Курс лекций:

Язык программирования

C++Лекция 14: Шаблоны проектирования: Поведенческие.

Преподаватель: Пысин Максим Дмитриевич, Краснов Дмитрий Олегович, аспиранты кафедры ИКТ.

Что запомнилось?

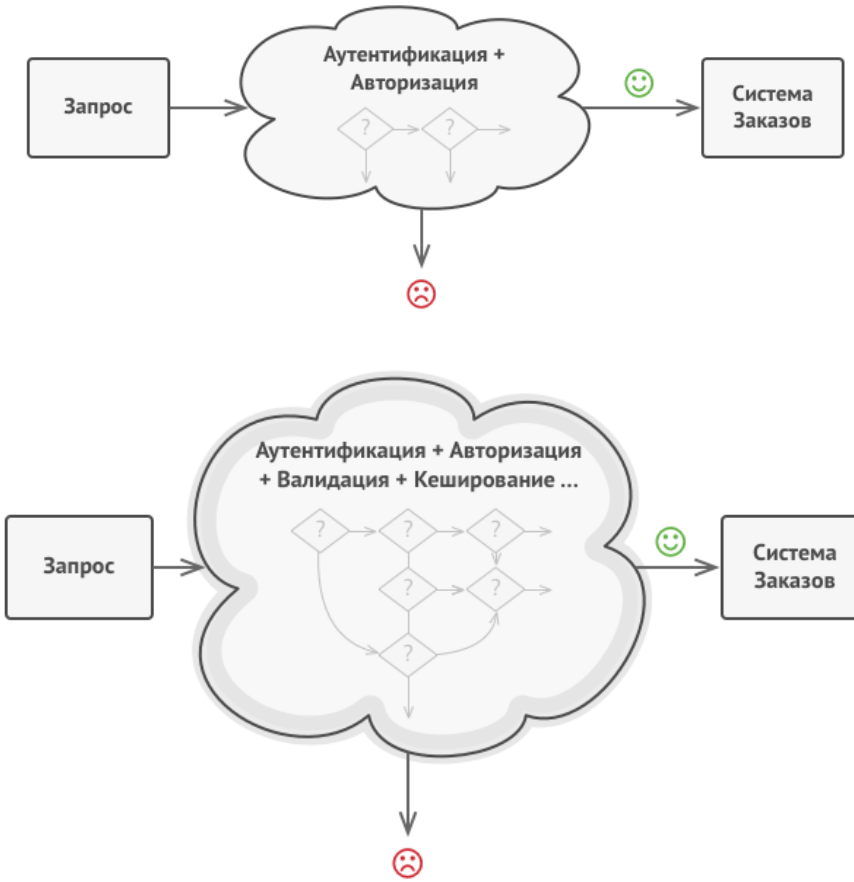
- **Что такое парадигма программирования? Какие парадигмы программирования есть? Какую изучаем мы?**
- **Абстрагирование и абстракция что это и для чего?**
- **Что такое класс? Объект? Наследования? Полиморфизм? Инкапсуляции?**
- **Что такое лямбда функция? Что такое захват[проброс] переменных?**
- **Что такое абстрактный класс? Для чего он нужен?**
- **Что такое итератор и для чего он используется?**
- **Что такое последовательные контейнеры? Что такое ассоциативные контейнеры? На чем основаны?**
- **С чем в основном работает стандартная библиотека алгоритмов?**
- **Что такое поток ввода вывода? Примеры?**
- **Что такое исключение? Как ловить? Как вызывать?**
- **Что такое паттерны проектирования?**
- **Чем занимаются порождающие паттерны?**
- **Виды отношений между классами?**

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.

Проблема

С каждой новой «фичей» код проверок, выглядящий как большой клубок условных операторов, всё больше и больше раздувался. При изменении одного правила приходилось трогать код всех проверок. А для того, чтобы применить проверки к другим ресурсам, пришлось продублировать их код в других классах.



Обработчик определяет общий для всех конкретных обработчиков интерфейс.
Базовый обработчик — опциональный класс, который позволяет избавиться от дублирования одного и того же кода во всех конкретных обработчиках.
Конкретные обработчики содержат код обработки запросов. При получении запроса каждый обработчик решает, может ли он обработать запрос, а также стоит ли передать его следующему объекту.

Преимущества

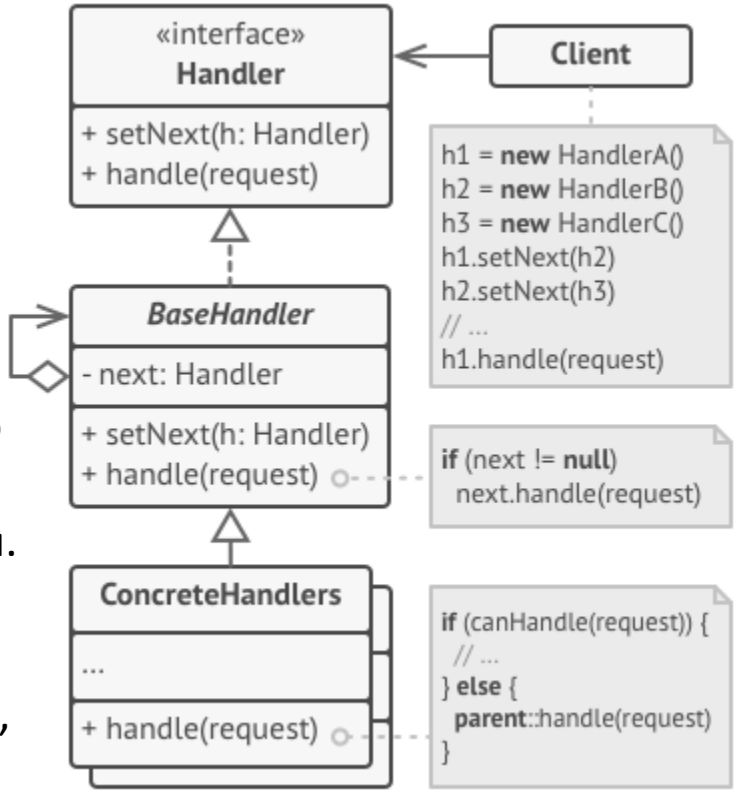
- Уменьшает зависимость между клиентом и обработчиками.
- Реализует принцип единственной обязанности.
- Реализует принцип открытости/закрытости.

