Курс лекций:

Язык программирования

C++Лекция 13:
Шаблоны проектирования:
Структурные.

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

Что запомнилось?

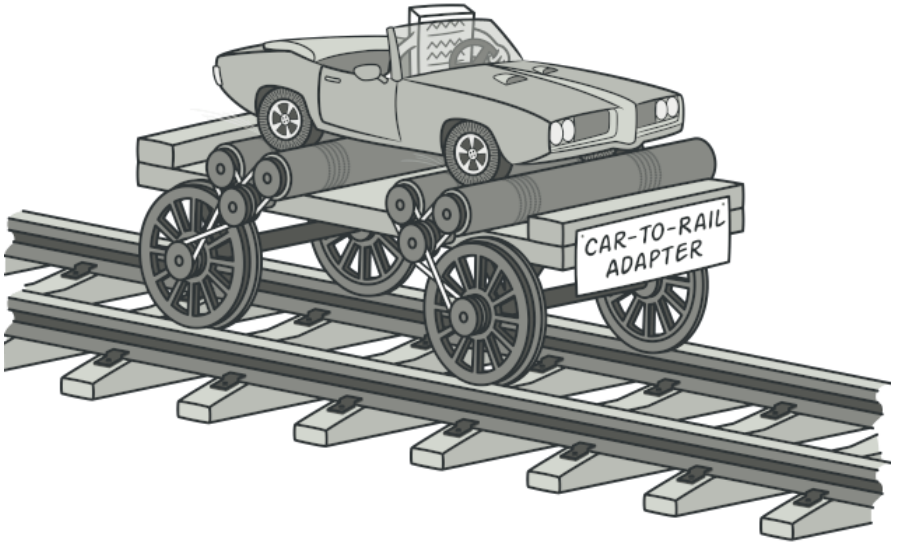
- **Что такое парадигма программирования? Какие парадигмы программирования есть? Какую изучаем мы?**
- **Абстрагирование и абстракция что это и для чего?**
- **Что такое класс? Объект? Наследования? Полиморфизм? Инкапсуляции?**
- **Что такое лямбда функция? Что такое захват[проброс] переменных?**
- **Что 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?**

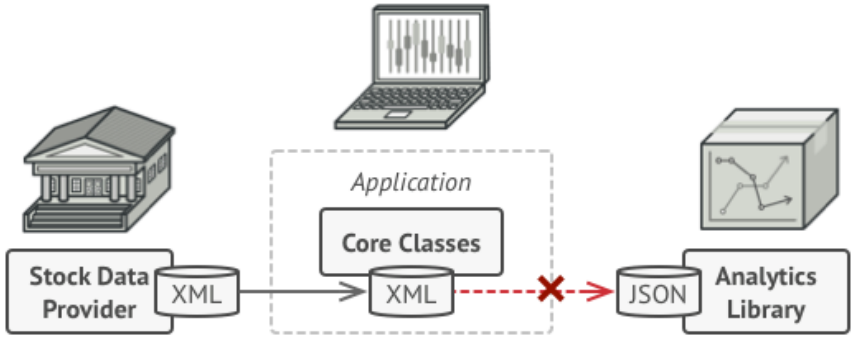
Адаптер

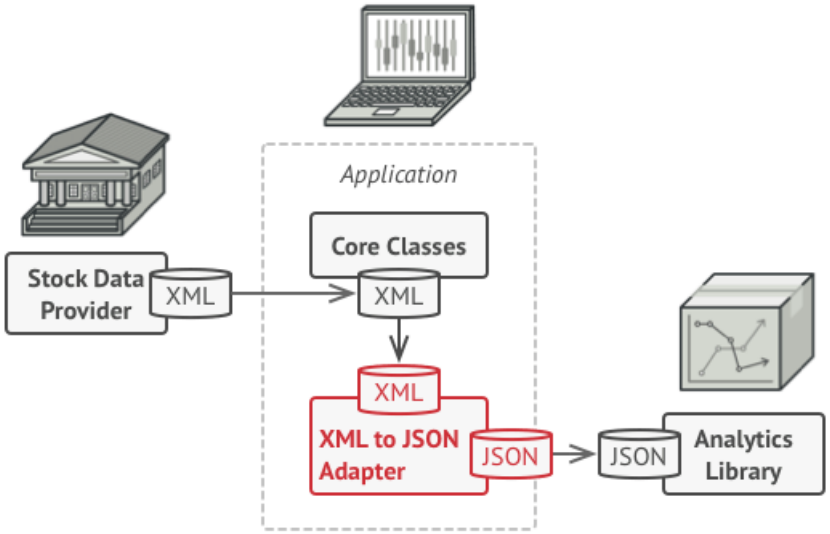
Адаптер — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.

Представьте, что вы делаете приложение для торговли на бирже. Ваше приложение скачивает биржевые котировки из нескольких источников в XML, а затем рисует красивые графики.

