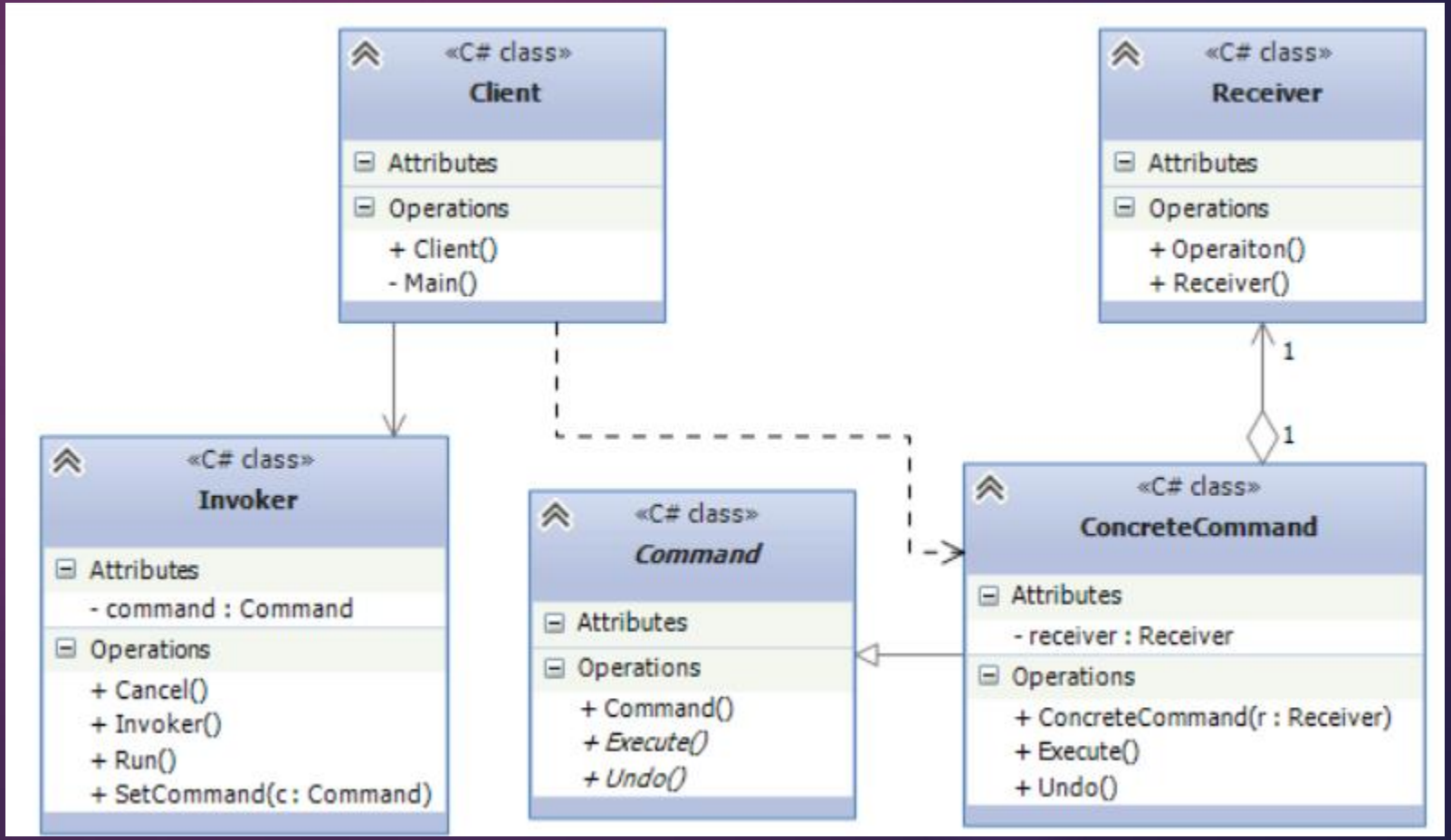
Цепочка обязанностей

- **Преимущества и недостатки:**
- Уменьшает зависимость между клиентом и обработчиками.
- Реализует *принцип единственной обязанности*.
- Реализует *принцип открытости/закрытости*.
- Запрос может остаться никем не обработанным.