Недостатки

- Запрос может остаться никем не обработанным.

Когда использовать?

- 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.

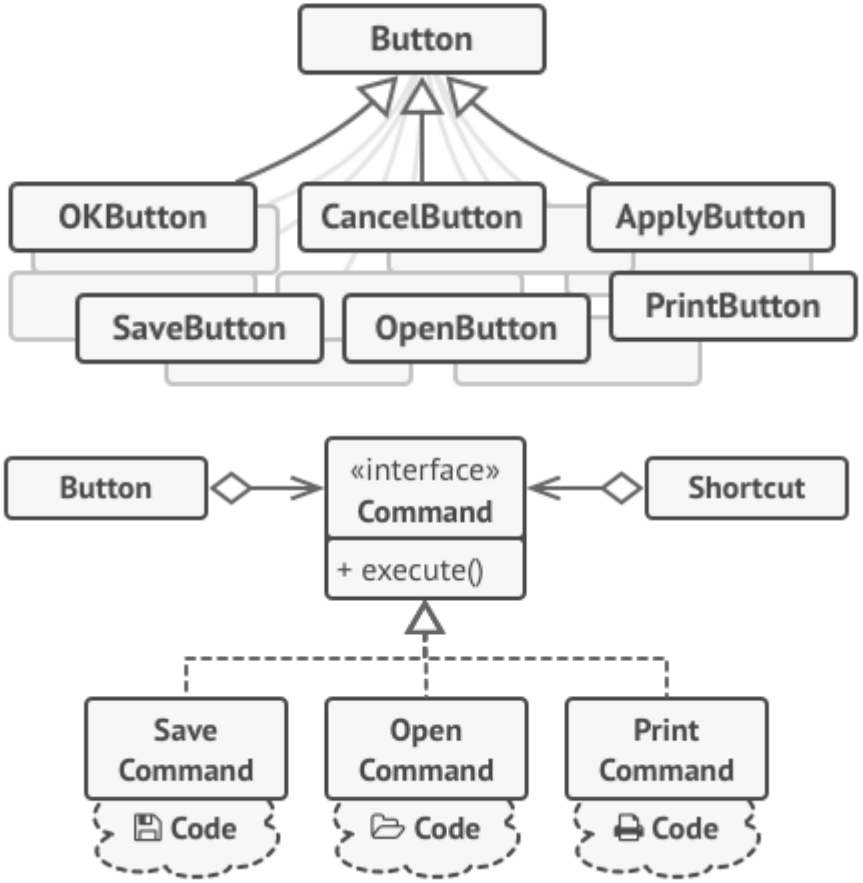


Команда.

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.

Проблема:

Представьте, что вы работаете над программой текстового редактора. Дело как раз подошло к разработке панели управления. Вы создали класс красивых Кнопок и хотите использовать его для всех кнопок приложения, начиная от панели управления, заканчивая простыми кнопками в диалогах.



Отправитель хранит ссылку на объект команды и обращается к нему, когда нужно выполнить какое-то действие.
Команда описывает общий для всех конкретных команд интерфейс.
Конкретные команды реализуют различные запросы, следуя общему интерфейсу команд.
Получатель содержит бизнес-логику программы.

Преимущества

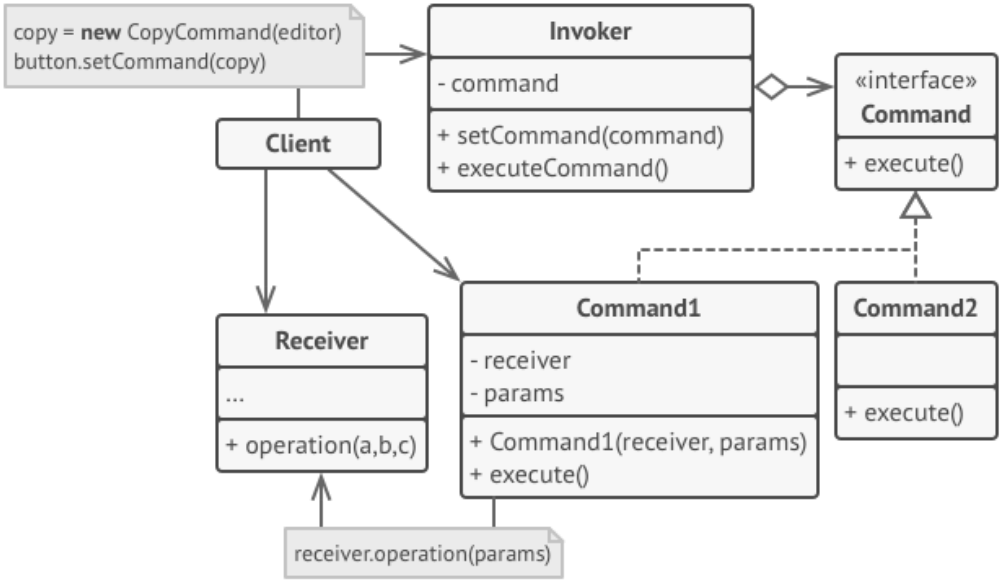
- Убирает прямую зависимость между объектами, вызывающими операции, и объектами, которые их непосредственно выполняют.
- Позволяет реализовать простую отмену и повтор операций.
- Позволяет реализовать отложенный запуск операций.
- Позволяет собирать сложные команды из простых.
- Реализует принцип открытости/закрытости.

Недостатки

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

Когда использовать?