В какой-то момент вы решаете улучшить приложение, применив стороннюю библиотеку аналитики. Но вот беда — библиотека поддерживает только формат данных JSON, несовместимый с вашим приложением.







Клиентский интерфейс описывает протокол, через который клиент может работать с другими классами.
Сервис – это какой-то полезный класс, обычно сторонний.
Адаптер — это класс, который может одновременно работать и с клиентом, и с сервисом. Работая с адаптером через интерфейс, клиент не привязывается к конкретному классу адаптера. Благодаря этому, вы можете добавлять в программу новые виды адаптеров, независимо от клиентского кода. Это может пригодиться, если интерфейс сервиса вдруг изменится, например, после выхода новой версии сторонней библиотеки.

Преемущества

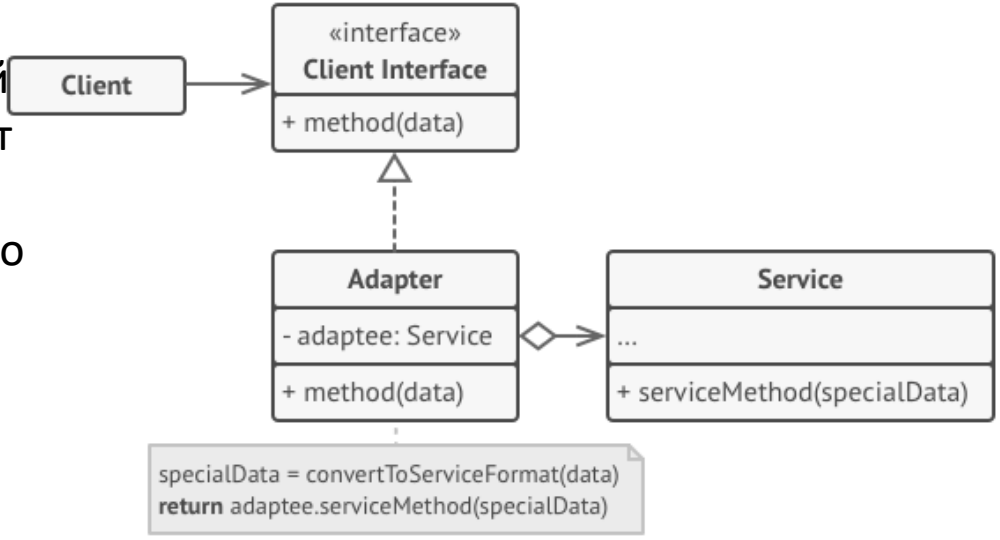
- Отделяет и скрывает от клиента подробности преобразования различных интерфейсов.

Недостатки

- Усложняет код программы из-за введения дополнительных классов.

Когда использовать?

- Когда вы хотите использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения.
- Когда вам нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности, причём расширить суперкласс вы не можете.

