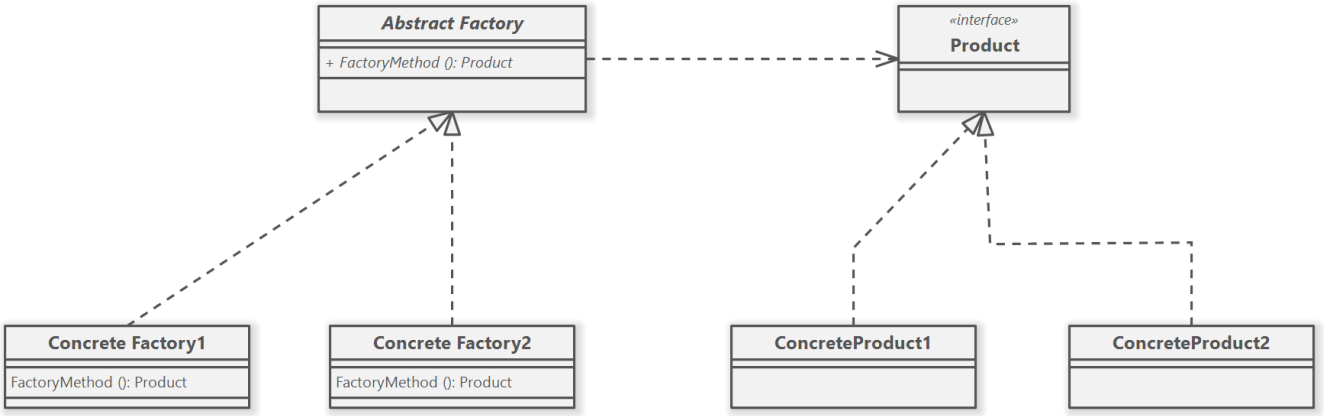
Шаблоны проектирования

Содержание

Порождающие паттерны

Фабричный метод

Фабричный метод - шаблон проектирования, определяющий общий интерфейс создания объектов. Позволяя дочерним классам изменять тип создаваемых объектов.



Применимость:

- Когда заранее неизвестны типы и зависимости порождаемых объектов.
- Когда необходимо дать пользователям возможность расширять части библиотеки.
- Повышает возможность переиспользования кода.

Шаблон позволяет разделить в отдельные иерархии производящий объекты код и сами необходимые объекты, благодаря чему добавление новых типов объектов и их создание производится без вмешательства в основной код

Шаги реализации

1. Привести создаваемые объекты к единому интерфейсу.
2. В производящем классе создать пустой метод, возвращающий интерфейс создаваемых объектов.
3. Для каждого типа создаваемых объектов создать отдельный подкласс, наследующий базовый производящий класс.
4. Фабричный метод можно параметризировать, чтобы в одном методе создавать несколько типов объектов.
5. Все создания объектов заменить на вызовы фабричных методов конкретных классов-фабрик.

Преимущества

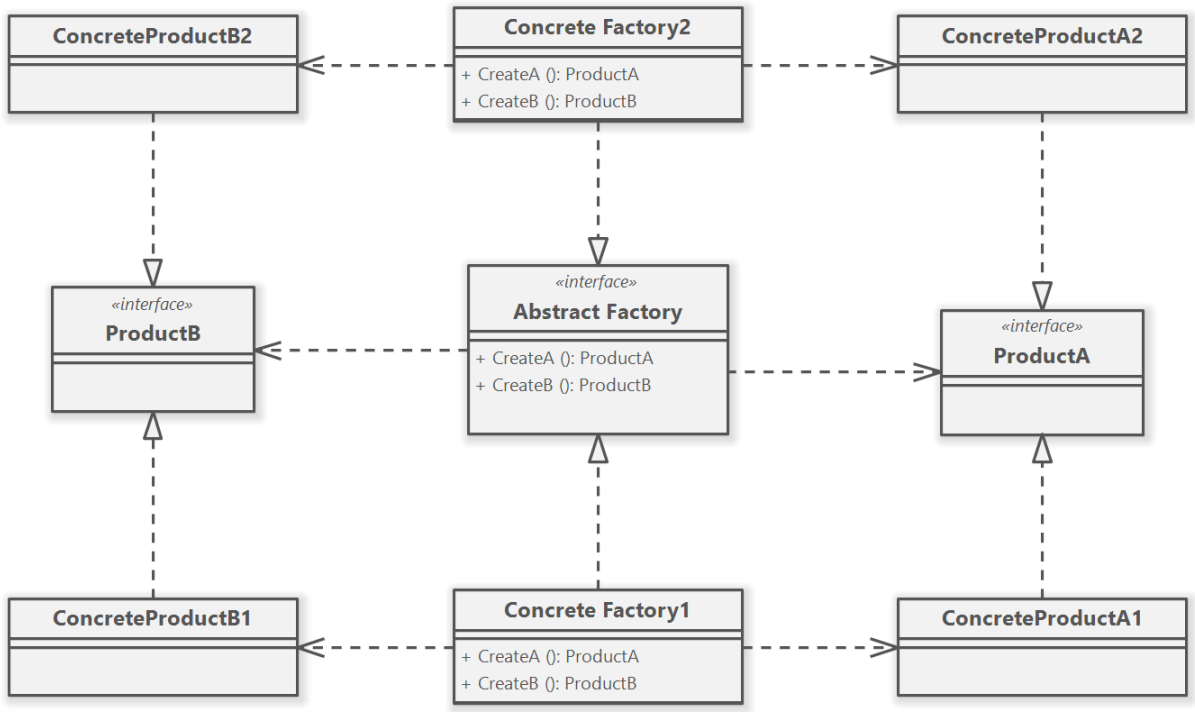
- Убирает прямую зависимость основного кода от конкретных классов.
- Выделяет код производства в отдельное место.
- Упрощает добавление новых типов объектов.
- Реализует принцип открытости к расширению / закрытости для изменений.

Недостатки

- Вместо одной иерархии классов получаем две, равных по мощности множеств классов.

Абстрактная фабрика

Абстрактная фабрика - шаблон проектирования, позволяющий создавать семейства связанных объектов, убирая прямую зависимость от классов порождаемых объектов.



Применимость:

- Когда необходимо работать с разными видами связанных объектов.

Шаги реализации

1. Привести группы объектов к общим интерфейсам.
2. Определить интерфейс абстрактной фабрики, где есть методы создания каждой группы объектов.
3. Реализовать абстрактную фабрику для каждой группы объектов.
4. Создать в основном коде конкретную фабрику и заменить создание объектов на вызовы методов фабрики.

Преимущества

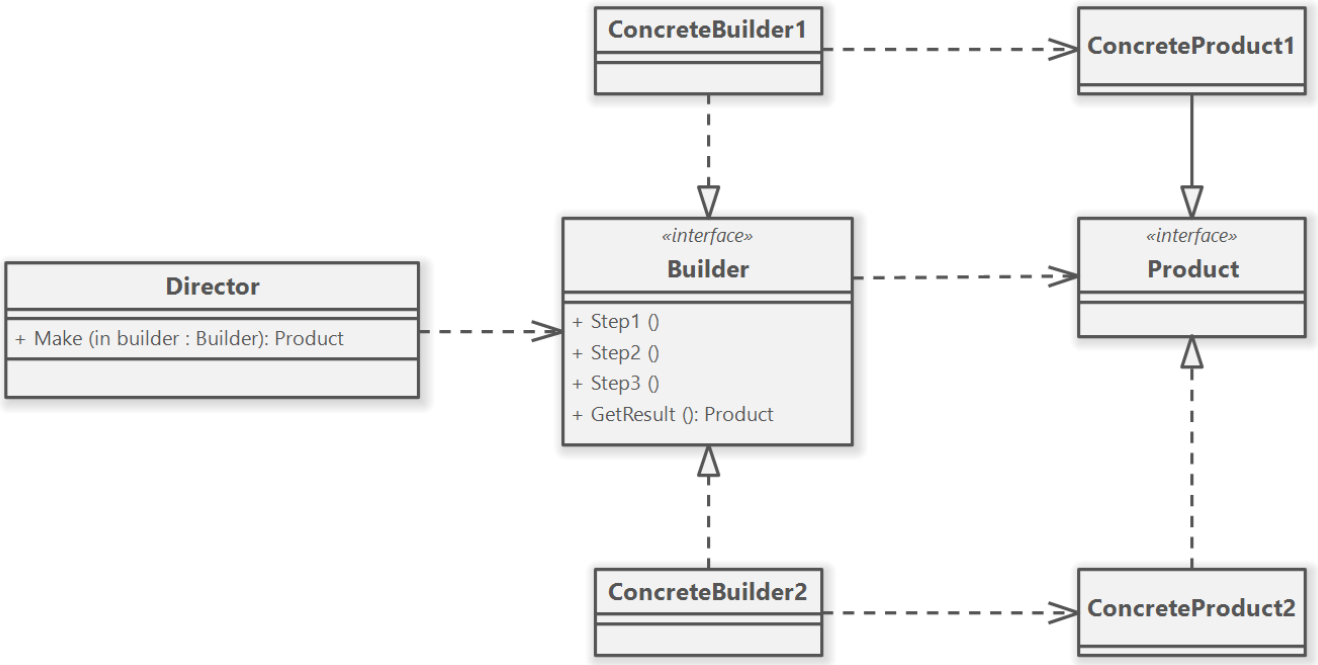
- Гарантирует сочетаемость создаваемых продуктов.
- Избавляет клиентский код от привязки к конкретным классам объектов.
- Выделяет код производства в одно место.
- Упрощает добавление новых типов объектов.
- Реализует принцип открытости к расширению / закрытости для изменений.

Недостатки

- Требует наличия всех типов объектов в каждой группе.

Строитель

Строитель - шаблон проектирования позволяющий создавать сложные объекты пошагово



Применимость:

- Когда необходимо избавиться от телескопического конструктора.
- Когда нужно создавать разные представления одного объекта.
- Когда нужно создавать составные объекты, например сложные деревья.

Шаги реализации

1. Определить интерфейс для шагов создания объектов.
2. Для каждого представления создать класс-строитель.
3. Создать класс-директор. Он вызывает методы строителя в нужном порядке.

