



ОНЛАЙН-ОБРАЗОВАНИЕ

Онлайн-образование



Меня хорошо видно && слышно?

Ставьте ☐, если все хорошо
Напишите в чат, если есть проблемы

Не забыть включить запись!





Структурные шаблоны проектирования

Приходько Роман

Правила вебинара



Активно участвуем



Задаем вопрос в чат или голосом



Off-topic обсуждаем в Slack #канал группы или #general



На вопросы отвечаю в конце секций

Цели вебинара | После занятия вы сможете

1

Понять что такое шаблоны проектирования и для чего они нужны

2

Разобраться в структурных шаблонах проектирования

Смысл | Зачем вам это уметь



1

Полезные «велосипеды» уже изобретены до вас

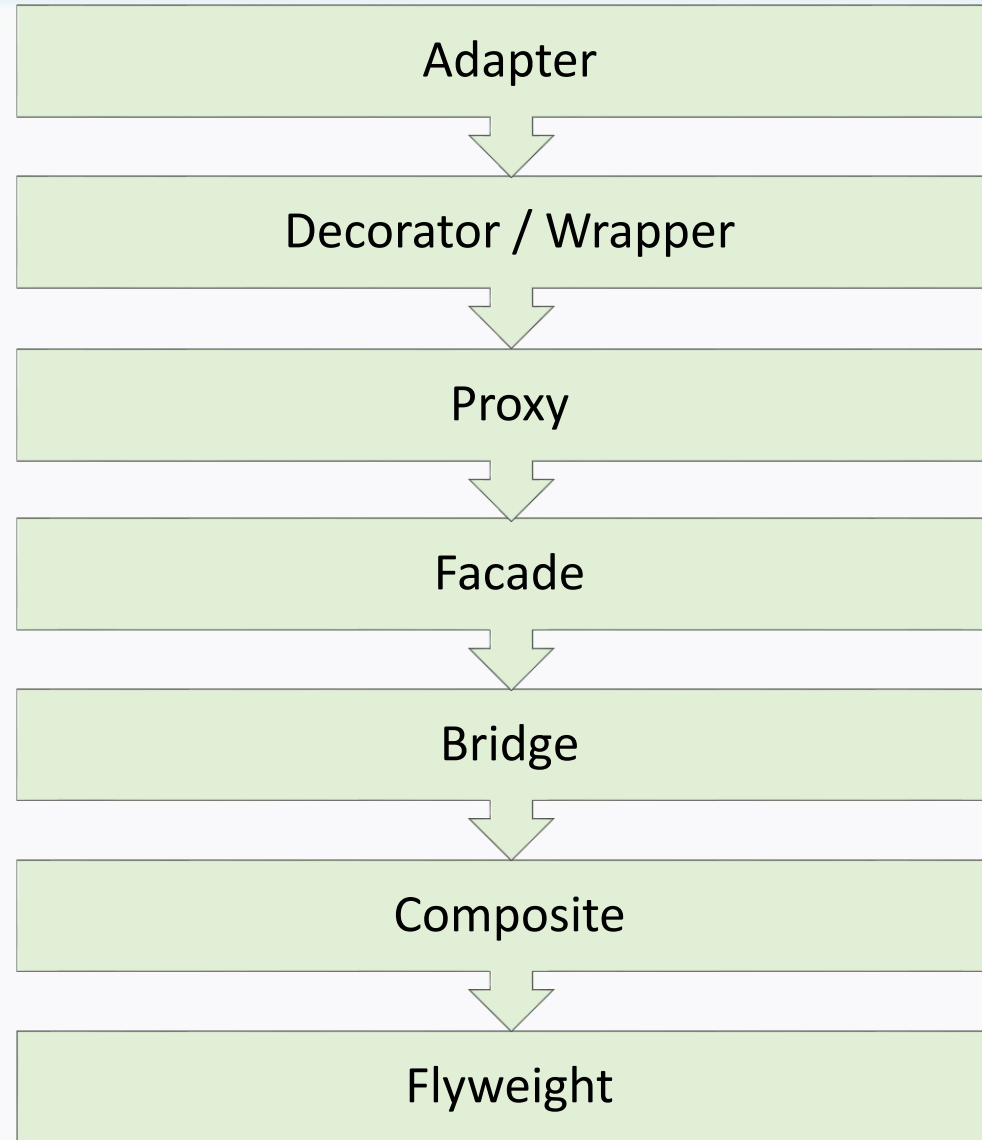
2

Использование общей терминологии помогает быстрее договариваться при командной работе