Команда

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.

Схематично в UML паттерн Команда представляется следующим образом:



Команда

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

Команда

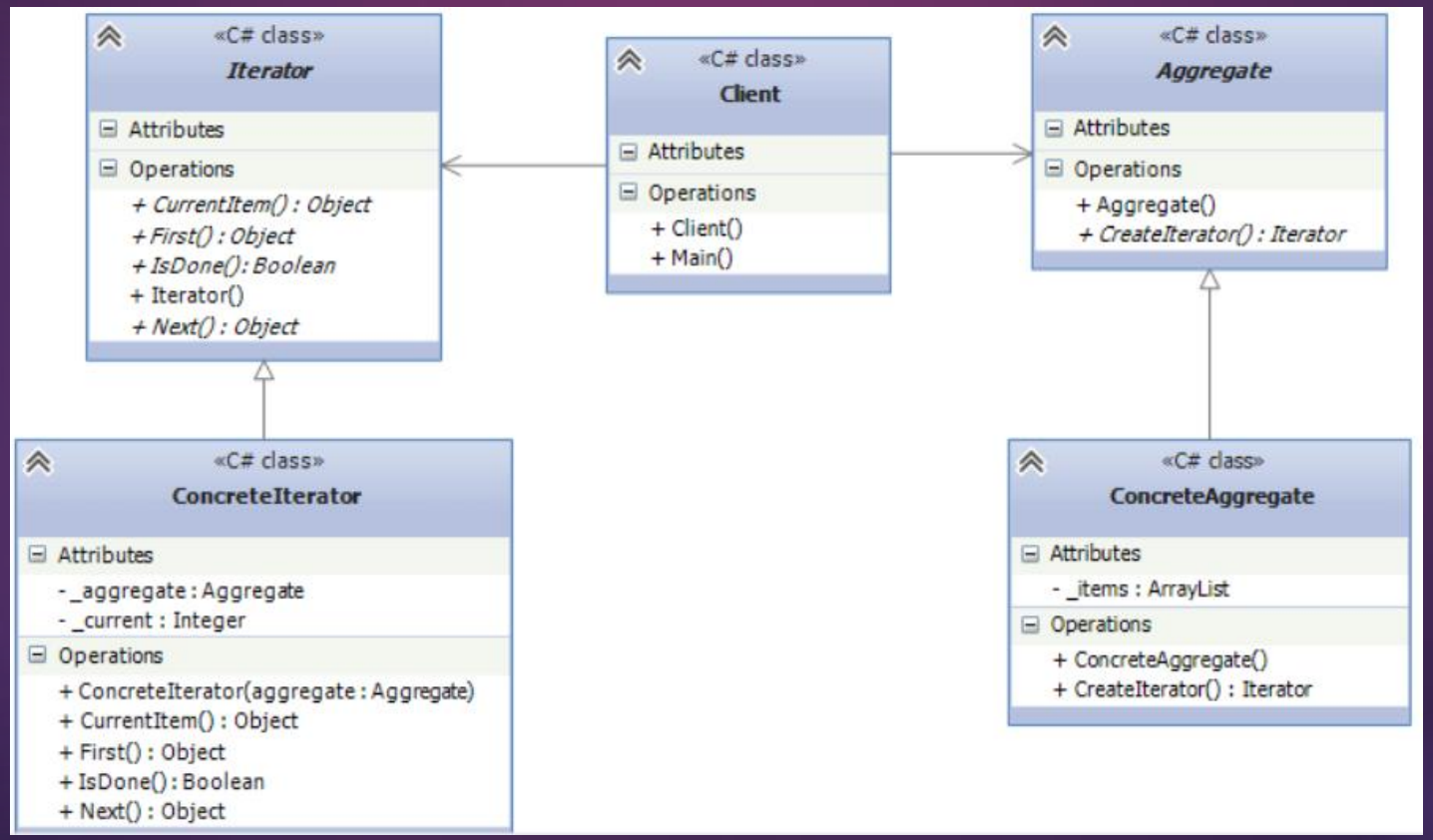
Преимущества и недостатки

- Убирает прямую зависимость между объектами, вызывающими операции, и объектами, которые их непосредственно выполняют.
- Позволяет реализовать простую отмену и повтор операций.
- Позволяет реализовать отложенный запуск операций.
- Позволяет собирать сложные команды из простых.
- Реализует *принцип открытости/закрытости*.
- Усложняет код программы из-за введения множества дополнительных классов.