Преимущества

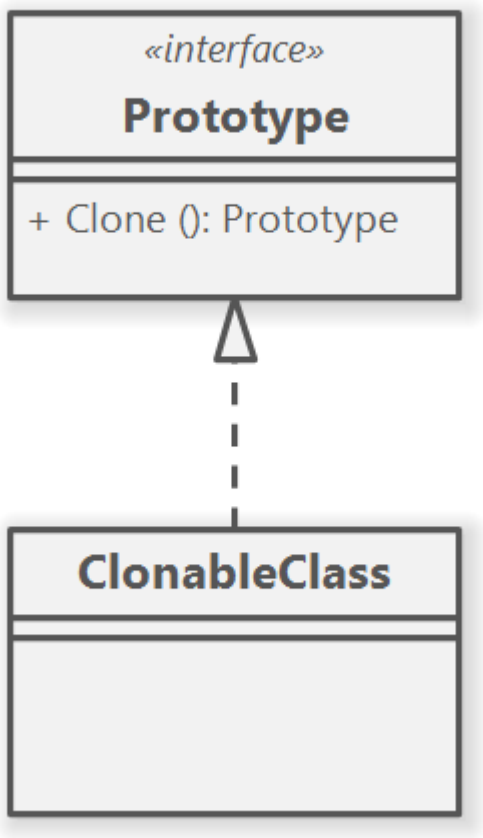
- Пошаговое создание объектов.
- Использование одного кода для создания разных объектов.
- Инкапсулирует сложный код сборки.

Недостатки

- Усложняет код программы.
- Клиентский код возможно будет привязан к конкретным классам строителей.

Прототип

Прототип - шаблон проектирования позволяющий копировать объекты.



Применимость:

- Когда клиентский код не должен зависеть от классов копируемых объектов.
- Когда нужно порождать много объектов с определённой конфигурацией.

Шаги реализации

1. Определить интерфейс с методом clone.
2. Реализовать этот интерфейс в классах которые требуется копировать.
3. Опционально можно создать коллекцию эталонных объектов.

Преимущества

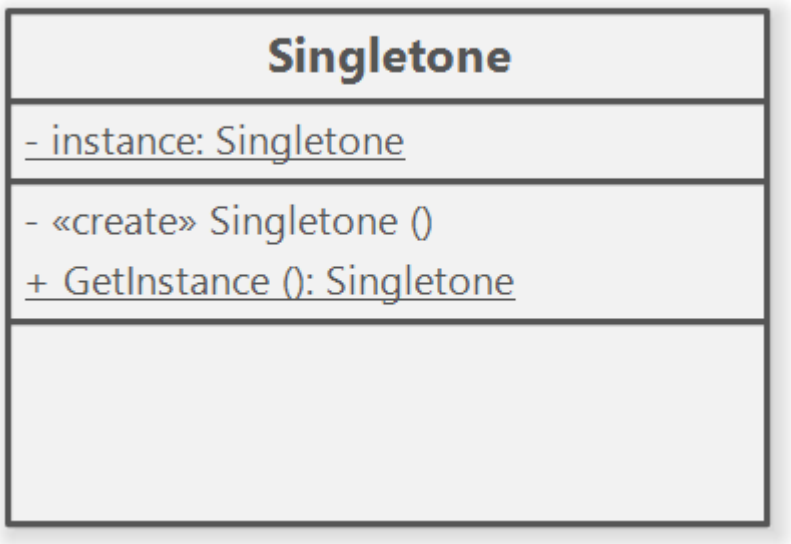
- Позволяет клонировать объекты, не привязываясь к их конкретным классам.
- Меньше повторяющегося кода инициализации объектов.
- Ускоряет создание объектов.

Недостатки

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

Синглтон

Синглтон - шаблон проектирования позволяющий гарантировать, что экземпляр класса будет создан только один.



Применимость:

- Когда в программе должен быть единственный экземпляр какого-то класса, доступный всем клиентам (например, общий доступ к базе данных из разных частей программы).
- Когда нужно централизовать управление глобальным состоянием.

Шаги реализации

1. Определить класс с приватным статическим полем, где будет содержаться ссылка на объект.
2. Определить статический метод для получения ссылки на объект.
3. Добавить "ленивую инициализацию" в статический метод.
4. Сделать конструктор класса приватным.

Преимущества

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

Недостатки

- Маскирует плохой дизайн.
- Проблемы многопоточности.
- Требует постоянного создания Mock-объектов при юнит-тестировании.

Структурные паттерны

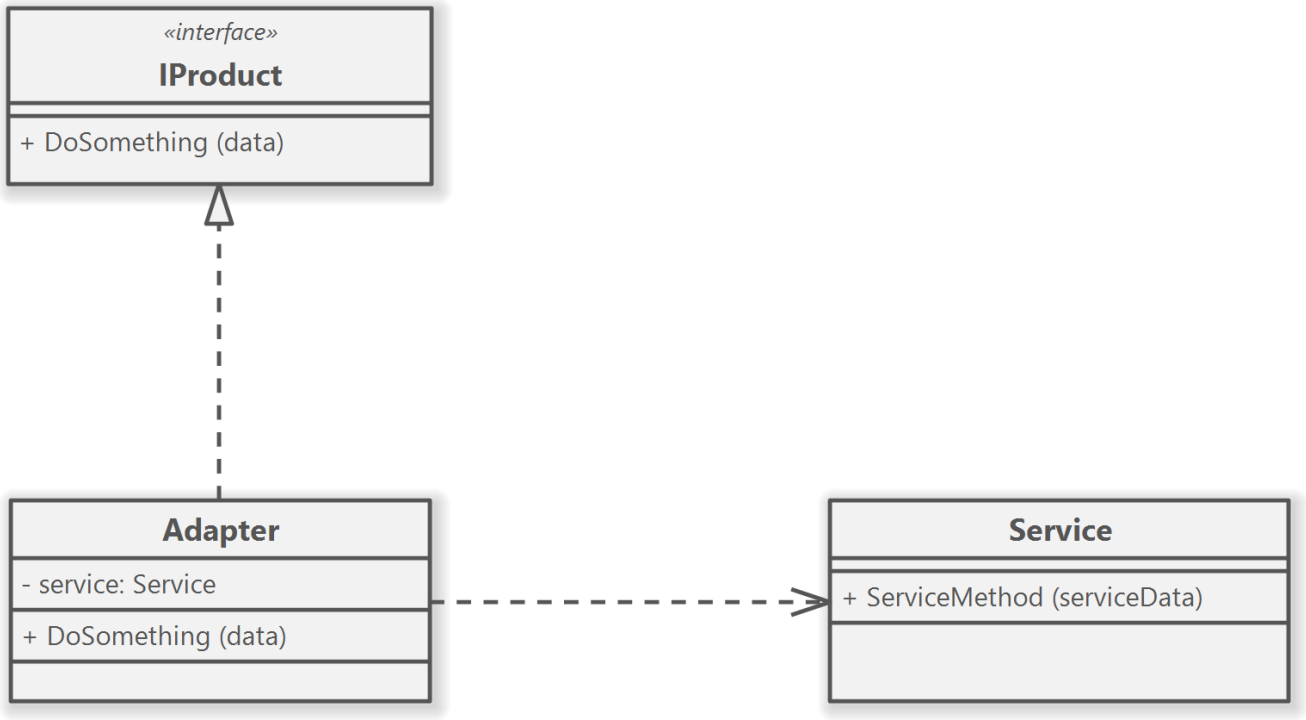
Структурные шаблоны проектирования фокусируются на описании подходов к комбинированию классов для решения определённых задач. Порождающие шаблоны фокусировались на построении структур классов, которые могут порождать объекты с заданными свойствами, а сами структуры удовлетворяют определённым условиям. Структурные шаблоны идут дальше и расширяют сферу влияния на другие задачи кроме порождения объектов.

Адаптер

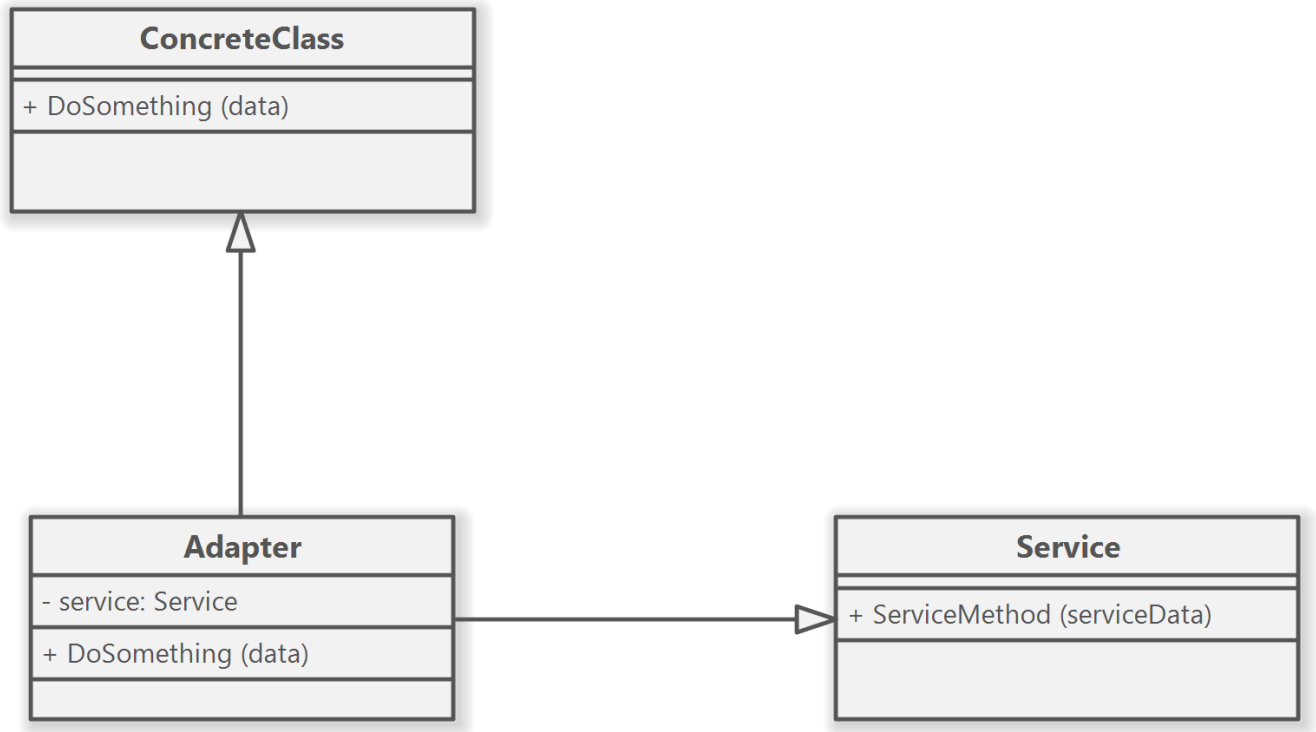
Адаптер - шаблон который позволяет заставить реализованный класс выполнять нужный контракт.

Существует два подхода к реализации:

- Адаптер объектов - использует агрегацию.



- Адаптер классов - использует наследование (возможен только в языках, поддерживающих множественное наследование классов).



Применимость:

- Когда необходимо использовать класс, но он нереализует нужный интерфейс.
- Для расширения общей функциональности классов.