3

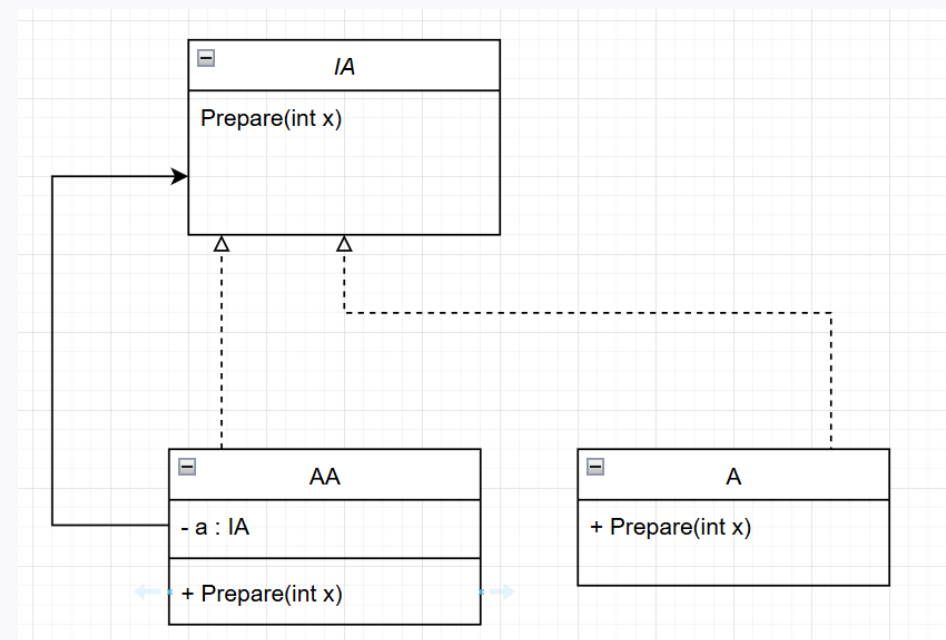
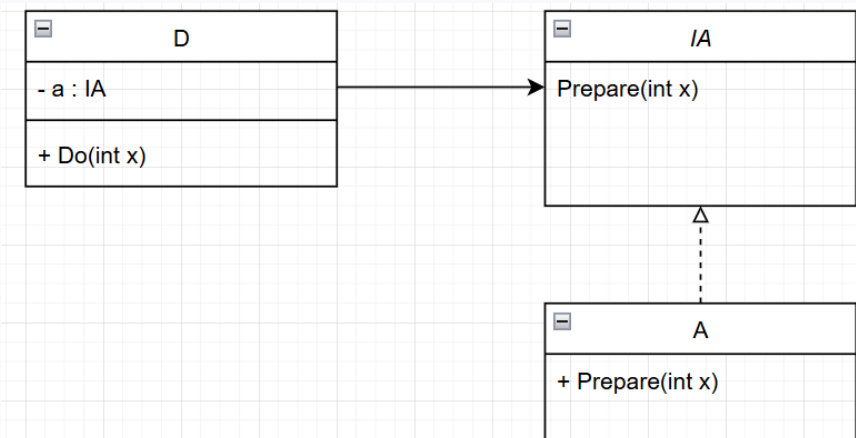
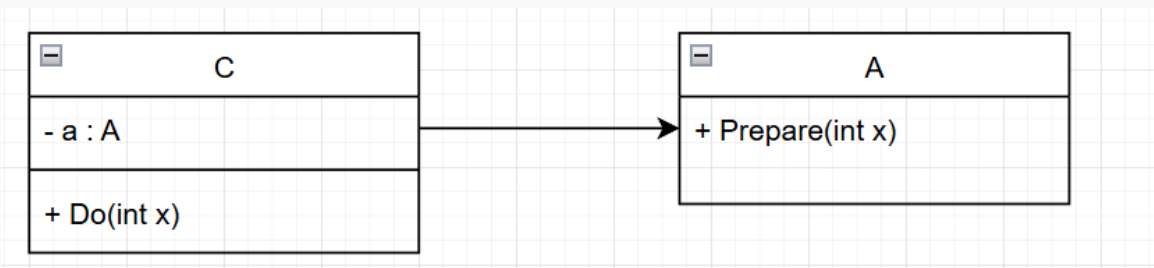
Почти на любом собеседовании вас спросят о шаблонах проектирования

Маршрут вебинара



Предварительное упражнение 1

В методе класса A вызвать метод класса B



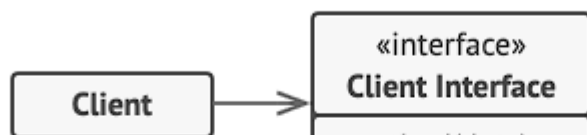
Ассоциация, типовые схемы

Проблема:

Нужно использовать сторонний класс, но его типы не соответствуют типам
вашего приложения

Adapter

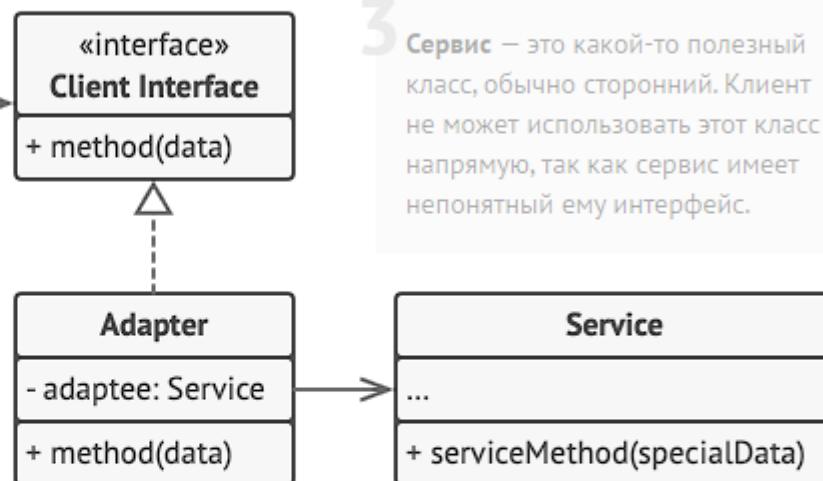
1 Клиент — это класс, который содержит существующую бизнес-логику программы.



2 Клиентский интерфейс описывает протокол, через который клиент может работать с другими классами.

3 Сервис — это какой-то полезный класс, обычно сторонний. Клиент не может использовать этот класс напрямую, так как сервис имеет непонятный ему интерфейс.

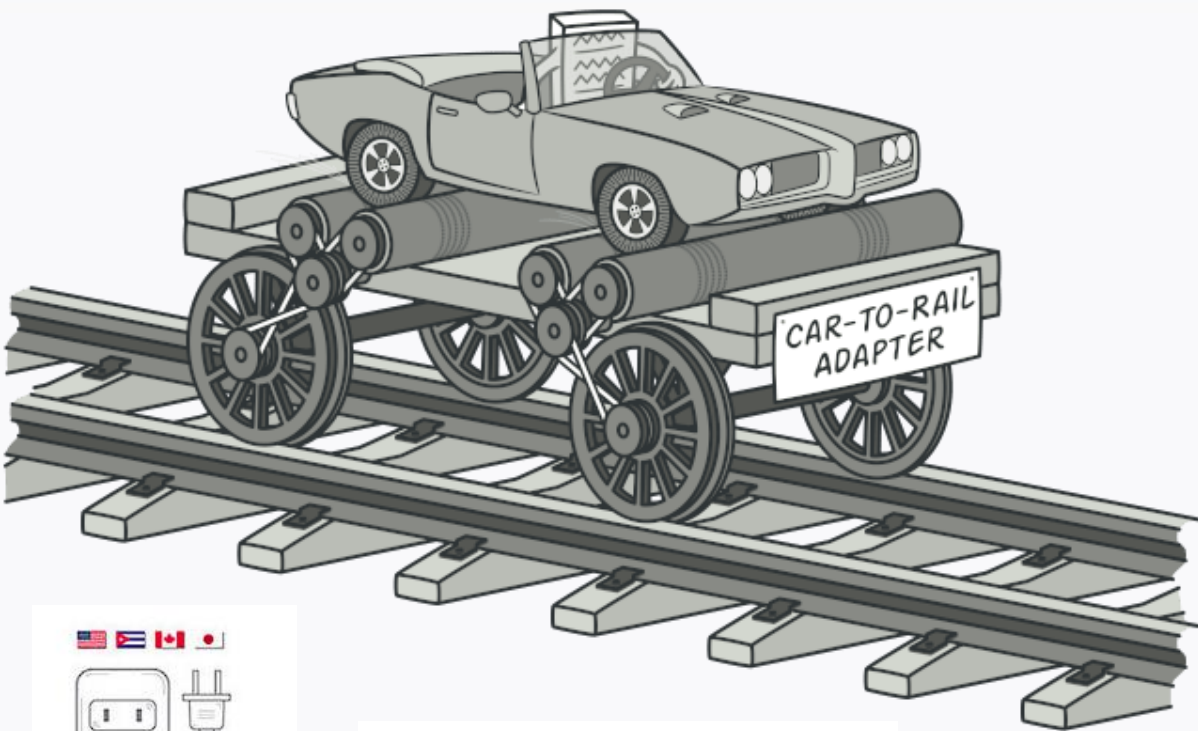
5 Работая с адаптером через интерфейс, клиент не привязывается к конкретному классу адаптера. Благодаря этому, вы можете добавлять в программу новые виды адаптеров, независимо от клиентского кода. Это может пригодиться, если интерфейс сервиса вдруг изменится, например, после выхода новой версии сторонней библиотеки.



```
specialData = convertToServiceFormat(data)
return adaptee.serviceMethod(specialData)
```