Итератор

Итератор — это поведенческий паттерн проектирования, который даёт возможность последовательно обходить элементы составных объектов, не раскрывая их внутреннего представления.

С помощью схем UML итераторы можно описать так:



Итератор

Когда использовать итераторы?

- Когда необходимо осуществить обход объекта без раскрытия его внутренней структуры
- Когда имеется набор составных объектов, и надо обеспечить единый интерфейс для их перебора
- Когда необходимо предоставить несколько альтернативных вариантов перебора одного и того же объекта

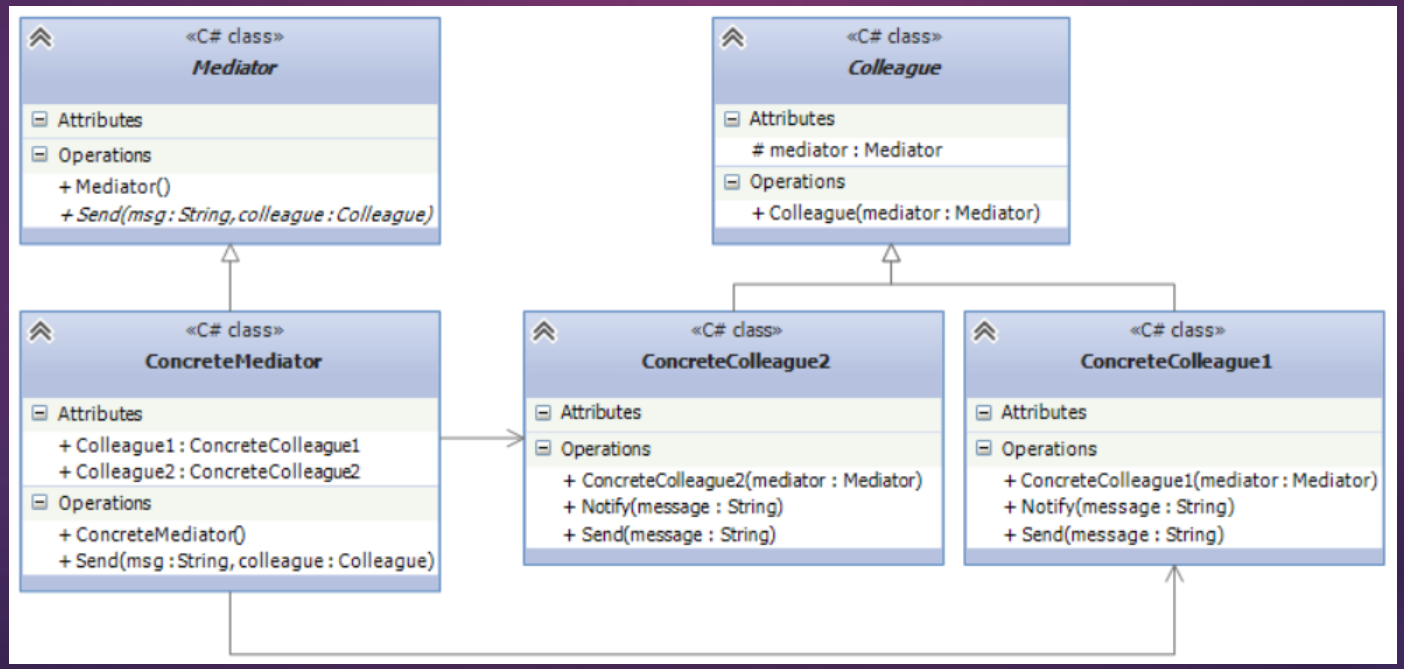
Итератор

- **Преимущества и недостатки**
- Упрощает классы хранения данных.
- Позволяет реализовать различные способы обхода структуры данных.
- Позволяет одновременно перемещаться по структуре данных в разные стороны.
- Не оправдан, если можно обойтись простым циклом.