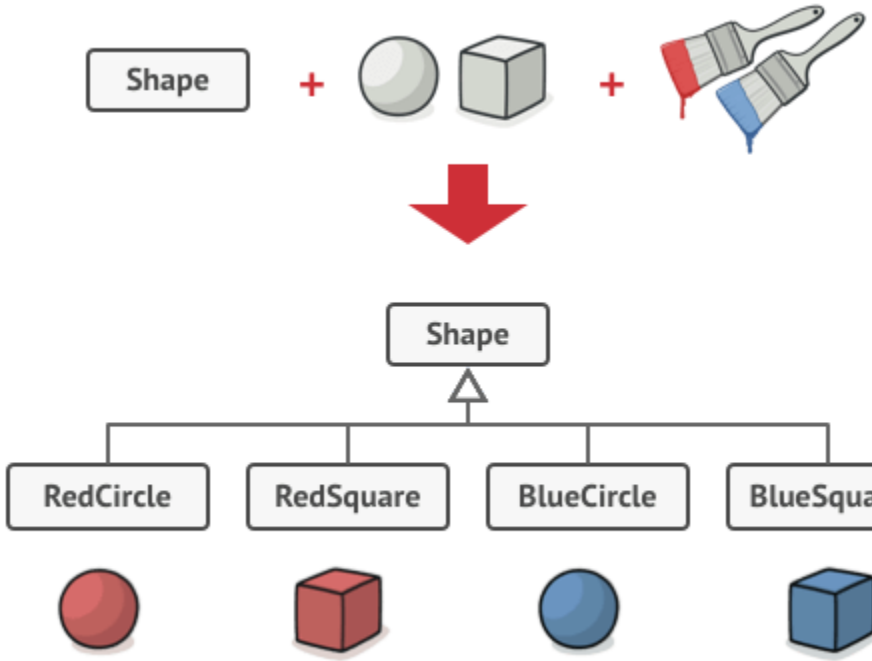


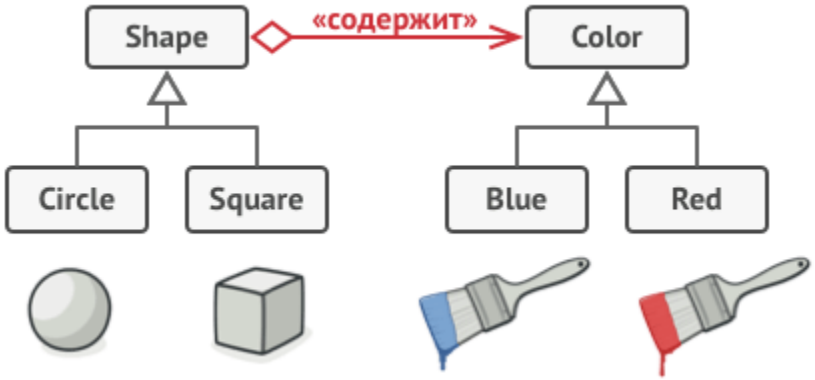
Мост

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.

При добавлении новых видов фигур и цветов количество комбинаций будет расти в геометрической прогрессии.

Корень проблемы заключается в том, что мы пытаемся расширить классы фигур сразу в двух независимых плоскостях — по виду и по цвету. Именно это приводит к разрастанию дерева классов.





Абстракция содержит управляющую логику. Код абстракции делегирует реальную работу связанному объекту реализации.
Реализация задаёт общий интерфейс для всех реализаций. Все методы, которые здесь описаны, будут доступны из класса абстракции и его подклассов.
Конкретные реализации содержат платформо-зависимый код.
Расширенные абстракции содержат различные вариации управляющей логики. Как и родитель, работает с реализациями только через общий интерфейс реализации.

Преимущества

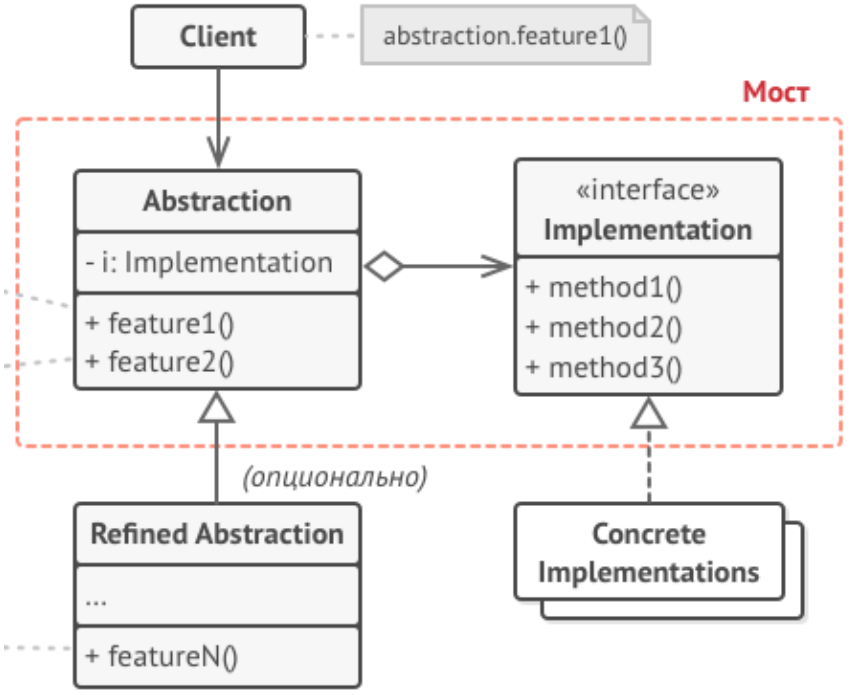
- Позволяет строить платформо-независимые программы.
- Скрывает лишние или опасные детали реализации от клиентского кода.
- Реализует *принцип открытости/закрытости*.

Недостатки

- Усложняет код программы из-за введения дополнительных классов.

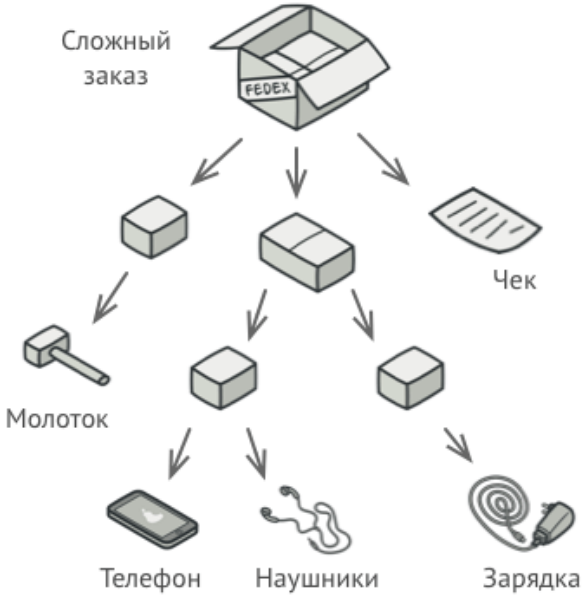
Когда использовать?