Шаги реализации:

1. Описать интерфейс, которому должен соответствовать класс.
2. Создать класс-адаптер, реализующий интерфейс.
3. Поместить в адаптер поле, хранящее ссылку на объект адаптируемого класса.
4. Реализовать интерфейс в адаптере.
5. Использовать адаптер через интерфейс.

Преимущества:

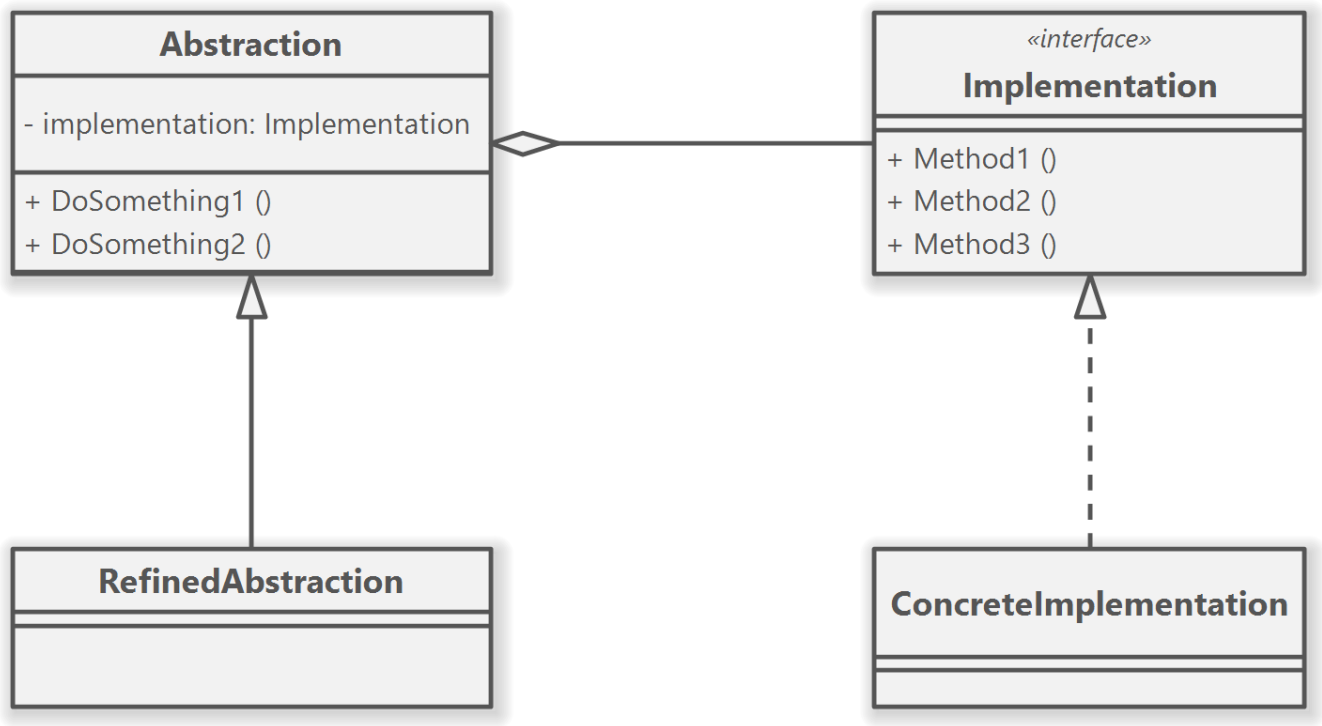
- Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

Недостатки:

- Усложняет код программы из-за введения дополнительных классов.

Мост

Мост разделяет классы на абстракцию и реализацию, позволяя менять поведение объектов.



Применимость:

- Разделение большого монолитного класса.
- Расширение класса в двух плоскостях.
- Если нужно менять реализацию во время выполнения программы.

Шаги реализации

1. Описать контракт взаимодействия с клиентским кодом в абстракции.
2. Определить поведение, нужное для функционирования абстракции и на основе него создать интерфейс для реализации.
3. Добавить в абстракцию ссылку на реализацию.
4. Реализовать методы абстракции, делегируя основную работу реализации.

Преимущества:

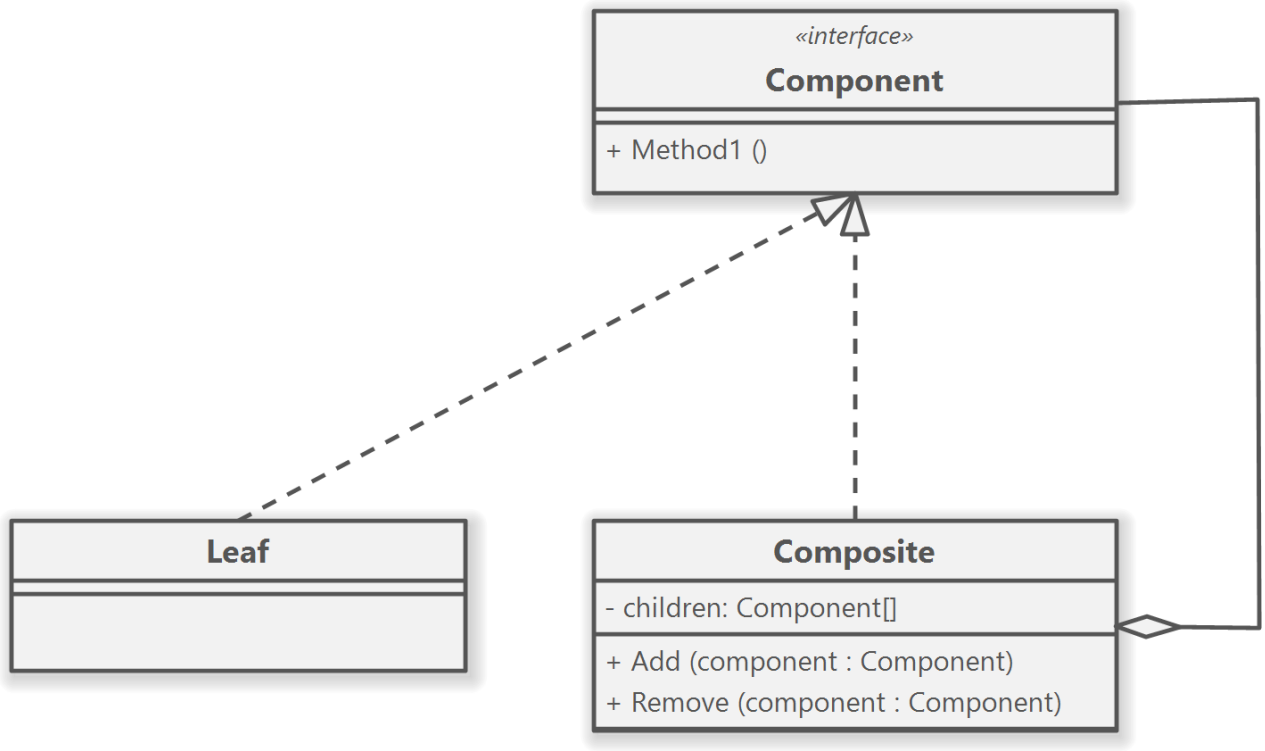
- Позволяет строить платформо-независимые программы.
- Скрывает лишние или опасные детали реализации от клиентского кода.
- Реализует принцип открытости/закрытости.

Недостатки:

- Усложняет код программы из-за введения дополнительных классов.

Компоновщик

Компоновщик позволяет сгруппировать множество объектов в древовидную структуру и работать с ней как с единым объектом.



Применимость:

- Когда нужно представить древовидную структуру.
- Когда нужно единообразно работать с простыми и составными объектами.

Шаги реализации:

1. Создать общий интерфейс компонентов.

2. Реализовать интерфейс в двух классах - листе и контейнере.
3. Добавить методы добавления/удаления дочерних компонентов в класс-контейнер.

Преимущества:

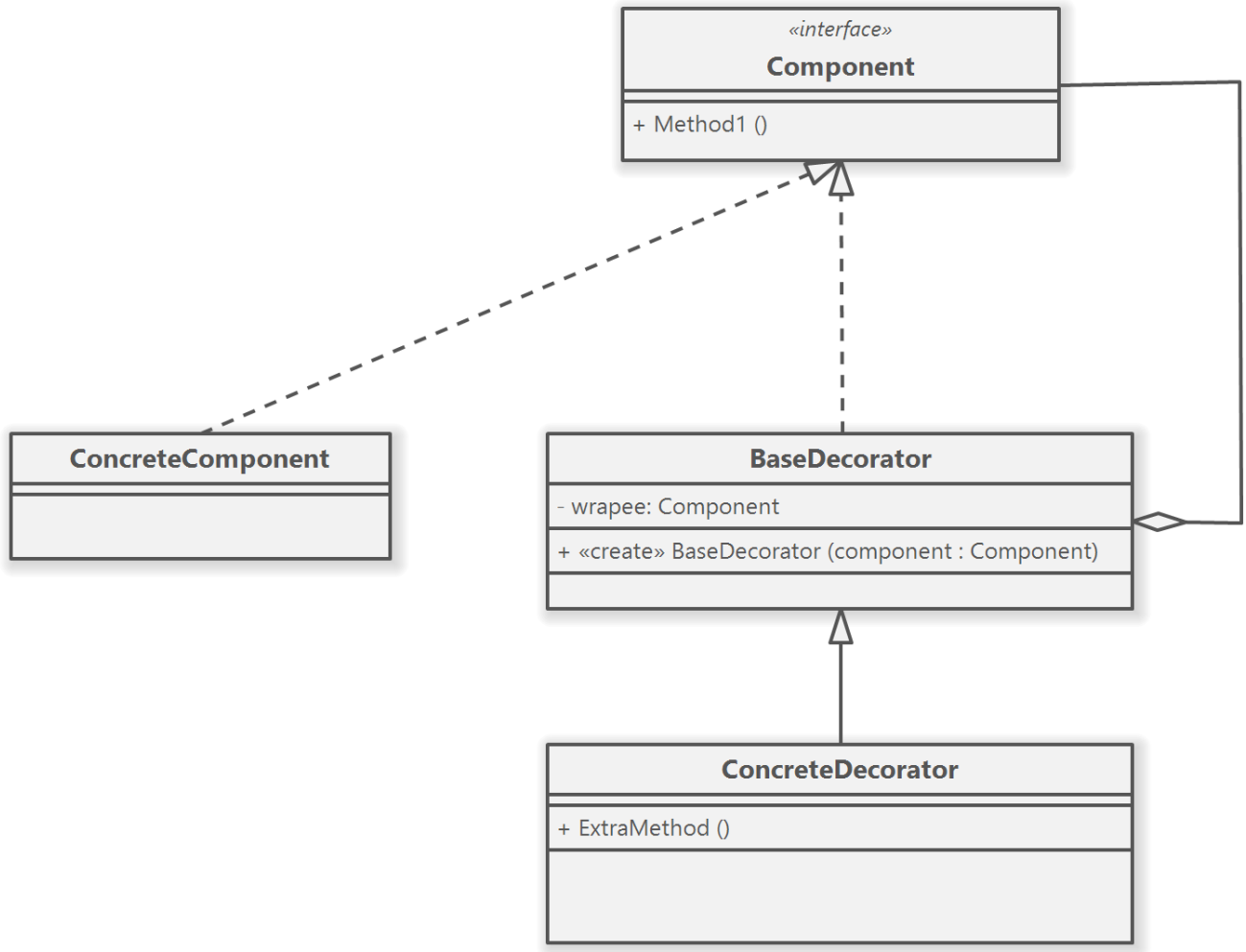
- Упрощает архитектуру клиента при работе со сложным деревом компонентов.
- Облегчает добавление новых видов компонентов.