Посредник

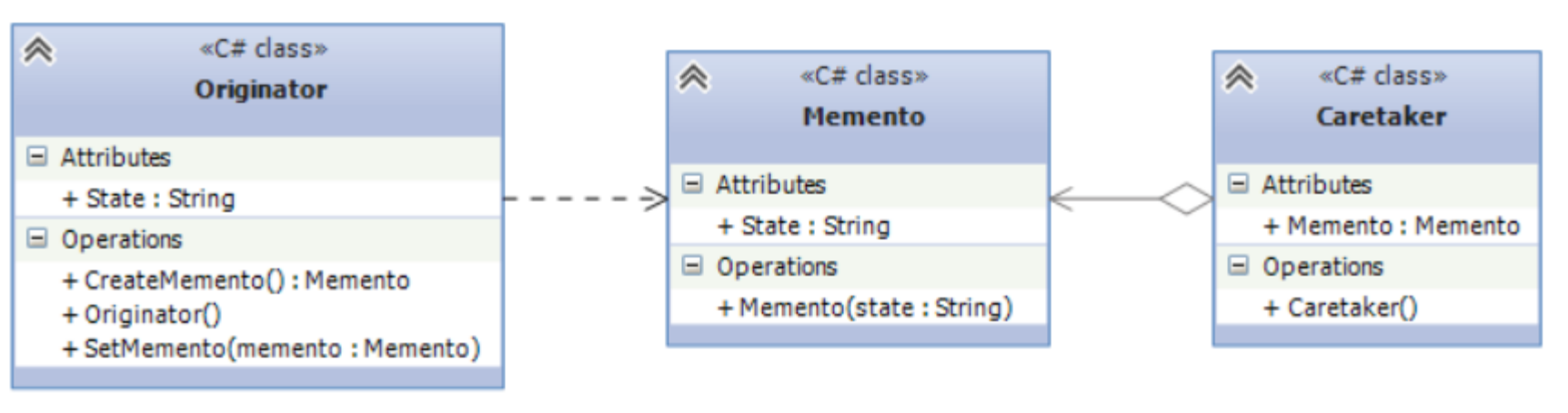
Посредник — это поведенческий паттерн проектирования, который позволяет уменьшить связанность множества классов между собой, благодаря перемещению этих связей в один класс-посредник.

Схематично с помощью UML паттерн можно описать следующим образом:



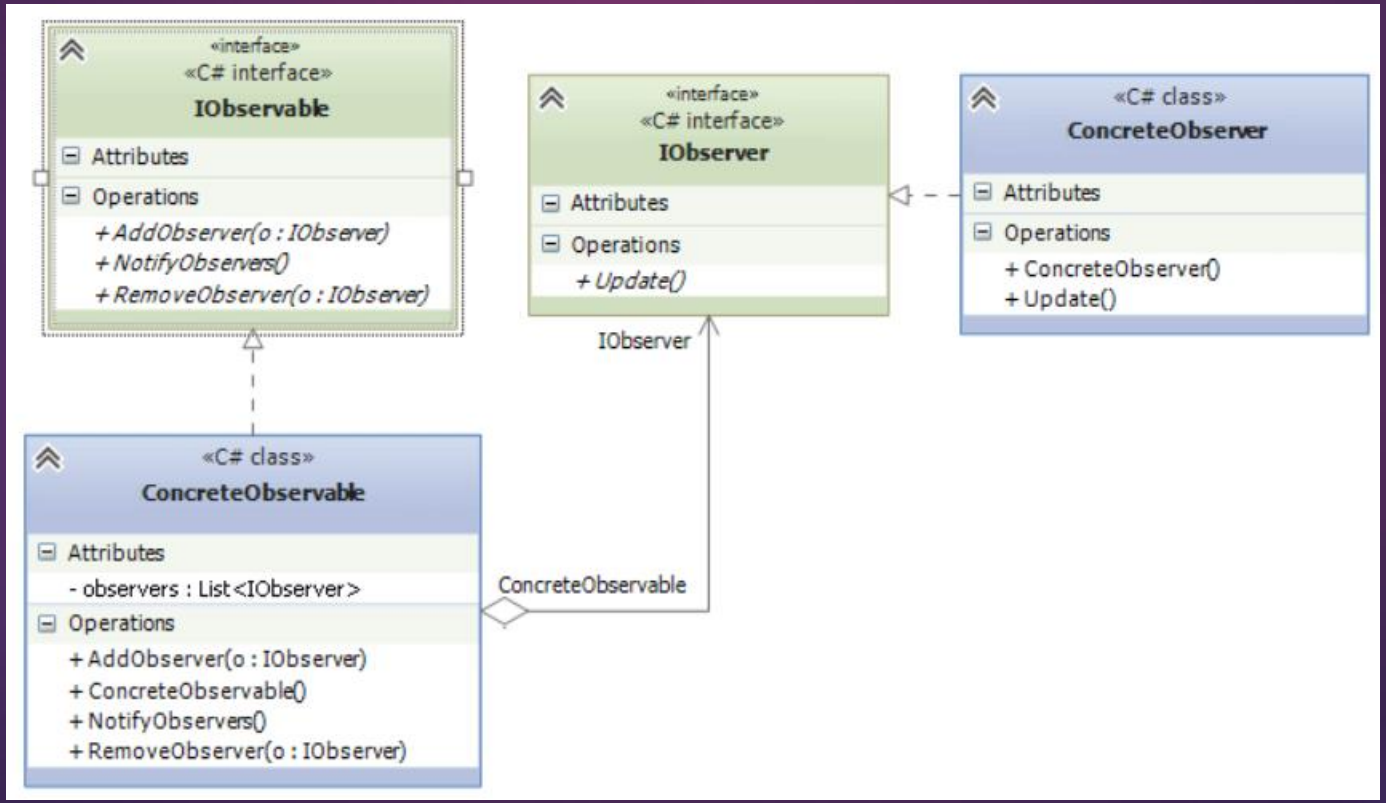
Снимок

Снимок — это поведенческий паттерн проектирования, который позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.