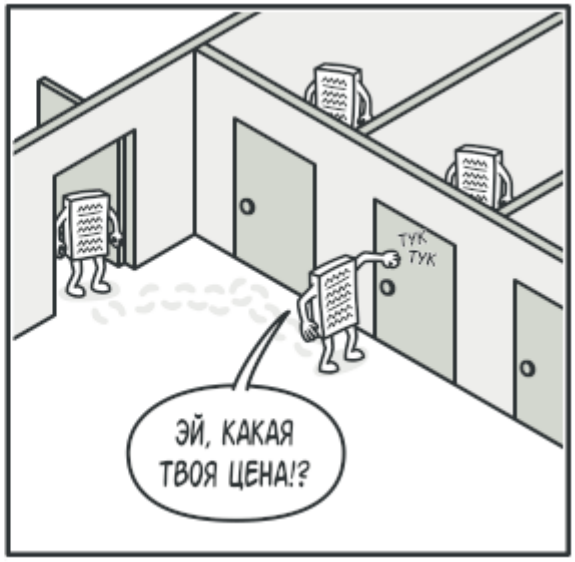
- Когда вы хотите разделить монолитный класс, который содержит несколько различных реализаций какой-то функциональности (например, если класс может работать с разными системами баз данных).
- Когда класс нужно расширять в двух независимых плоскостях.
- Когда вы хотите, чтобы реализацию можно было бы изменять во время выполнения программы.



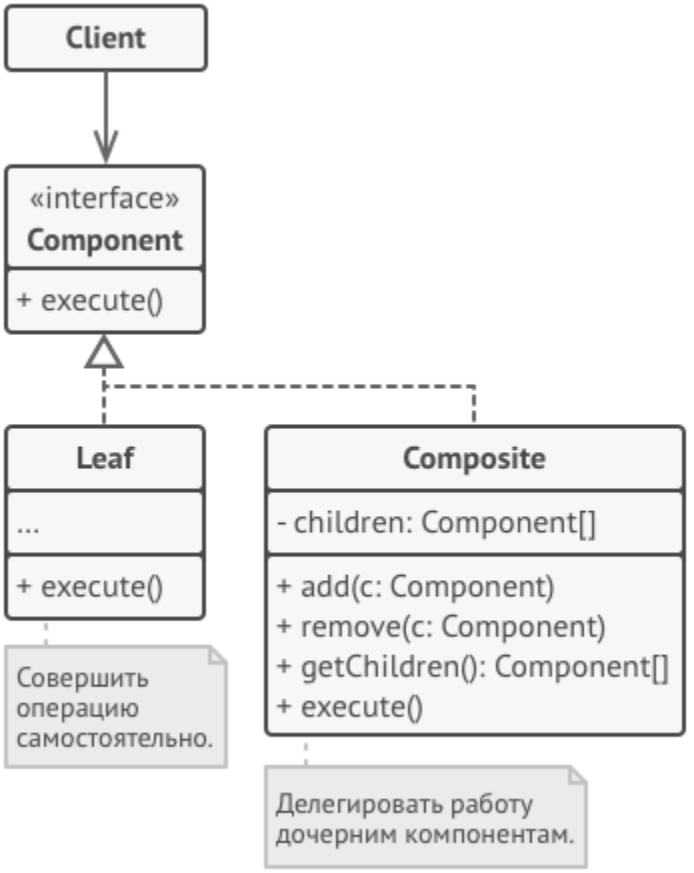
Компоновщик

Компоновщик — это структурный паттерн проектирования, который позволяет сгруппировать множество объектов в древовидную структуру, а затем работать с ней так, как будто это единичный объект.
Теперь предположим, ваши Продукты и Коробки могут быть частью заказов. Каждый заказ может содержать как простые Продукты без упаковки, так и составные Коробки. Ваша задача состоит в том, чтобы узнать цену всего заказа.
Компоновщик предлагает рассматривать Продукт и Коробку через единый интерфейс с общим методом получения стоимости.





Компонент определяет общий интерфейс для простых и составных компонентов дерева.
Лист – это простой компонент дерева, не имеющий ответвлений.
Контейнер (или композит) — это составной компонент дерева. Он содержит набор дочерних компонентов, но ничего не знает об их типах. Это могут быть как простые компоненты-листья, так и другие компоненты-контейнеры. Но это не является проблемой, если все дочерние компоненты следуют единому интерфейсу.



Преимущества

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

Недостатки

- Создаёт слишком общий дизайн классов.

Когда использовать?