Недостатки:

- Создаёт слишком общий дизайн классов. В интерфейсе объектов может быть недостаточно функционала.

Декоратор

Декоратор позволяет добавлять функциональность в реализованные классы, оборачивая их в классы-обёртки.



Применимость:

- Когда нужно добавлять функционал на лету.
- Когда нельзя расширить функционал при помощи наследования.

Шаги реализации:

1. Реализовать интерфейс оборачиваемого класса в базовом декораторе.
2. Добавить в базовый декоратор ссылку на интерфейс.
3. Наследовать декораторы от базового, реализуя в них дополнительное поведение.

Преимущества:

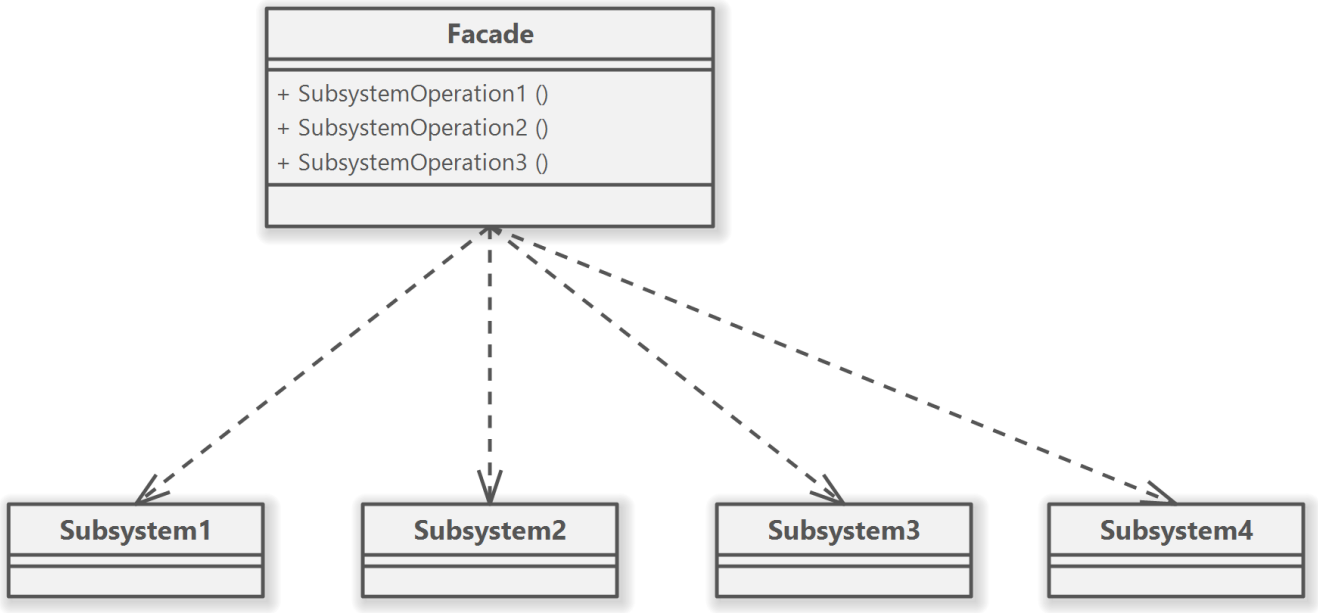
- Большая гибкость, чем у наследования.
- Позволяет добавлять обязанности на лету.
- Можно добавлять несколько новых функций сразу.
- Позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.

Недостатки:

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

Фасад

Фасад предоставляет простой интерфейс к сложной системе.



Применимость:

- Когда нужно представить простой или урезанный интерфейс к сложной подсистеме.
- Когда нужно разложить подсистему на отдельные слои.

Шаги реализации:

1. Создать интерфейс, описывающий поведение нужное клиенту.
2. Реализовать интерфейс в классе-фасаде, который передаёт вызовы клиентского кода нужным объектам.
3. Добавить ещё фасады, если существующий станет слишком большим.

Преимущества:

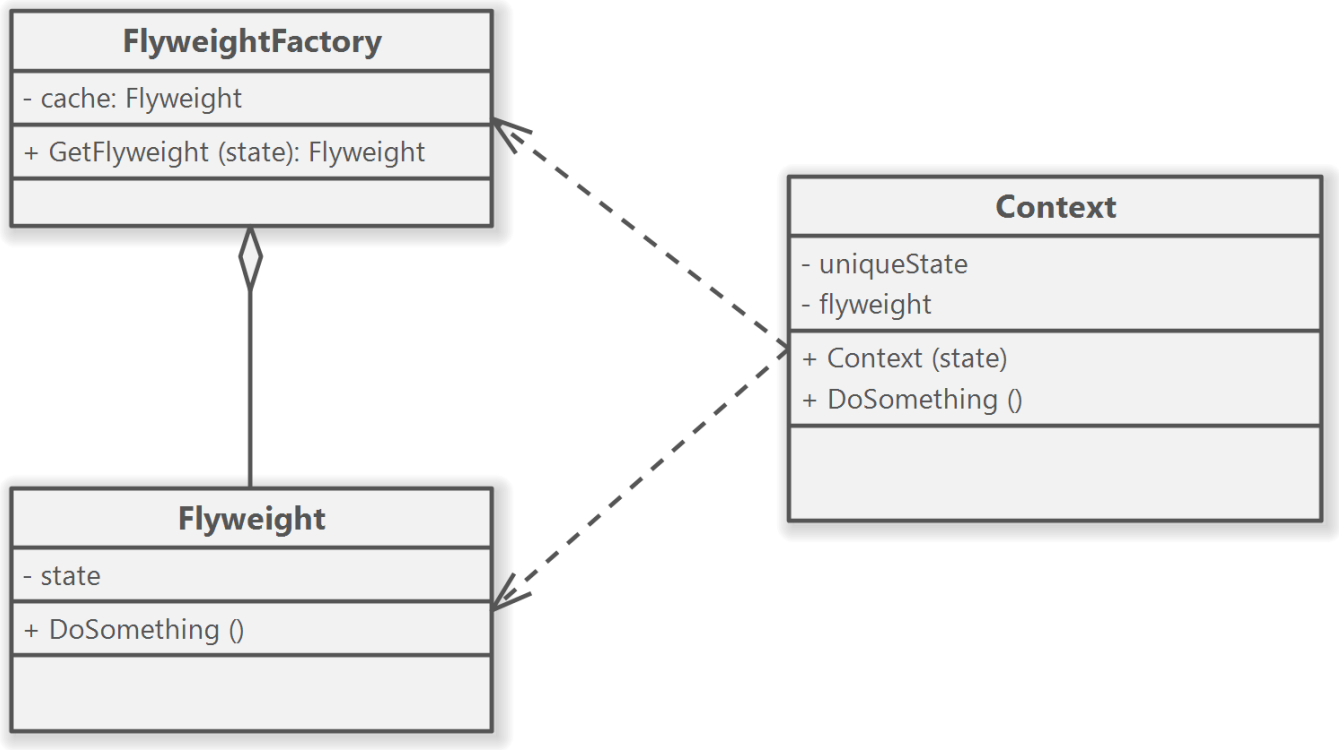
- Изолирует клиентов от компонентов сложной подсистемы.

Недостатки:

- Фасад рискует стать божественным объектом, привязанным ко всем классам программы.

Легковес

Позволяет вмещать большее количество объектов в отведённую оперативную память.



Применимость:

- Когда не хватает оперативной памяти для поддержки всех нужных объектов.

Шаги реализации:

1. Разделить поля класса, на две части: внутреннее состояние - значение одинаково для большей части объектов, и контекст.
2. Оставить внутреннее состояние в классе.
3. Поля, представляющие контекст, удалить из класса и передавать из через методы.
4. Создать фабрику, которая хранит легковесов с определённым внутренним состоянием.
5. Клиент должен запрашивать объект с внутренним состоянием через фабрику и передавать полученному объекту контекст.

Преимущества:

- Экономит оперативную память.

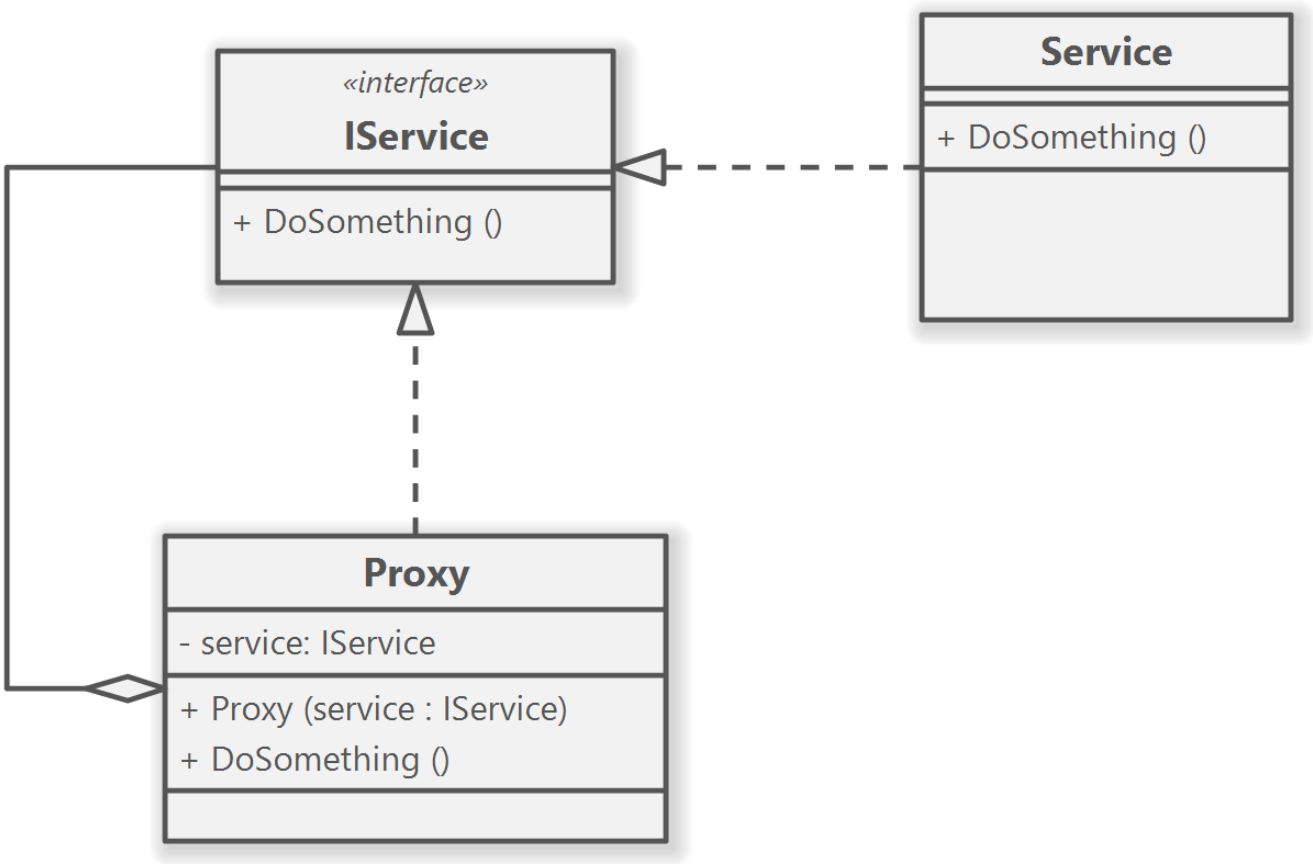
Недостатки:

- Расходует процессорное время на поиск/вычисление контекста.

- Усложняет код программы из-за введения множества дополнительных классов.

Заместитель

Заместитель позволяет подставлять вместо реальных объектов объекты, перехватывающие вызовы к оригинальному объекту, позволяя делать что-то до и после передачи вызова к объекту.



Применимость:

- Ленивая инициализация (виртуальный прокси).
- Защита доступа (защищающий прокси). Когда в программе есть разные типы пользователей, и нужно защищать объект от неавторизованного доступа.
- Локальный запуск сервиса (удалённый прокси). Когда настоящий сервисный объект находится на удалённом сервере.
- Логирование запросов (логирующий прокси). Когда требуется хранить историю обращений к сервисному объекту.
- Кэширование объектов. Когда нужно кэшировать результаты запросов клиентов и управлять их жизненным циклом.

Шаги реализации:

1. Реализовать интерфейс оригинального объекта в заместителе.
2. Добавить в заместитель ссылку на оригинальный объект.
3. Реализовать методы в заместителе так чтобы основная работа делегировалась оригинальному объекту, а функционал до и после вызова были реализованы в заместителе.

Преимущества:

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

Недостатки:

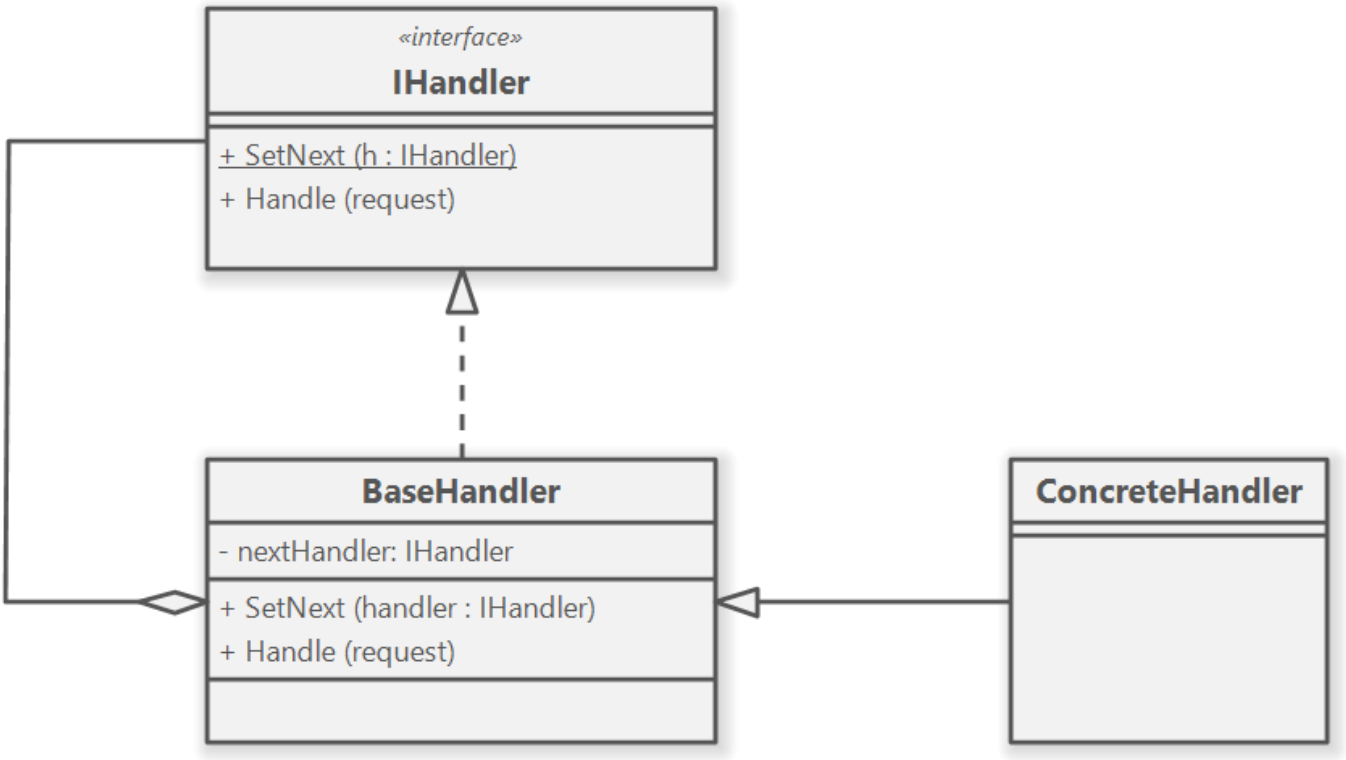
- Усложняет код программы из-за введения дополнительных классов.
- Увеличивает время отклика от сервиса.

Поведенческие паттерны

Поведенческие шаблоны фокусируются на создании структур классов, взаимодействующих между собой определённым образом.

Цепочка обязанностей

Цепочка обязанностей позволяет передавать запрос по цепочке до тех пор пока не найдётся объект, который сможет обработать запрос.



Применимость:

- Когда программа должна обрабатывать разнообразные запросы несколькими способами, но заранее неизвестно, какие конкретно запросы будут приходить и какие обработчики для них понадобятся.
- Когда важно, чтобы обработчики выполнялись один за другим в строгом порядке.
- Когда набор объектов, способных обработать запрос, должен задаваться динамически.

Шаги реализации:

1. Создать интерфейс обработчика.
2. Создать базовый класс-обработчик в котором описаны методы добавления следующего обработчика и проверки на то, что дальше есть ещё обработчик.
3. Реализовать конкретные классы-обработчики.

Преимущества:

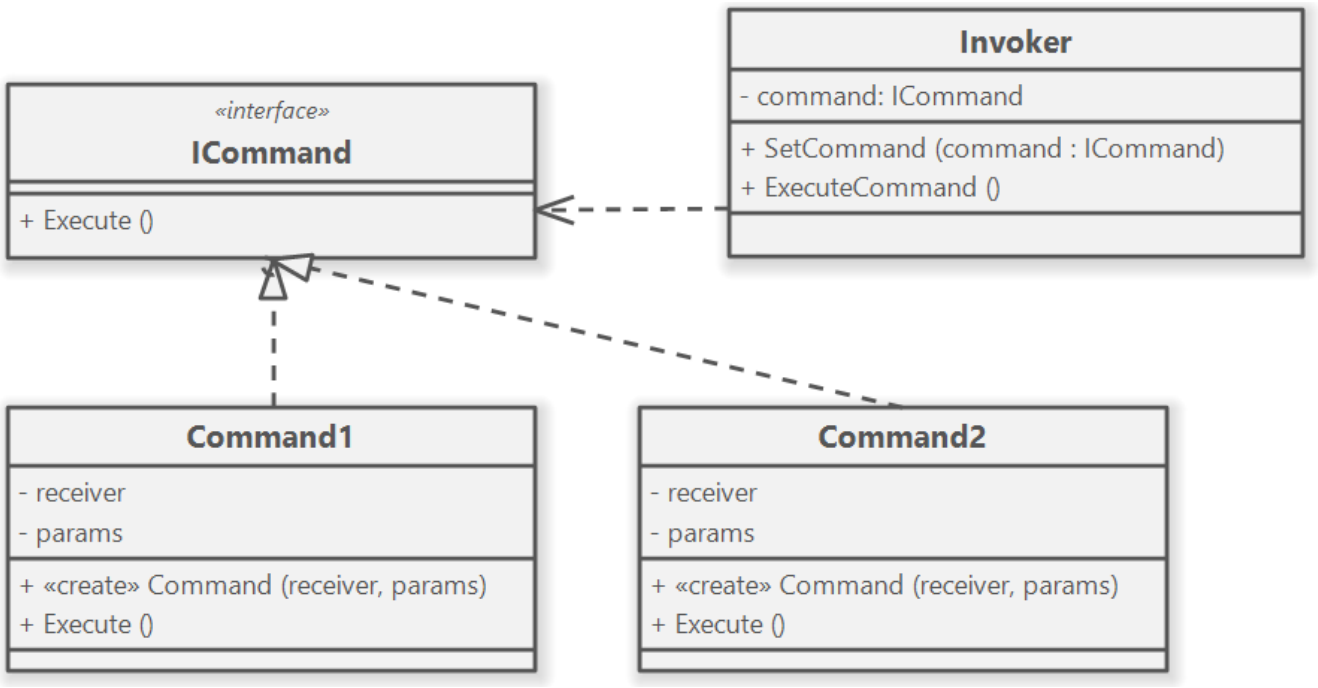
- Уменьшает зависимость между клиентом и обработчиками.
- Реализует принцип единственной обязанности.
- Реализует принцип открытости/закрытости.

Недостатки:

- Запрос может остаться никем не обработанным.

Команда

Команда позволяет превратить запросы в полноценные объекты.



Применимость:

- Когда нужно параметризовать объекты выполняемым действием.
- Когда нужно ставить операции в очередь, выполнять их по расписанию или передавать по сети.
- Когда нужна операция отмены.

Шаги реализации:

1. Создать общий интерфейс для команд.
2. Создать классы команд. В котором хранятся ссылки на объекты-получатели.
3. Реализовать классы-отправители в которых хранятся ссылки на нужные команды.
4. Отправители должны делегировать действия команде.