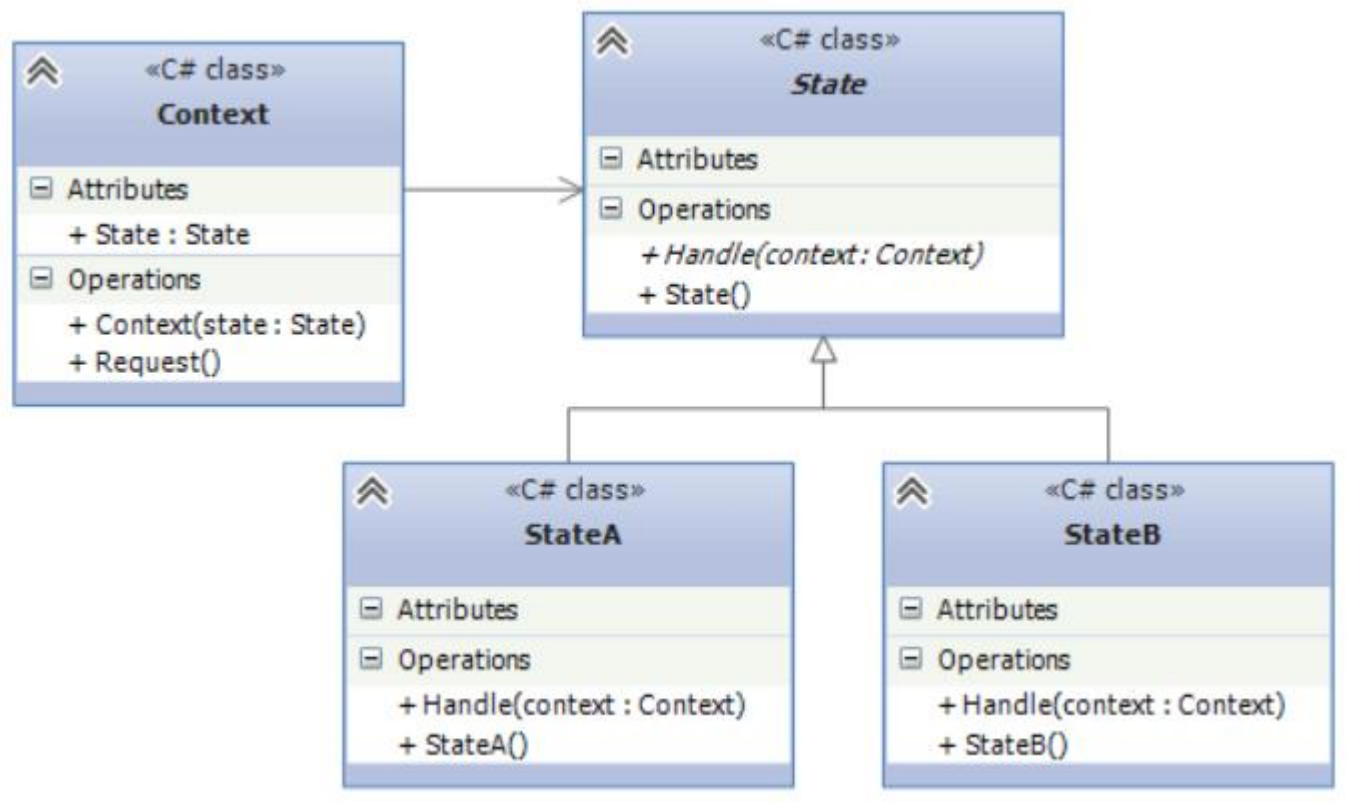
Наблюдатель

Наблюдатель — это поведенческий паттерн проектирования, который создаёт механизм подписки, позволяющий одним объектам следить и реагировать на события, происходящие в других объектах.