- Когда вам нужно представить древовидную структуру объектов.
- Когда клиенты должны единообразно трактовать простые и составные объекты.

Декоратор

Декоратор — это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».
Вы работаете над библиотекой оповещений, которую можно подключать к разнообразным программам, чтобы получать уведомления о важных событиях.
В какой-то момент стало понятно, что одних email-оповещений пользователям мало. Некоторые из них хотели бы получать извещения о критических проблемах через SMS. Другие хотели бы получать их в виде сообщений Facebook. Корпоративные пользователи хотели бы видеть сообщения в Slack.

Декоратор имеет альтернативное название — обёртка. Оно более точно описывает суть паттерна: вы помещаете целевой объект в другой объект-обёртку, который запускает базовое поведение объекта, а затем добавляет к результату что-то своё.

Компонент задаёт общий интерфейс обёрток и оборачиваемых объектов.
Конкретный компонент определяет класс оборачиваемых объектов. Он содержит какое-то базовое поведение, которое потом изменяют декораторы.
Базовый декоратор хранит ссылку на вложенный объект-компонент.
Конкретные декораторы — это различные вариации декораторов, которые содержат добавочное поведение.
Клиент может оборачивать простые компоненты и декораторы в другие декораторы, работая со всеми объектами через общий интерфейс компонентов.

Преимущества

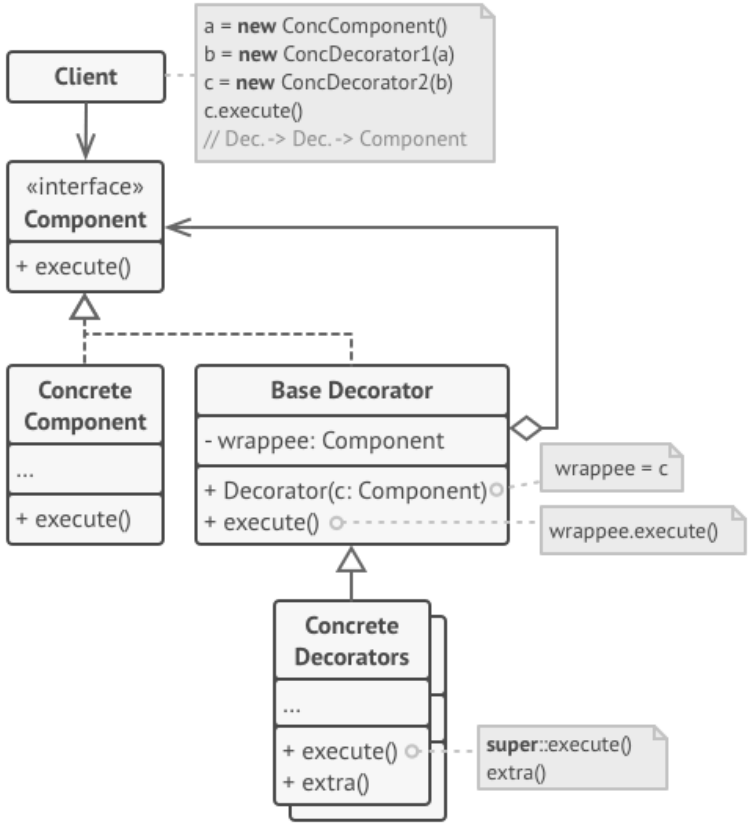
- Большая гибкость, чем у наследования.
- Позволяет добавлять обязанности на лету.
- Можно добавлять несколько новых обязанностей сразу.
- Позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.

Недостатки

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

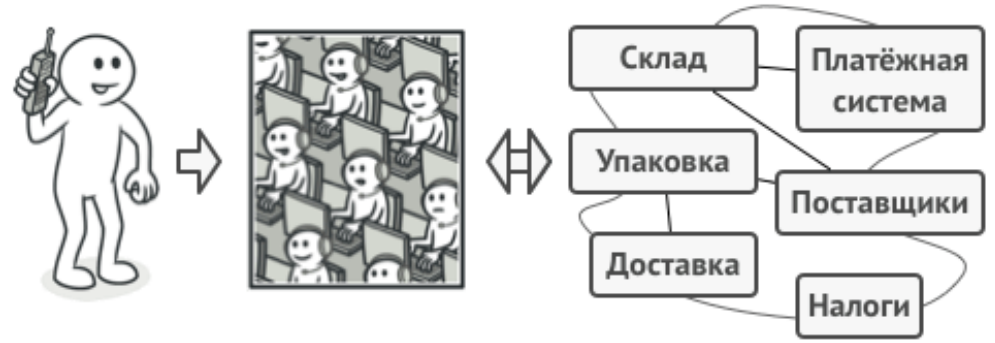
Когда использовать?