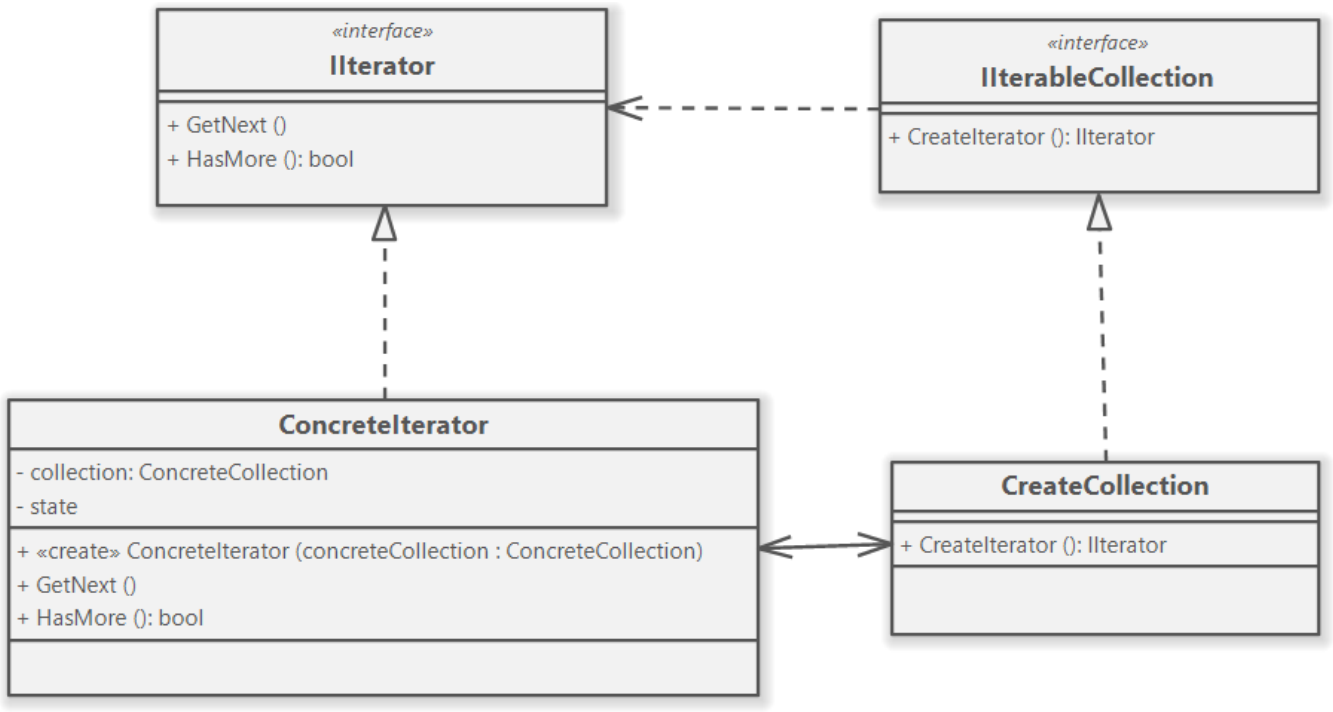
Преимущества:

- 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.

Недостатки:

- Усложняет код программы из-за введения множества дополнительных классов.

Итератор



Итератор даёт возможность обходить элементы коллекции, не раскрывая деталей её реализации.

Применимость:

- Когда у есть сложная структура данных, и нужно скрыть от клиента детали её реализации (из-за сложности или вопросов безопасности).
- Когда нужно иметь несколько вариантов обхода одной и той же структуры данных.
- Когда нужно иметь единый интерфейс обхода различных структур данных.

Шаги реализации:

1. Создать интерфейс для итераторов, в котором будет метод получения следующего элемента.
2. Создать интерфейс для коллекций, в котором будет метод получения итератора.
3. Создать классы конкретных итераторов и коллекций и добавить в них ссылки друг на друга. Через коллекцию можно получить ссылку на итератор и через него получать каждый следующий объект коллекции.

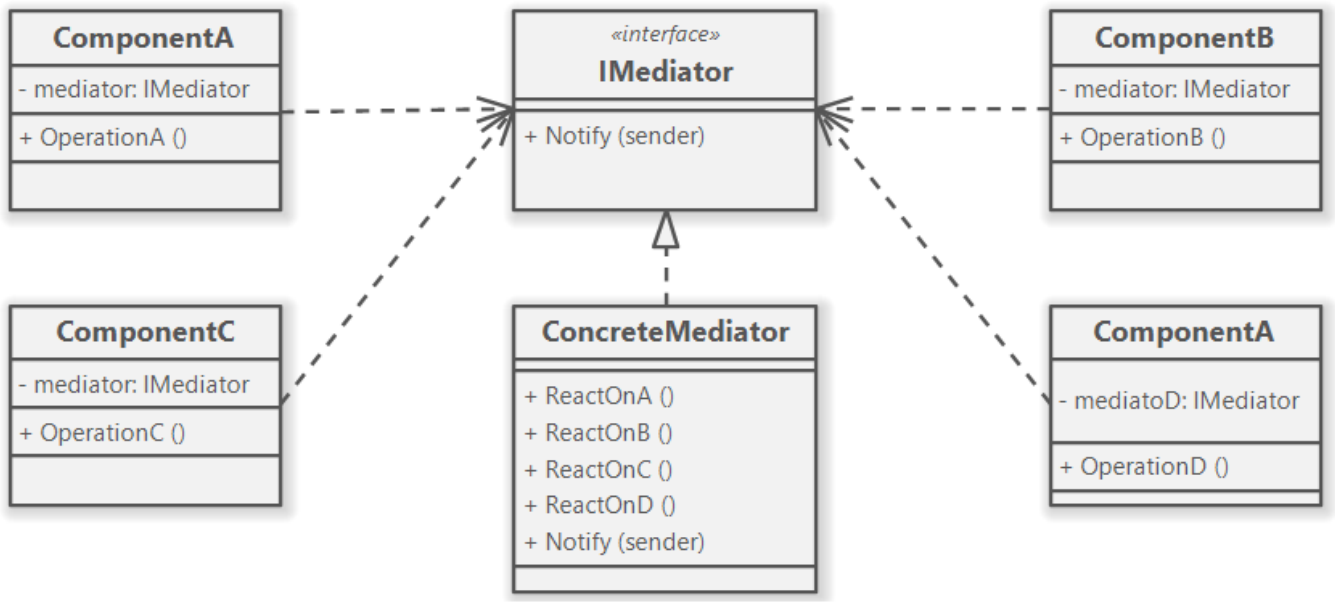
Преимущества:

- Упрощает классы хранения данных.
- Позволяет реализовать различные способы обхода структуры данных.
- Позволяет одновременно перемещаться по структуре данных в разные стороны.

Недостатки:

- Не оправдан, если можно обойтись простым циклом.

Посредник



Посредник позволяет уменьшить связность классов между собой, перемещая связи в один класс-посредник.

Применимость:

- Когда сложно менять некоторые классы из-за того, что они имеют множество хаотичных связей с другими классами.
- Когда невозможно повторно использовать класс, поскольку он зависит от уймы других классов.
- Когда приходится создавать множество подклассов компонентов, чтобы использовать одни и те же компоненты в разных контекстах.

Шаги реализации:

1. Создать интерфейс для посредников.
2. Реализовать конкретных посредников, поместив в них ссылки на связанные компоненты.
3. Поместить в компоненты ссылку на посредника.
4. Изменить код компонента так чтобы он вызывал методы посредника, а не других компонентов.

Преимущества:

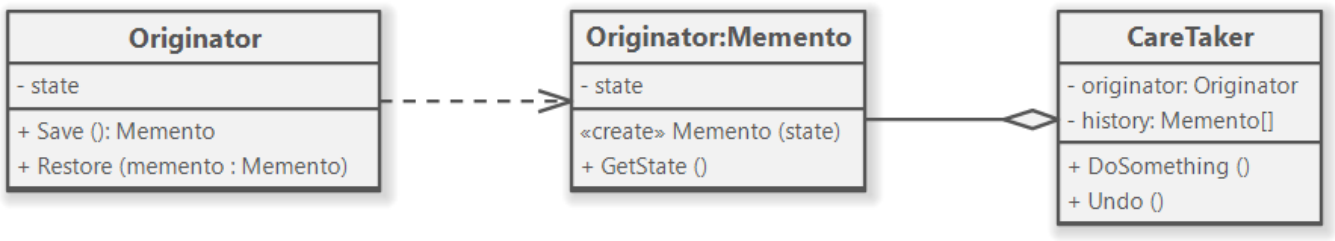
- Устраняет зависимости между компонентами, позволяя повторно их использовать.
- Упрощает взаимодействие между компонентами.
- Централизует управление в одном месте.

Недостатки:

- Посредник может сильно раздуться.

Снимок

Снимок позволяет сохранять и восстанавливать состояния объектов.



Применимость:

- Когда нужно сохранять мгновенные снимки состояния объекта (или его части), чтобы впоследствии объект можно было восстановить в том же состоянии.
- Когда прямое получение состояния объекта раскрывает приватные детали его реализации, нарушая инкапсуляцию.

Шаги реализации:

1. Определить класс создателя, объекты которого должны создавать снимки своего состояния.
2. Создать внутри создателя класс снимка и описать в нём все те же поля, которые имеются в оригинальном классе-создателе.
3. Сделать объекты снимков неизменяемыми. Они должны получать начальные значения только один раз, через свой конструктор.
4. Добавить в класс создателя методы получения и восстановления снимков.