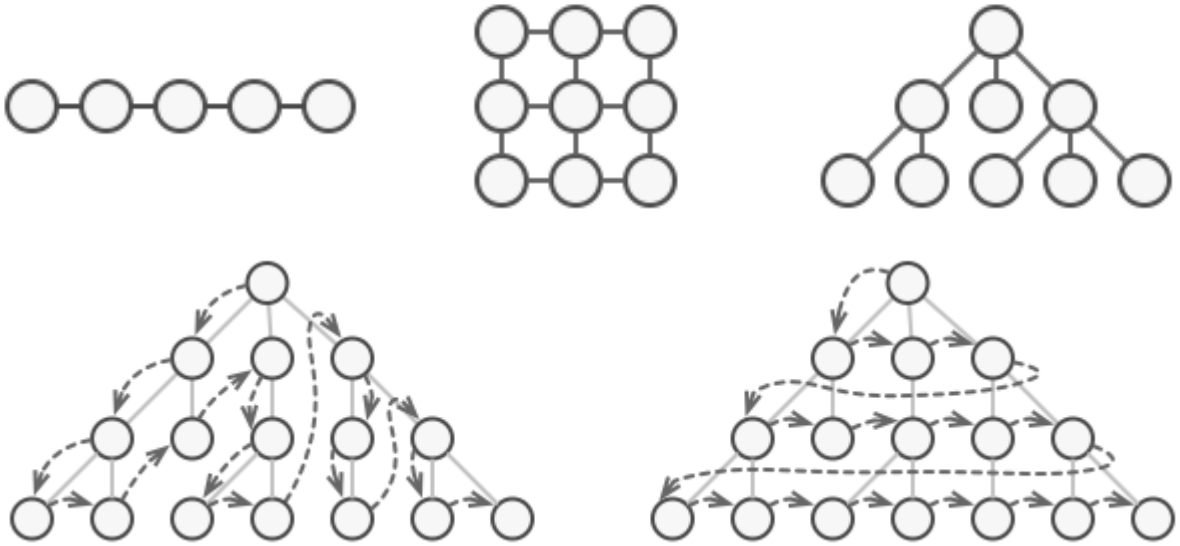
- Когда вы хотите параметризовать объекты выполняемым действием.
- Когда вы хотите ставить операции в очередь, выполнять их по расписанию или передавать по сети.
- Когда вам нужна операция отмены.



Итератор.

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

Проблема:
Но каким способом следует перемещаться по сложной структуре данных? Например, сегодня может быть достаточным обход дерева в глубину, но завтра потребуется возможность перемещаться по дереву в ширину. А на следующей неделе и того хуже — понадобится обход коллекции в случайном порядке.



Итератор описывает интерфейс для доступа и обхода элементов коллекции.
Конкретный итератор реализует алгоритм обхода какой-то конкретной коллекции.
Коллекция описывает интерфейс получения итератора из коллекции.
Конкретная коллекция возвращает новый экземпляр определённого конкретного итератора, связав его с текущим объектом коллекции.

Преимущества

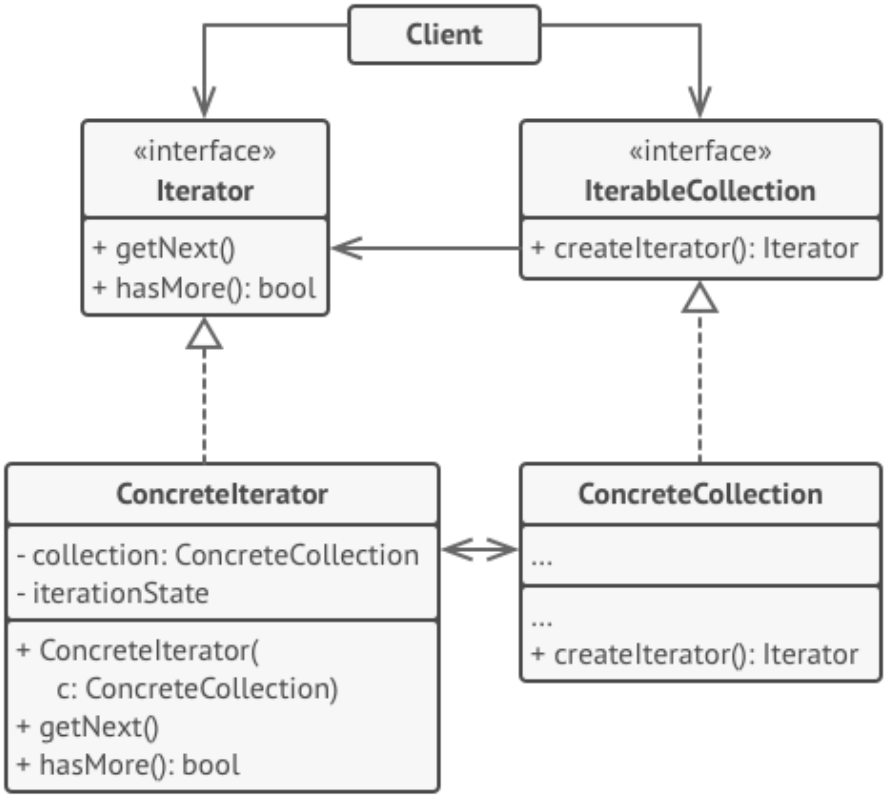
- Упрощает классы хранения данных.
- Позволяет реализовать различные способы обхода структуры данных.
- Позволяет одновременно перемещаться по структуре данных в разные стороны.

Недостатки

- Не оправдан, если можно обойтись простым циклом.

Когда использовать?