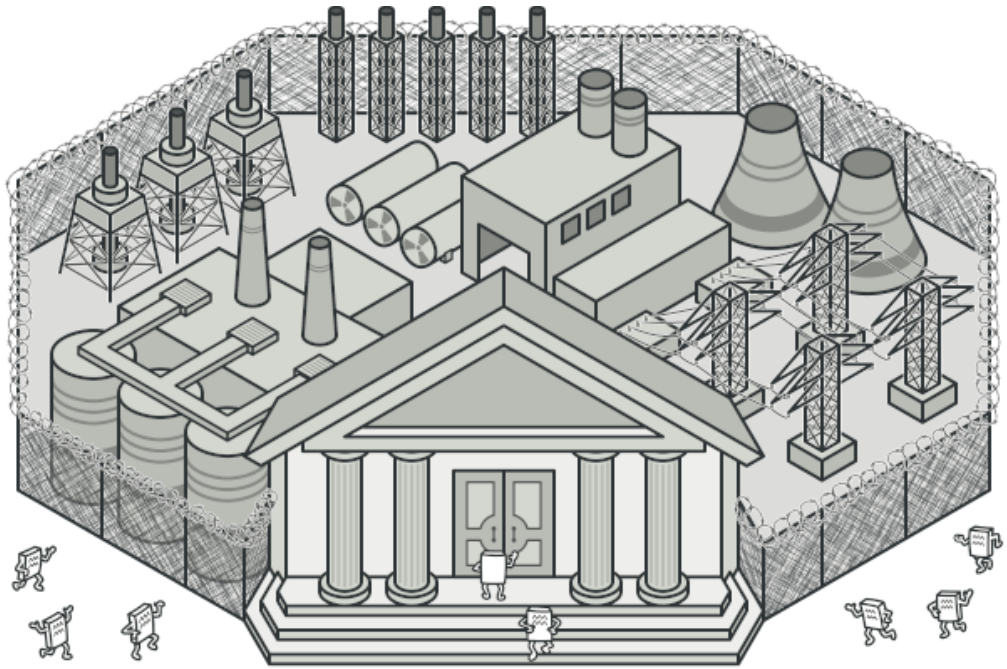
- Когда вам нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует.
- Когда нельзя расширить обязанности объекта с помощью наследования.



Фасад

Фасад — это структурный паттерн проектирования, который предоставляет простой интерфейс к сложной системе классов, библиотеке или фреймворку. Вашему коду приходится работать с большим количеством объектов некой сложной библиотеки или фреймворка. Вы должны самостоятельно инициализировать эти объекты, следить за правильным порядком зависимостей и так далее. В результате бизнес-логика ваших классов тесно переплетается с деталями реализации сторонних классов. Такой код довольно сложно понимать и поддерживать.





Фасад предоставляет быстрый доступ к определённой функциональности подсистемы. Он «знает», каким классам нужно переадресовать запрос, и какие данные для этого нужны.

Дополнительный фасад можно ввести, чтобы не «захламлять» единственный фасад разнородной функциональностью. Он может использоваться как клиентом, так и другими фасадами.

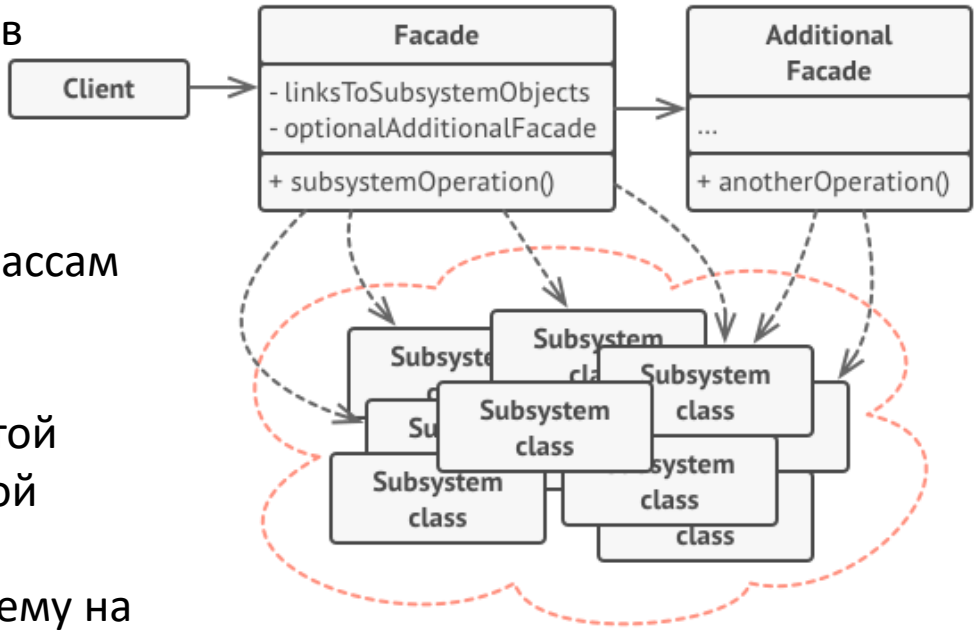
Сложная подсистема состоит из множества разнообразных классов. Для того, чтобы заставить их что-то делать, нужно знать подробности устройства подсистемы, порядок инициализации объектов и так далее.