Преимущества:

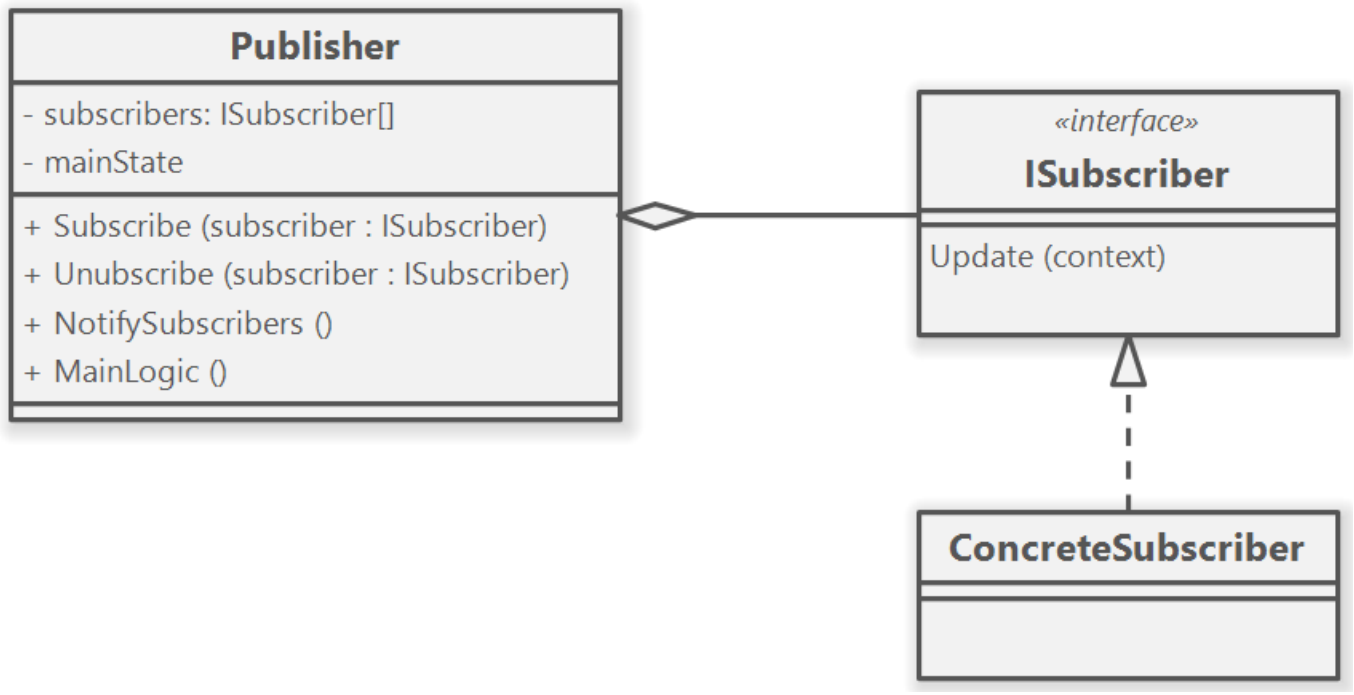
- Не нарушает инкапсуляции исходного объекта.
- Упрощает структуру исходного объекта. Ему не нужно хранить историю версий своего состояния.

Недостатки:

- Требует много памяти, если клиенты слишком часто создают снимки.
- Может повлечь дополнительные издержки памяти, если объекты, хранящие историю, не освобождают ресурсы, занятые устаревшими снимками.
- В некоторых языках (например, PHP, Python, JavaScript) сложно гарантировать, чтобы только исходный объект имел доступ к состоянию снимка.

Наблюдатель

Наблюдатель создаёт механизм позволяющий реагировать на события, происходящие в других объектах.



Применимость:

- Когда после изменения состояния одного объекта требуется что-то сделать в других, но вы не знаете наперёд, какие именно объекты должны отреагировать.
- Когда одни объекты должны наблюдать за другими, но только в определённых случаях.

Шаги реализации:

1. Создать интерфейс для подписчиков в котором будет метод оповещения.
2. Создать интерфейс издателя в котором будут методы управления подписками.
3. Реализовать интерфейсы.
4. Клиент добавляет подписчиков издателю - подписчики будут отслеживать изменения издателя.

Преимущества:

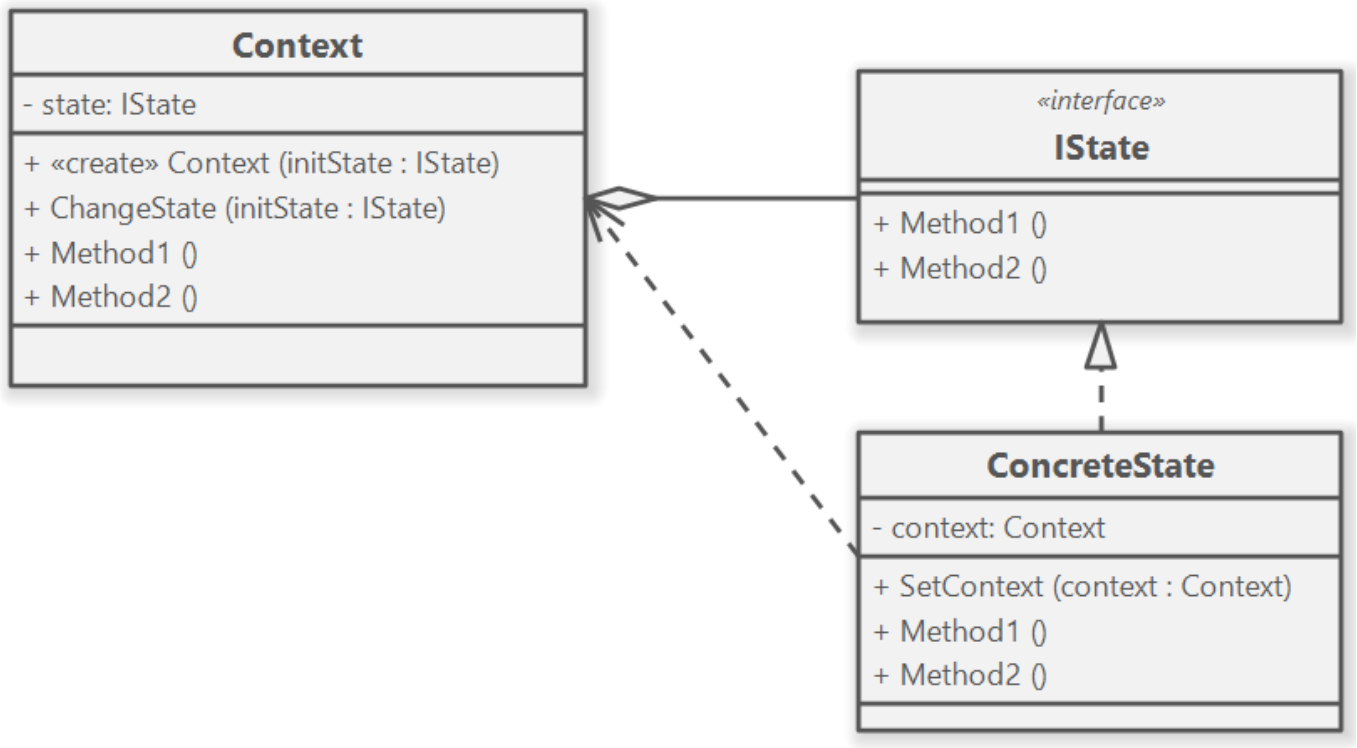
- Издатели не зависят от конкретных классов подписчиков и наоборот.
- Можно подписывать и отписывать получателей на лету.
- Реализует принцип открытости/закрытости.

Недостатки:

- Подписчики оповещаются в случайном порядке.

Состояние

Состояние позволяет объектам менять поведение в зависимости от их состояния.



Применимость:

- Когда у есть объект, поведение которого кардинально меняется в зависимости от внутреннего состояния, причём типов состояний много, и их код часто меняется.
- Когда код класса содержит множество больших, похожих друг на друга, условных операторов, которые выбирают поведения в зависимости от текущих значений полей класса.
- Когда сознательно используется табличная машина состояний, построенную на условных операторах, но приходится мириться с дублированием кода для похожих состояний и переходов.

Шаги реализации:

1. Определить интерфейс состояний в котором описаны методы общие для всех состояний методы.
2. Создать класс-контекст.
3. Реализовать интерфейс в классах-состояниях.
4. Создать в контексте поле для хранения состояния.
5. Заменить вызов кода зависящего от состояния на вызовы методов объекта-состояния.
6. Добавить код переключения состояний.

Преимущества:

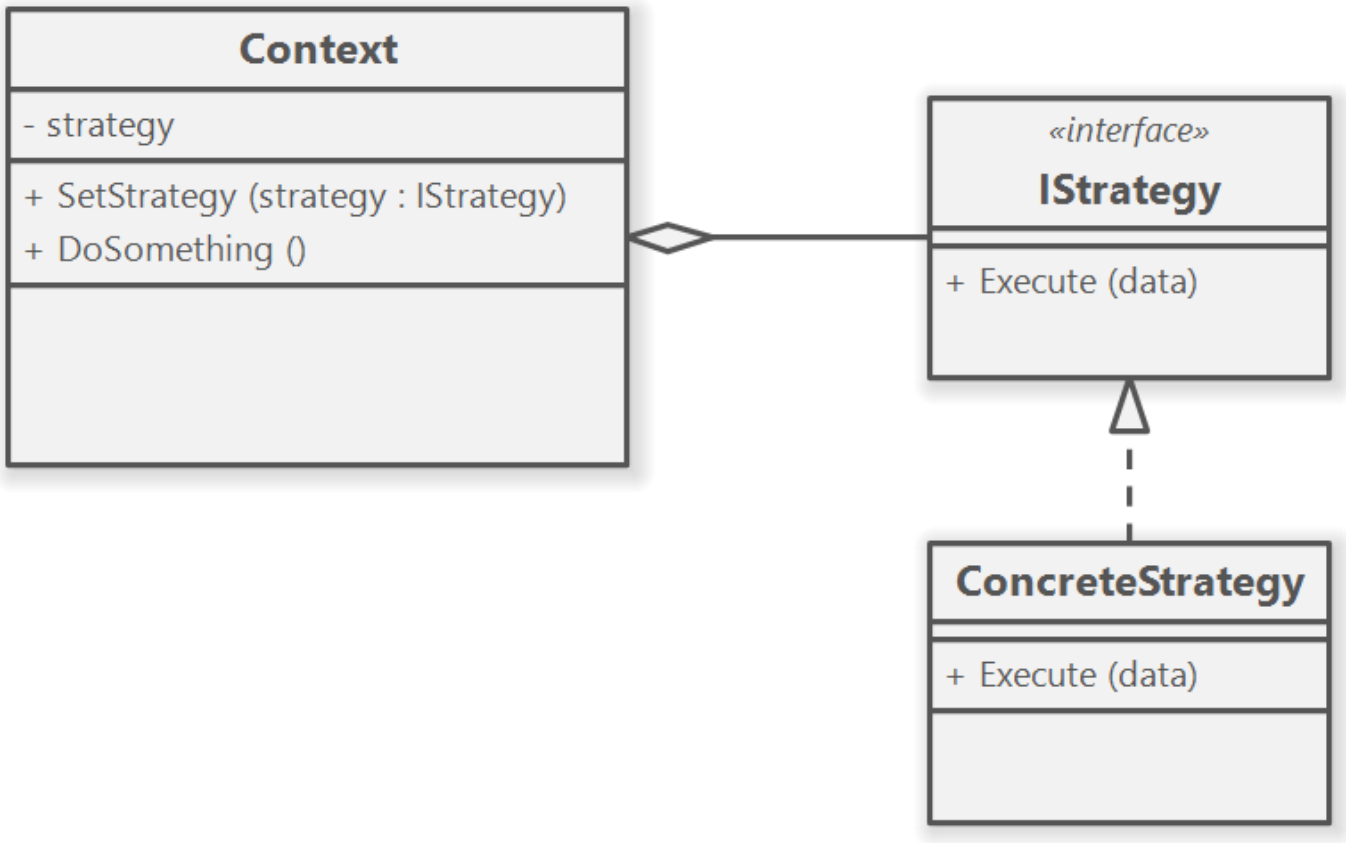
- Избавляет от множества больших условных операторов машины состояний.
- Концентрирует в одном месте код, связанный с определённым состоянием.
- Упрощает код контекста.