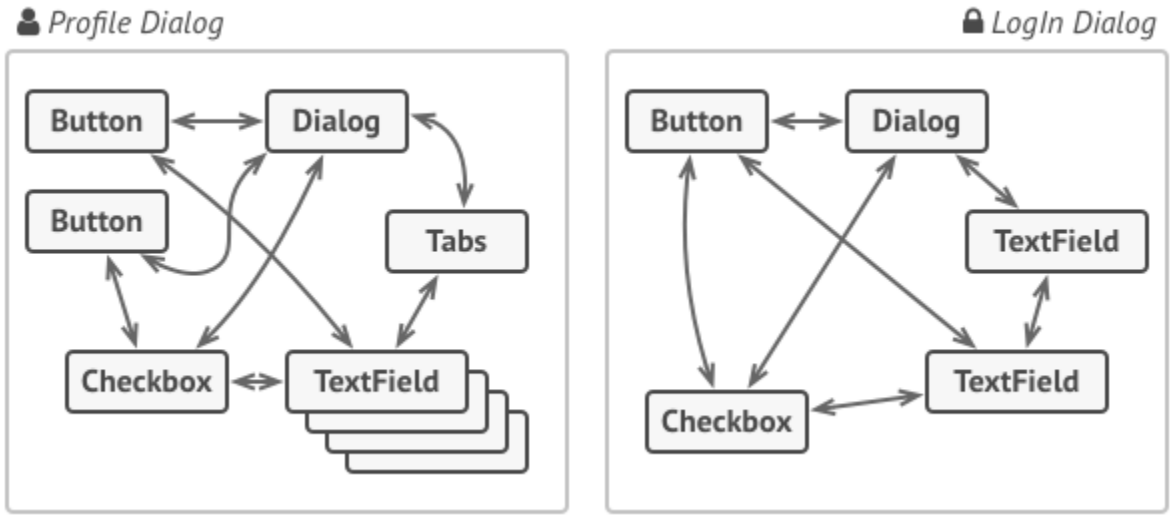
- Когда у вас есть сложная структура данных, и вы хотите скрыть от клиента детали её реализации (из-за сложности или вопросов безопасности).
- Когда вам нужно иметь несколько вариантов обхода одной и той же структуры данных.
- Когда вам хочется иметь единый интерфейс обхода различных структур данных.



Посредник.

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

Проблема:
Предположим, что у вас есть диалог создания профиля пользователя. Он состоит из всевозможных элементов управления — текстовых полей, чекбоксов, кнопок. Отдельные элементы диалога должны взаимодействовать друг с другом. Так, например, чекбокс «у меня есть собака» открывает скрытое поле для ввода имени домашнего любимца, а клик по кнопке отправки запускает проверку значений всех полей формы.



Компоненты — это разнородные объекты, содержащие бизнес-логику программы.
Посредник определяет интерфейс для обмена информацией с компонентами.
Конкретный посредник содержит код взаимодействия нескольких компонентов между собой.

Преимущества

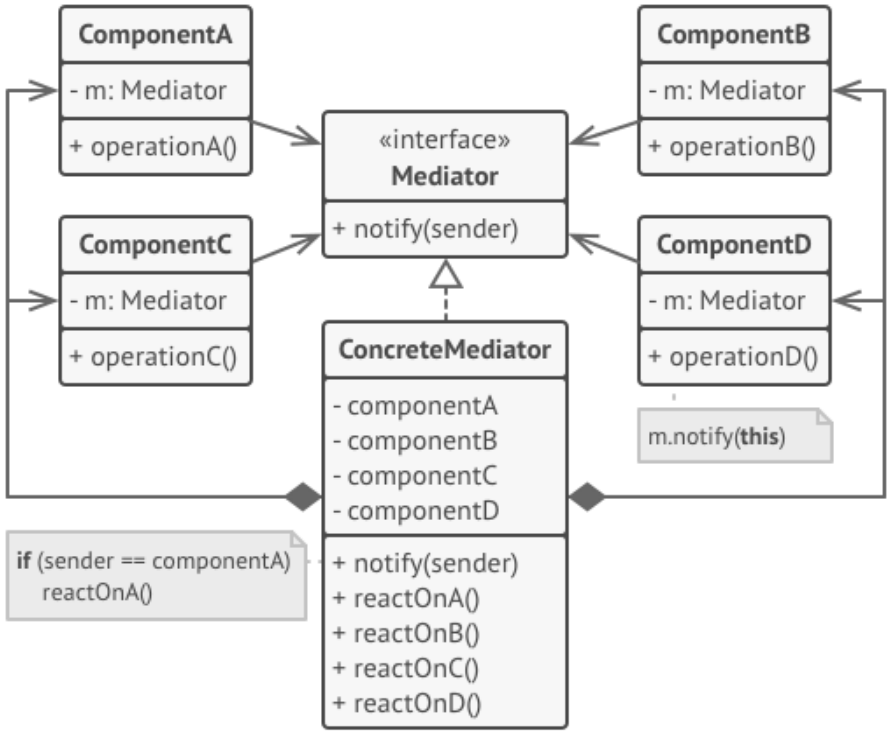
- Устраняет зависимости между компонентами, позволяя повторно их использовать.
- Упрощает взаимодействие между компонентами.
- Централизует управление в одном месте.

Недостатки

- Посредник может сильно раздуться.

Когда использовать?

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



Снимок.

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

Проблема

Предположим, что вы пишете программу текстового редактора. Помимо обычного редактирования, ваш редактор позволяет менять форматирование текста, вставлять картинки и прочее.

В какой-то момент вы решили сделать все эти действия отменяемыми. Для этого вам нужно сохранять текущее состояние редактора перед тем, как выполнить любое действие.

Создатель может производить снимки своего состояния, а также воспроизводить прошлое состояние, если подать в него готовый снимок.

Снимок — это простой объект данных, содержащий состояние создателя.

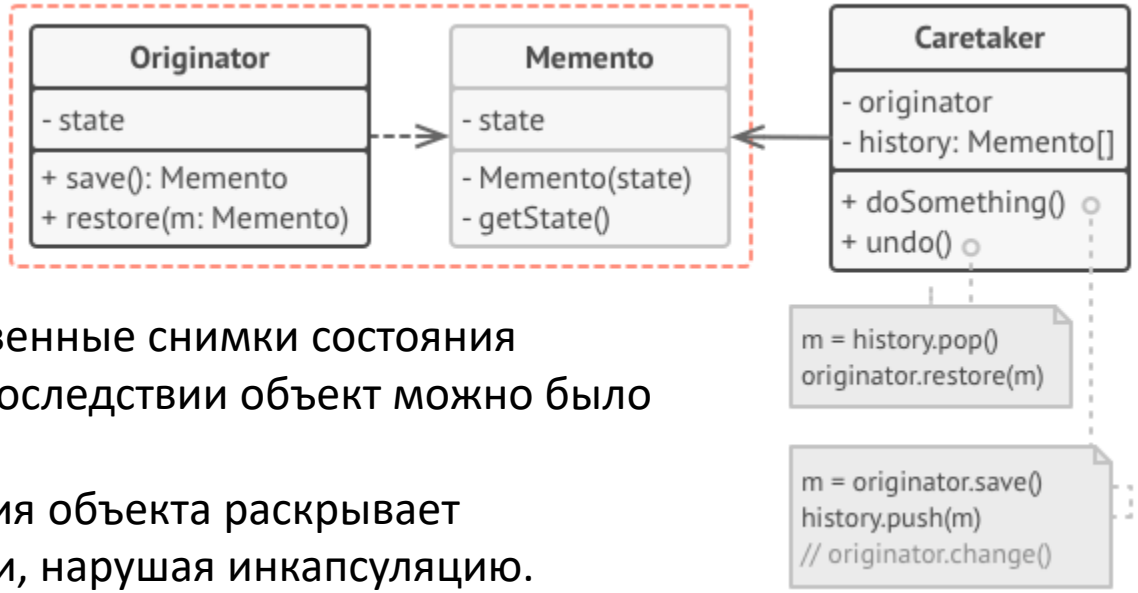
Опекун должен знать, когда делать снимок создателя и когда его нужно восстанавливать.