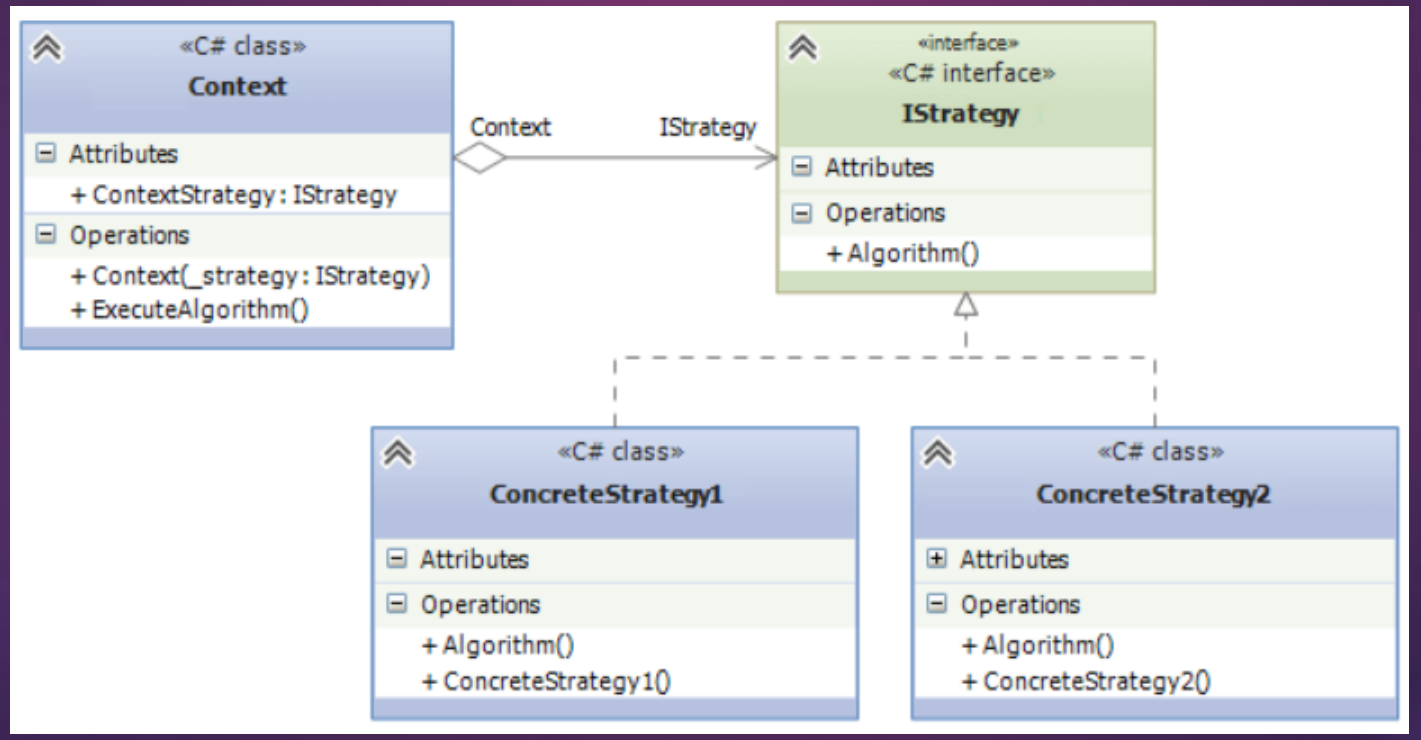
Состояние

Состояние — это поведенческий паттерн проектирования, который позволяет объектам менять поведение в зависимости от своего состояния. Извне создаётся впечатление, что изменился класс объекта.