Недостатки:

- Может неоправданно усложнить код, если состояний мало и они редко меняются.

Стратегия

Стратегия определяет семейство схожих алгоритмов, позволяя менять их во время работы программы.



Применимость:

- Когда вам нужно использовать разные вариации какого-то алгоритма внутри одного объекта.
- Когда у вас есть множество похожих классов, отличающихся только некоторым поведением.
- Когда вы не хотите обнажать детали реализации алгоритмов для других классов.
- Когда различные вариации алгоритмов реализованы в виде развесистого условного оператора. Каждая ветка такого оператора представляет собой вариацию алгоритма.

Шаги реализации:

1. Создать интерфейс для стратегий.
2. Реализовать интерфейс в классах-стратегиями.
3. В классе-контексте создать ссылку на стратегию.
4. Клиентский код должен передавать в контекст нужную стратегию для получения нужного поведения.

Преимущества:

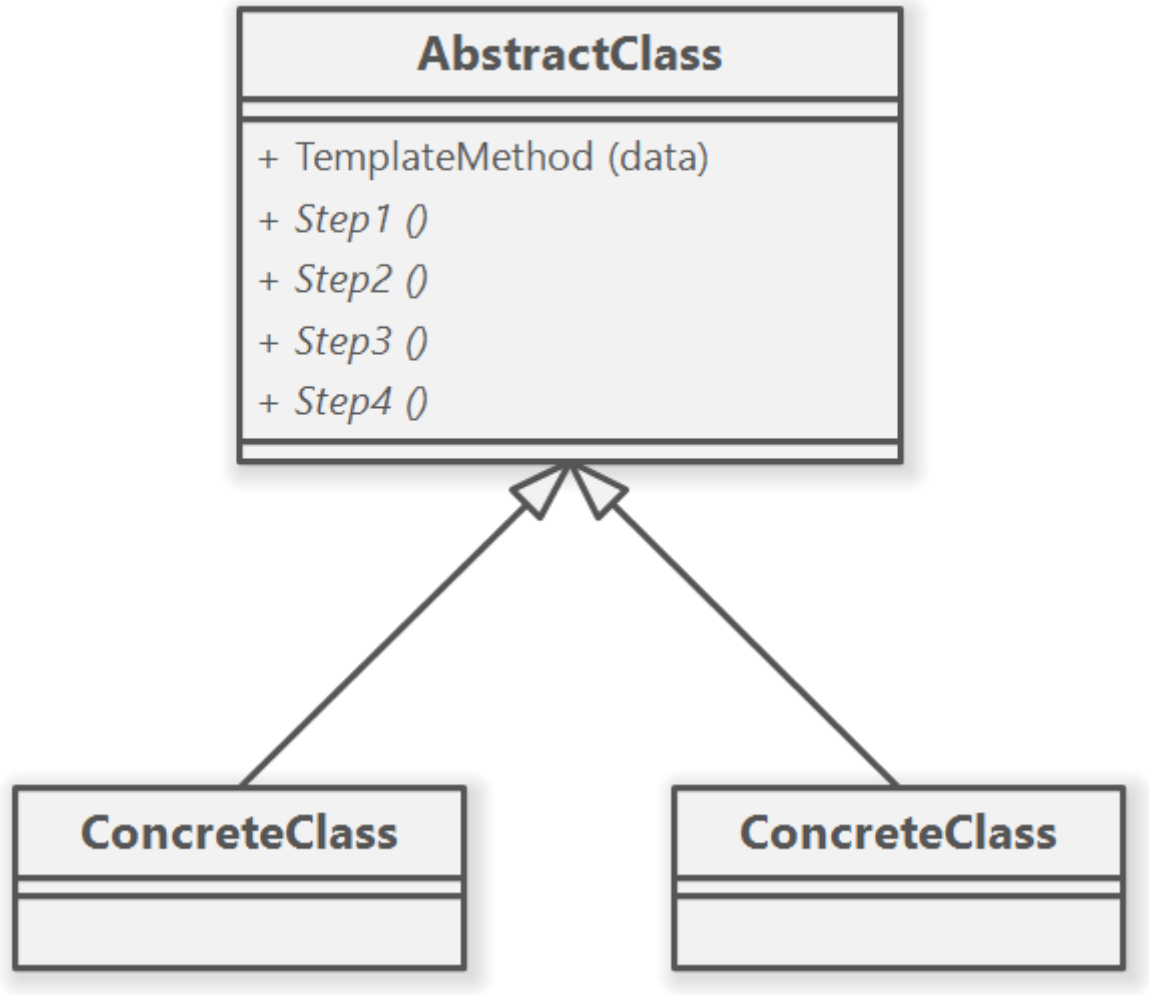
- Горячая замена алгоритмов на лету.
- Изолирует код и данные алгоритмов от остальных классов.
- Уход от наследования к делегированию.
- Реализует принцип открытости/закрытости.

Недостатки:

- Усложняет программу за счёт дополнительных классов.

- Клиент должен знать, в чём состоит разница между стратегиями, чтобы выбрать подходящую.

Шаблонный метод



Шаблонный метод определяет скелет алгоритма, позволяя переопределить шаги алгоритма.

Применимость:

- Когда подклассы должны расширять базовый алгоритм, не меняя его структуры.
- Когда у есть несколько классов, делающих одно и то же с незначительными отличиями.

Шаги реализации:

1. Разбить алгоритм на шаги.
2. Создать абстрактный класс в котором определён шаблонный метод, состоящий из вызовов шагов алгоритма.
3. Наследники абстрактного класса должны реализовать шаги шаблонного метода.

Преимущества:

- Облегчает повторное использование кода.

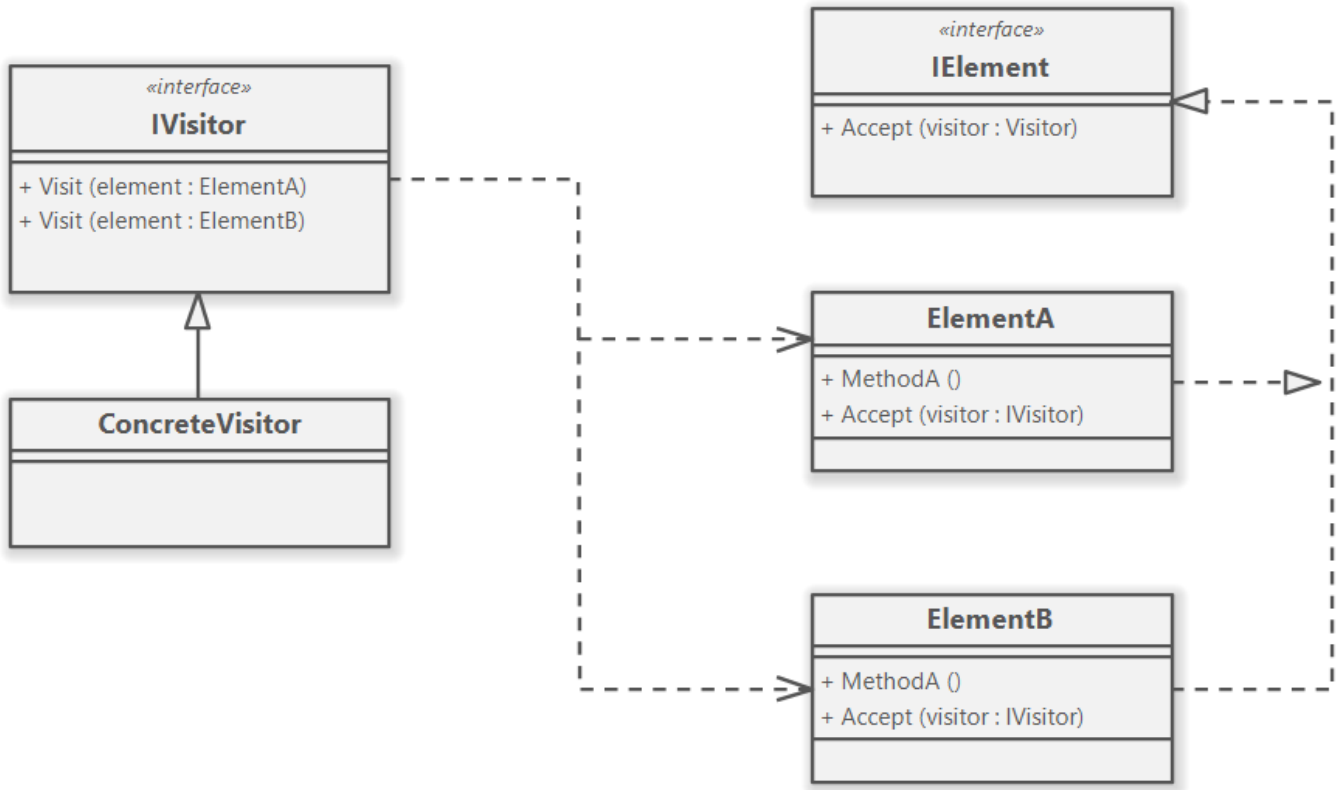
Недостатки:

- Жёсткое ограничение скелетом существующего алгоритма.

- Риск нарушить принцип подстановки Барбары Лисков, изменяя базовое поведение одного из шагов алгоритма через подкласс.
- С ростом количества шагов шаблонный метод становится слишком сложно поддерживать.

Посетитель

Посетитель позволяет добавлять в классы новые операции, не изменяя их.



Применимость:

- Когда нужно выполнить какую-то операцию над всеми элементами сложной структуры объектов, например, деревом.
- Когда над объектами сложной структуры объектов надо выполнять некоторые не связанные между собой операции, но не хочется «засорять» классы такими операциями.
- Когда новое поведение имеет смысл только для некоторых классов из существующей иерархии.

Шаги реализации:

1. Создать интерфейс посетителя, в котором определить методы посещения.
2. Описать интерфейс элементов в котором описан метод принятия посетителей.
3. Реализовать методы принятия в которых идёт переадресация методам посетителя в котором тип параметра совпадает с типом текущего объекта.
4. Для каждого нового поведения создать отдельный класс-посетитель.

Преимущества:

- Упрощает добавление операций, работающих со сложными структурами объектов.
- Объединяет родственные операции в одном классе.
- Посетитель может накапливать состояние при обходе структуры элементов.

Недостатки:

- Паттерн не оправдан, если иерархия элементов часто меняется.
- Может привести к нарушению инкапсуляции элементов.