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.



Когда использовать?

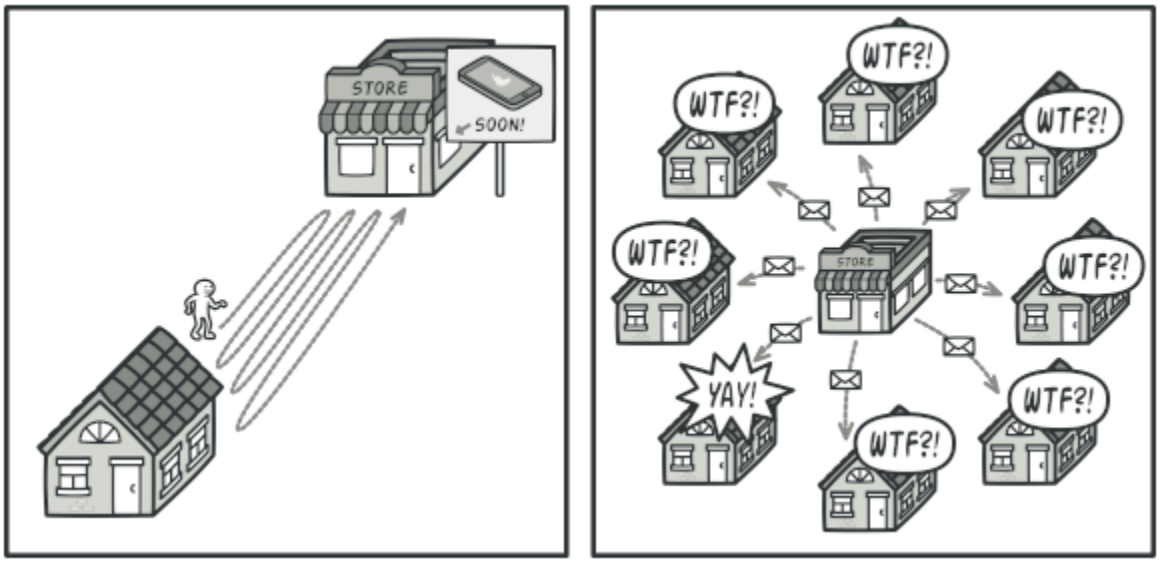
- Когда вам нужно сохранять мгновенные снимки состояния объекта (или его части), чтобы впоследствии объект можно было восстановить в том же состоянии.
- Когда прямое получение состояния объекта раскрывает приватные детали его реализации, нарушая инкапсуляцию.

Наблюдать.

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

Проблема:
Представьте, что вы имеете два объекта: Покупатель и Магазин. В магазин вот-вот должны завезти новый товар, который интересен покупателю.

Покупатель может каждый день ходить в магазин, чтобы проверить наличие товара. Но при этом он будет злиться, без толку тратя своё драгоценное время. С другой стороны, магазин может разослать спам каждому своему покупателю. Многих это расстроит, так как товар специфический, и не всем он нужен.

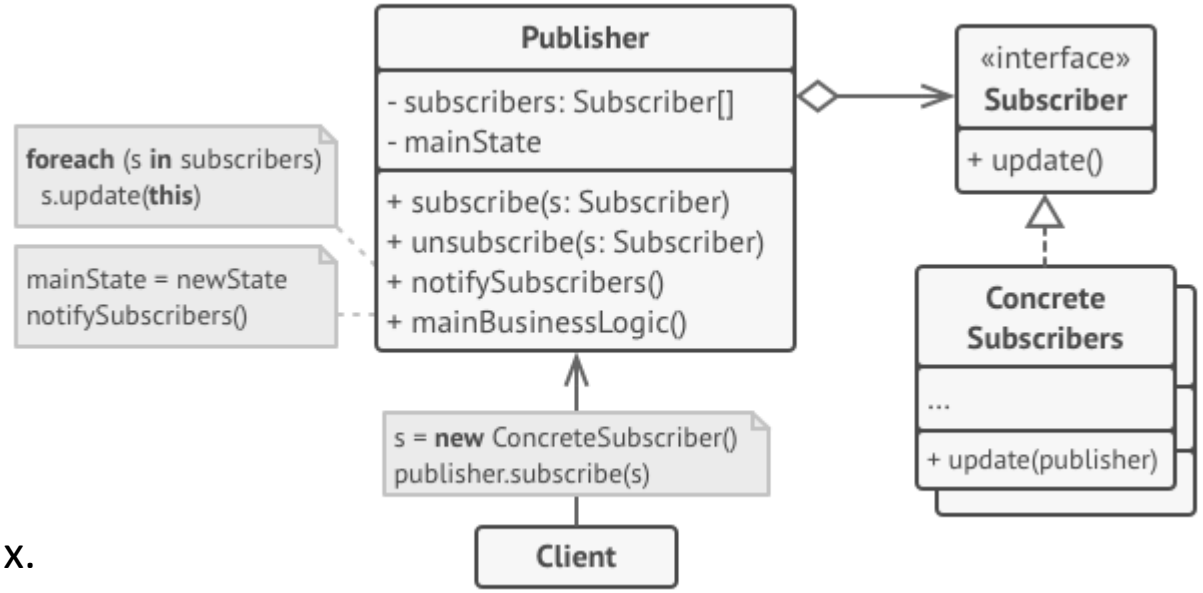


Издатель владеет внутренним состоянием, изменение которого интересно отслеживать подписчикам.
Подписчик определяет интерфейс, которым пользуется издатель для отправки оповещения.
Конкретные подписчики выполняют что-то в ответ на оповещение, пришедшее от издателя.