Преимущества

- Изолирует клиентов от компонентов сложной подсистемы.

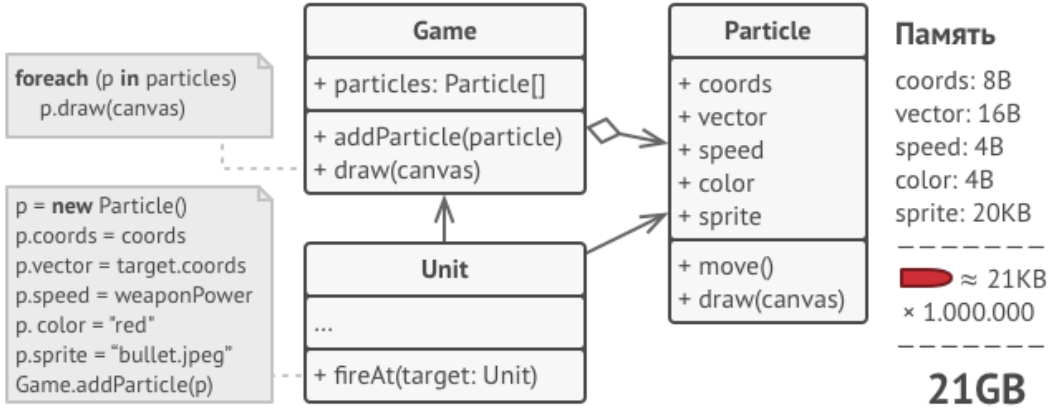
Недостатки

- Фасад рискует стать божественным объектом, привязанным ко всем классам программы.
- Когда использовать?
- Когда вам нужно представить простой или урезанный интерфейс к сложной подсистеме.
- Когда вы хотите разложить подсистему на отдельные слои.

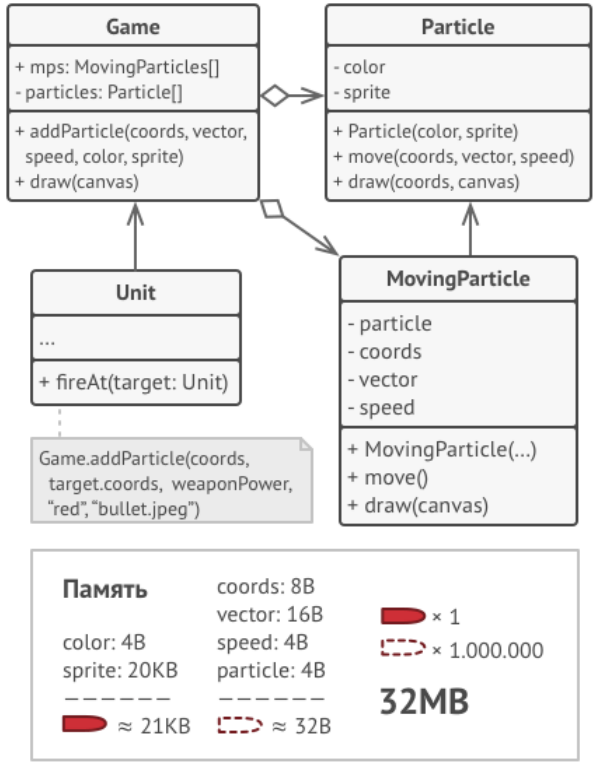


Легковес

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.



На досуге вы решили написать небольшую игру, в которой игроки перемещаются по карте и стреляют друг в друга. Фишкой игры должна была стать реалистичная система частиц. Пули, снаряды, осколки от взрывов — всё это должно красиво летать и радовать взгляд.



Неизменяемость Легковесов

Так как объекты легковесов будут использованы в разных контекстах, вы должны быть уверены в том, что их состояние невозможно изменить после создания. Всё внутреннее состояние легковес должен получать через параметры конструктора.

Легковес содержит состояние, которое повторялось во множестве первоначальных объектов. Один и тот же легковес можно использовать в связке со множеством контекстов.

Контекст содержит «внешнюю» часть состояния, уникальную для каждого объекта. Поведение оригинального объекта чаще всего оставляют в Легковесе, передавая значения контекста через параметры методов.

Когда использовать?