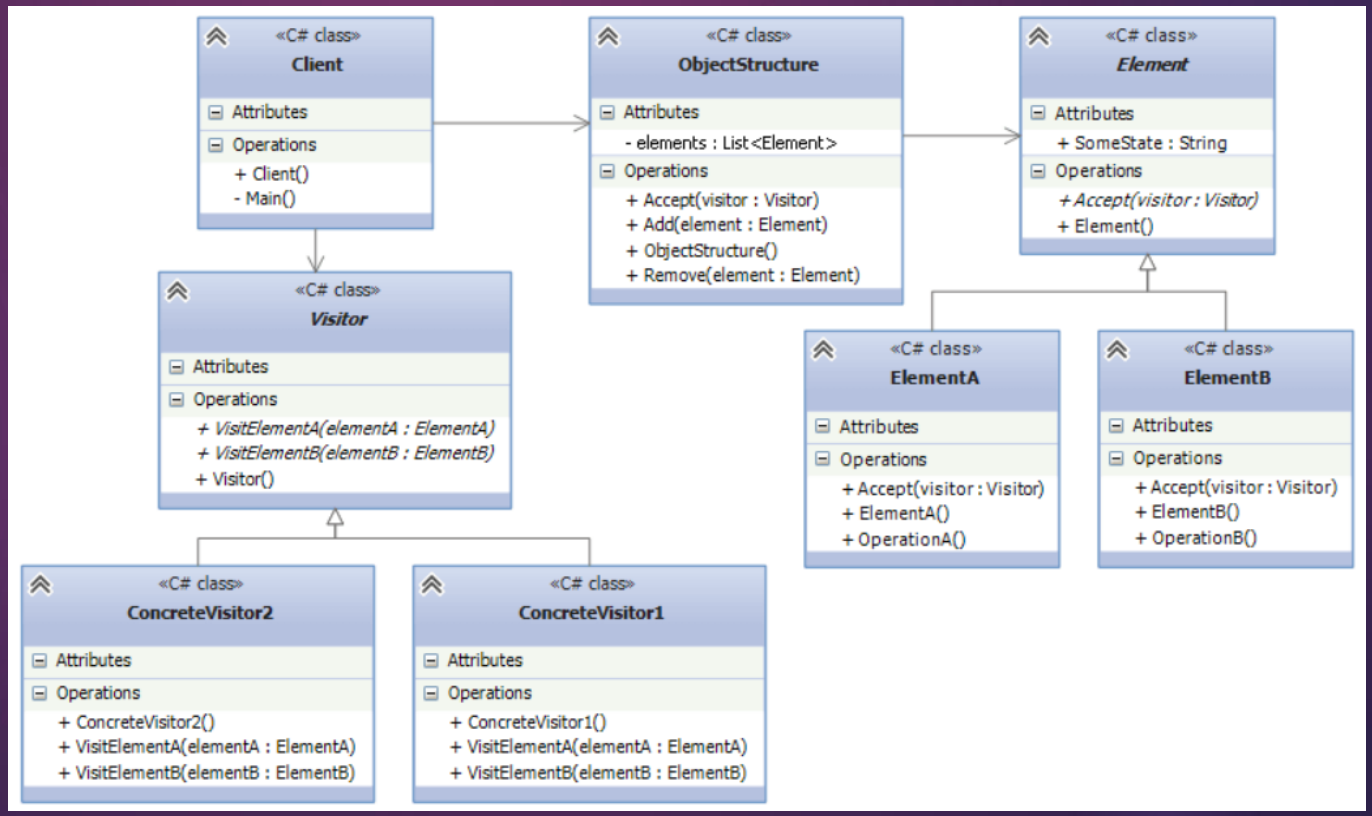
Стратегия

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.



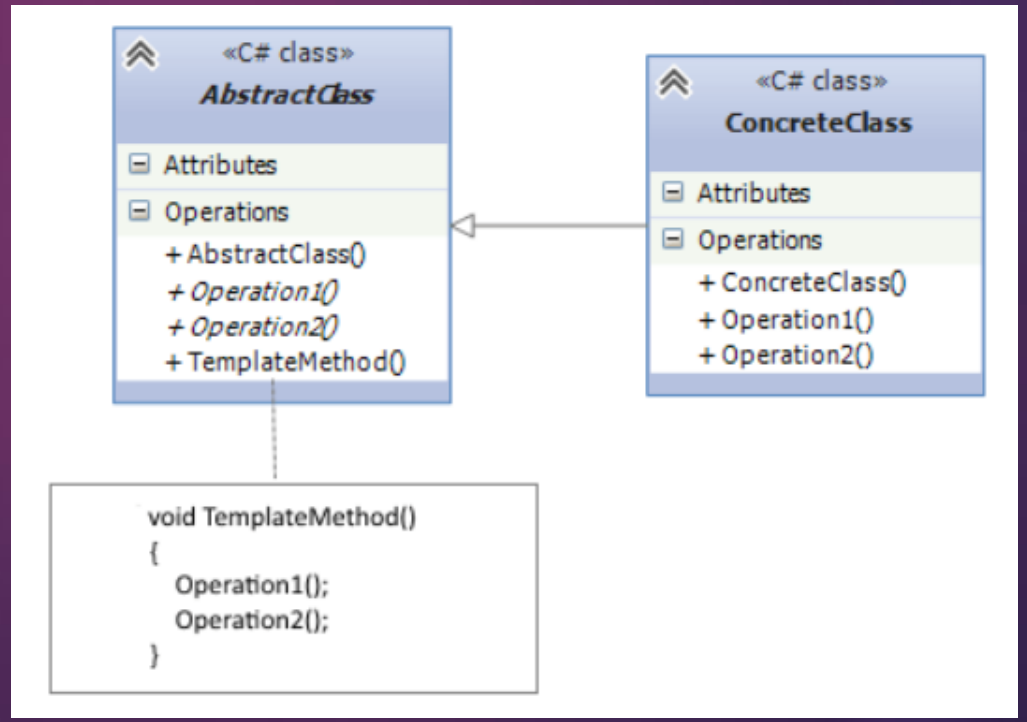
Посетитель

Посетитель — это поведенческий паттерн проектирования, который позволяет добавлять в программу новые операции, не изменяя классы объектов, над которыми эти операции могут выполняться.



Шаблонный метод

Шаблонный метод — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.



Конец.

Презентацию подготовил Гапоненко Александр из группы 750505