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

Недостатки
- Подписчики оповещаются в случайном порядке.

Когда использовать?
- 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.

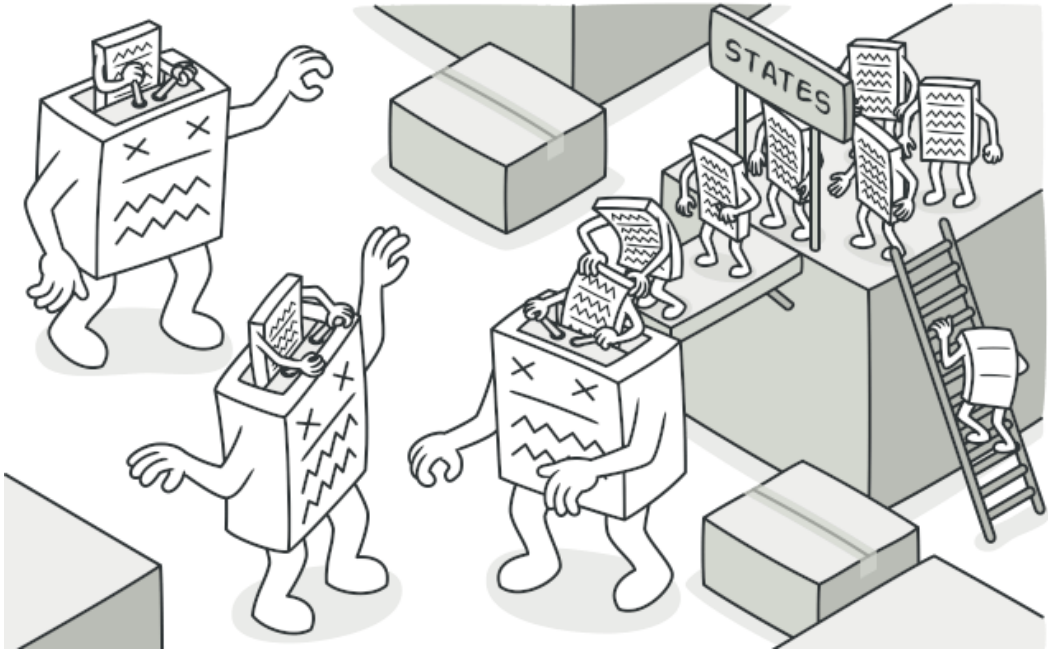


Состояние.

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

Основная идея в том, что программа может находиться в одном из нескольких состояний, которые всё время сменяют друг друга. Набор этих состояний, а также переходов между ними, предопределён и конечен. Находясь в разных состояниях, программа может по-разному реагировать на одни и те же события, которые происходят с ней.

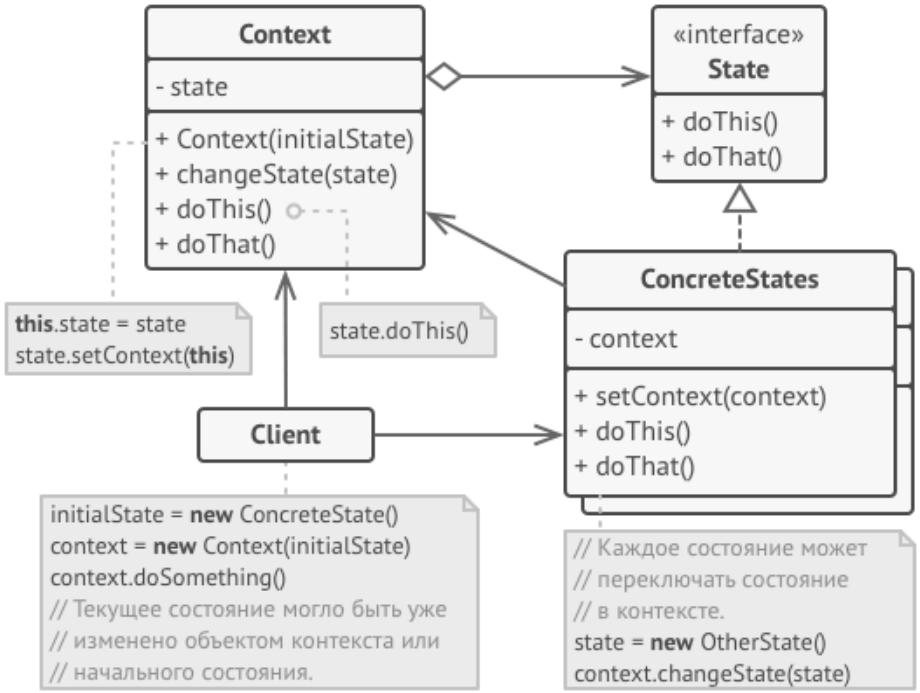
Такой подход можно применить и к отдельным объектам. Например, объект Документ может принимать три состояния: Черновик, Модерация или Опубликован. В каждом из этих состоянии метод опубликовать будет работать по-разному:
Из черновика он отправит документ на модерацию.
Из модерации — в публикацию, но при условии, что это сделал администратор.
В опубликованном состоянии метод не будет делать ничего.



Контекст хранит ссылку на объект состояния и делегирует ему часть работы, зависящей от состояний.

Состояние описывает общий интерфейс для всех конкретных состояний.

Конкретные состояния реализуют поведения, связанные с определённым состоянием контекста.



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.

Когда использовать?

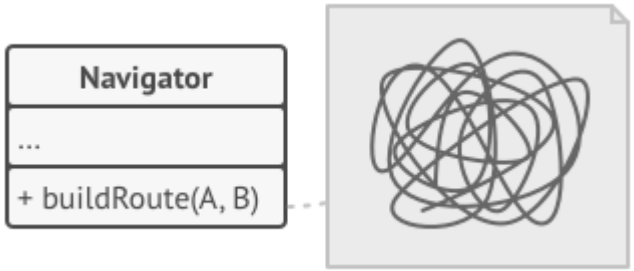
- Когда у вас есть объект, поведение которого кардинально меняется в зависимости от внутреннего состояния, причём типов состояний много, и их код часто меняется.
- Когда код класса содержит множество больших, похожих друг на друга, условных операторов, которые выбирают поведения в зависимости от текущих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

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.