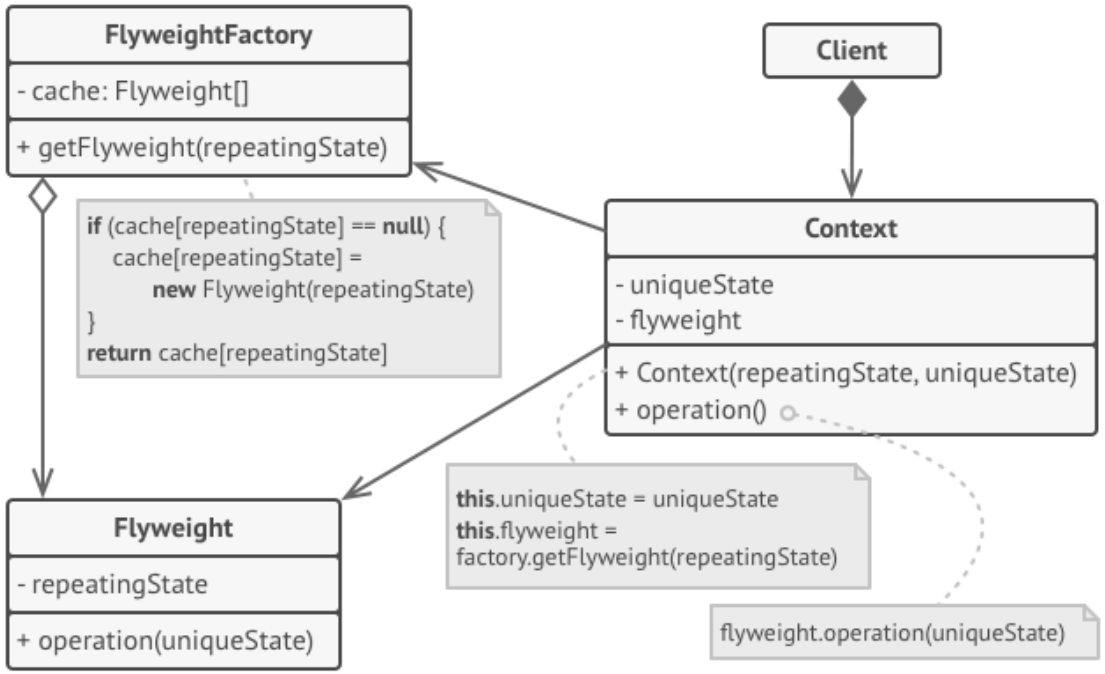
- Когда не хватает оперативной памяти для поддержки всех нужных объектов.

Преимущества

- Экономит оперативную память.

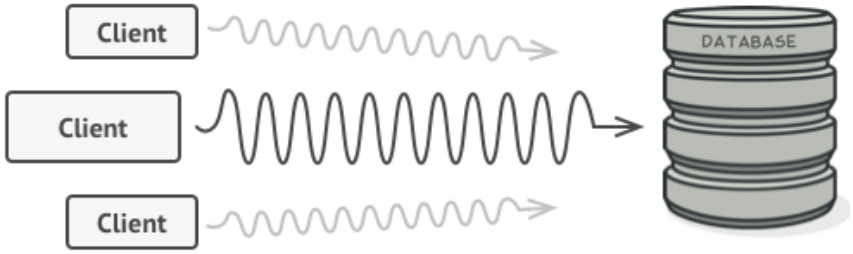
Недостатки

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

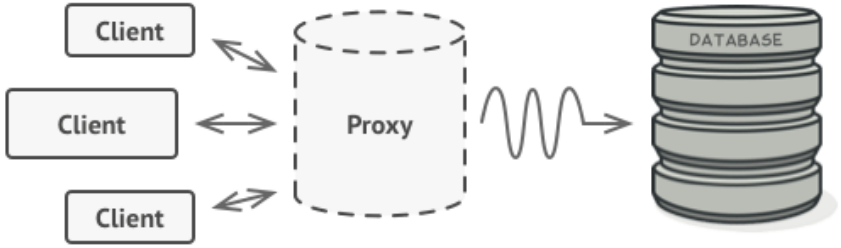


Заместитель

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.
Мы могли бы создавать этот объект не в самом начале программы, а только тогда, когда он кому-то реально понадобится. Каждый клиент объекта получил бы некий код отложенной инициализации. Но, вероятно, это привело бы к множественному дублированию кода.
В идеале, этот код хотелось бы поместить прямо в служебный класс, но это не всегда возможно. Например, код класса может находиться в закрытой сторонней библиотеке.



Запросы к базе данных могут быть очень медленными.



Заместитель «притворяется» базой данных, ускоряя работу за счёт ленивой инициализации и кеширования повторяющихся запросов.

Интерфейс сервиса определяет общий интерфейс для сервиса и заместителя.
Сервис содержит полезную бизнес-логику.
Заместитель хранит ссылку на объект сервиса. После того как заместитель заканчивает свою работу (например, инициализацию, логирование, защиту или другое), он передаёт вызовы вложенному сервису.

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.

Когда использовать?

- Ленивая инициализация.
- Защита доступа.
- Локальный запуск сервиса.
- Логирование запросов.
- Кеширование объектов.