Проблема
Вы решили написать приложение-навигатор для путешественников. Оно должно показывать красивую и удобную карту, позволяющую с лёгкостью ориентироваться в незнакомом городе.

Одной из самых востребованных функций являлся поиск и прокладывание маршрутов. Пребывая в неизвестном ему городе, пользователь должен иметь возможность указать начальную точку и пункт назначения, а навигатор — проложит оптимальный путь.



Контекст хранит ссылку на объект конкретной стратегии, работая с ним через общий интерфейс стратегий.

Стратегия определяет интерфейс, общий для всех вариаций алгоритма. Контекст использует этот интерфейс для вызова алгоритма.

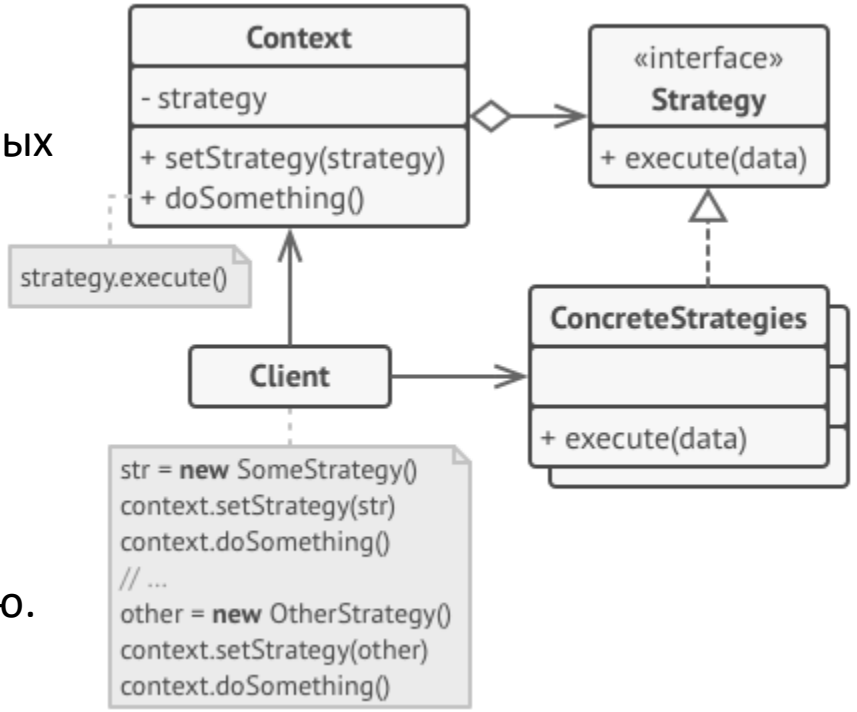
Конкретные стратегии реализуют различные вариации алгоритма.

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.



Когда использовать?

- 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.

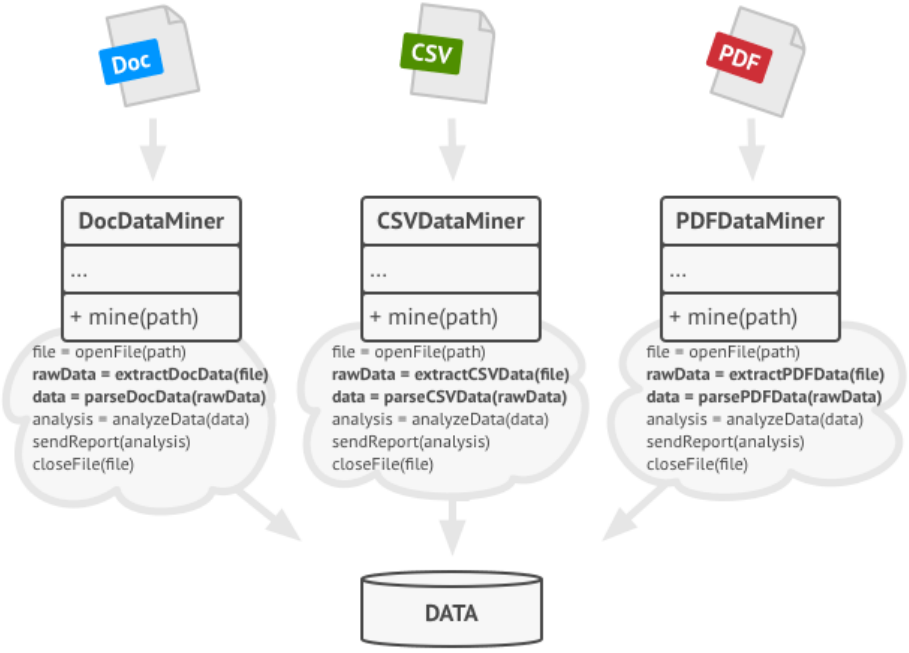
Шаблонный метод.

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.

Проблема

Вы пишете программу для дата-майнинга в офисных документах. Пользователи будут загружать в неё документы в разных форматах (PDF, DOC, CSV), а программа должна извлекать из них полезную информацию.

В первой версии вы ограничились только обработкой DOC-файлов. В следующей версии добавили поддержку CSV. А через месяц прикрутили работу с PDF-документами.



Абстрактный класс определяет шаги алгоритма и содержит шаблонный метод, состоящий из вызовов этих шагов.
Конкретный класс переопределяет некоторые (или все) шаги алгоритма. Конкретные классы не переопределяют сам шаблонный метод.

Преимущества

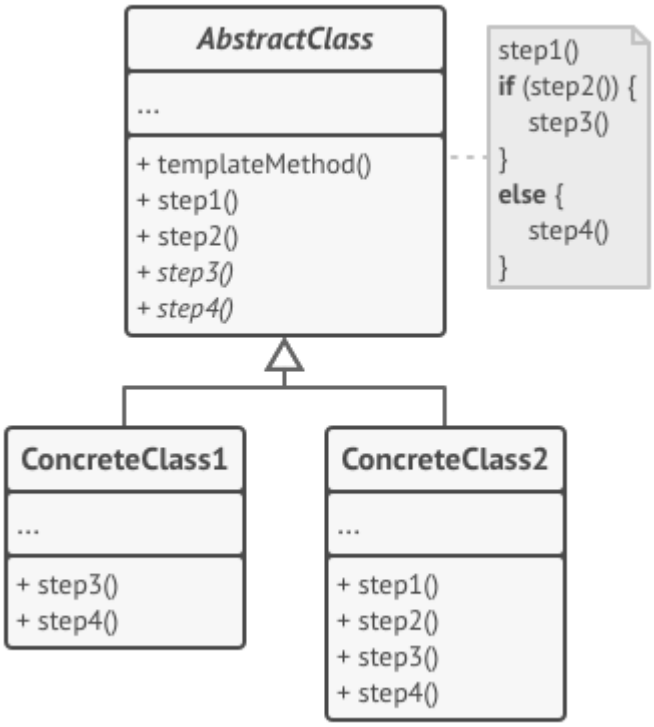
- Облегчает повторное использование кода.

Недостатки

- Вы жёстко ограничены скелетом существующего алгоритма.
- Вы можете нарушить принцип подстановки Барбары Лисков, изменяя базовое поведение одного из шагов алгоритма через подкласс.
- С ростом количества шагов шаблонный метод становится слишком сложно поддерживать.

Когда использовать?

- Когда подклассы должны расширять базовый алгоритм, не меняя его структуры.
- Когда у вас есть несколько классов, делающих одно и то же с незначительными отличиями. Если вы редактируете один класс, то приходится вносить такие же правки и в остальные классы.



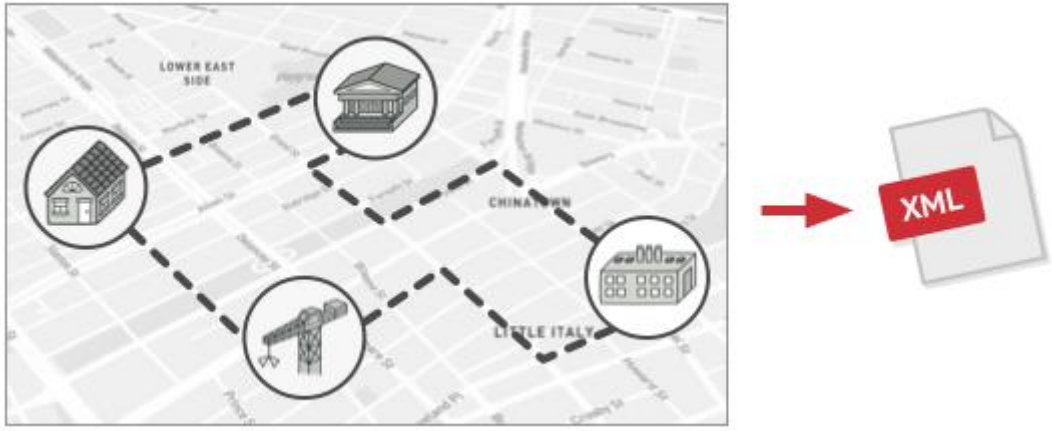
Посетитель.

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

Проблема

Ваша команда разрабатывает приложение, работающее с геоданными в виде графа. Ваша задача — сделать экспорт этого графа в XML. Дело было бы плёвым, если бы вы могли редактировать классы узлов. Достаточно было бы добавить метод экспорта в каждый тип узла, а затем, перебирая узлы графа, вызывать этот метод для каждого узла. Благодаря полиморфизму, решение получилось бы изящным, так как вам не пришлось бы привязываться к конкретным классам узлов.

Но, к сожалению, классы узлов вам изменить не удалось. Системный архитектор сослался на то, что код классов узлов сейчас очень стабилен, и от него многое зависит, поэтому он не хочет рисковать и позволять кому-либо его трогать.



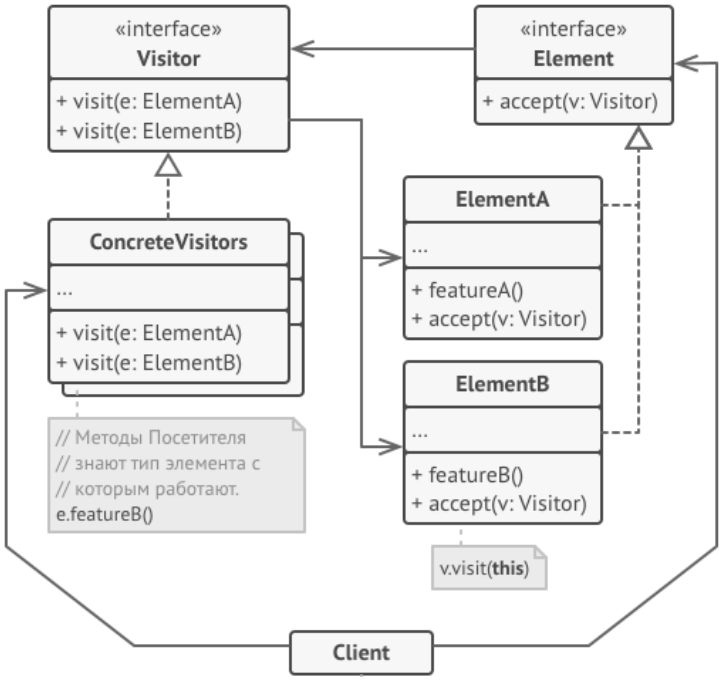
Посетитель описывает общий интерфейс для всех типов посетителей.
Конкретные посетители реализуют какое-то особенное поведение для всех типов элементов, которые можно подать через методы интерфейса посетителя.
Элемент описывает метод принятия посетителя.
Конкретные элементы реализуют методы принятия посетителя.

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.



Когда использовать?

Когда вам 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 вы не хотите «засорять» классы такими операциями.
Когда новое поведение имеет смысл только для некоторых классов из существующей иерархии.