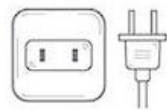
4 Адаптер — это класс, который может одновременно работать и с клиентом, и с сервисом. Он реализует клиентский интерфейс и содержит ссылку на объект сервиса. Адаптер получает вызовы от клиента через методы клиентского интерфейса, а затем переводит их в вызовы методов обёрнутого объекта в правильном формате.

Adapter



Когда вы хотите использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения.

Адаптер позволяет создать объект-прокладку, который будет превращать вызовы приложения в формат, понятный стороннему классу.



type A



type C



Adapter

- + Скрывает от клиента подробности преобразования разных интерфейсов
- Дополнительные классы

Проблема:

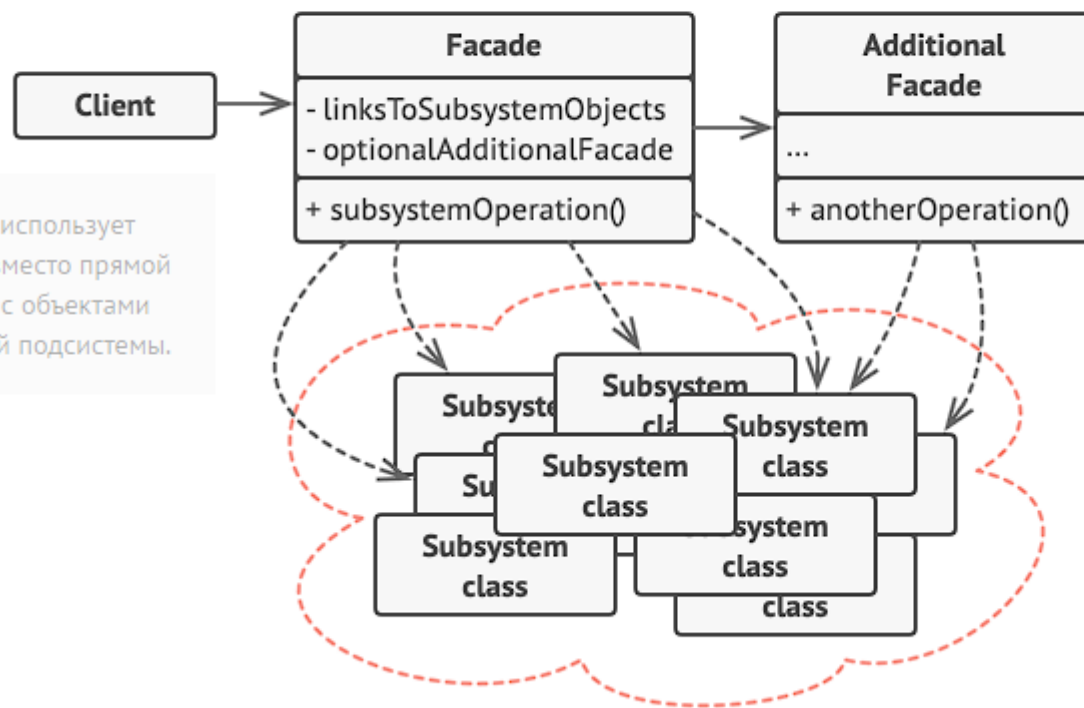
Существует совокупность классов с большим функционалом. Нашему классу требуется упрощенная часть этого функционала

Facade

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

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

4 **Клиент** использует фасад вместо прямой работы с объектами сложной подсистемы.



3 **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.

Классы подсистемы не знают о существовании фасада и работают друг с другом напрямую.

Facade



Когда вам нужно представить простой или урезанный интерфейс к сложной подсистеме.

Facade

✓ Изолирует клиентов от компонентов сложной подсистемы.

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

Проблема:

Объект совершает тяжелую операцию

Но в некоторых случаях она может выполняться простым способом

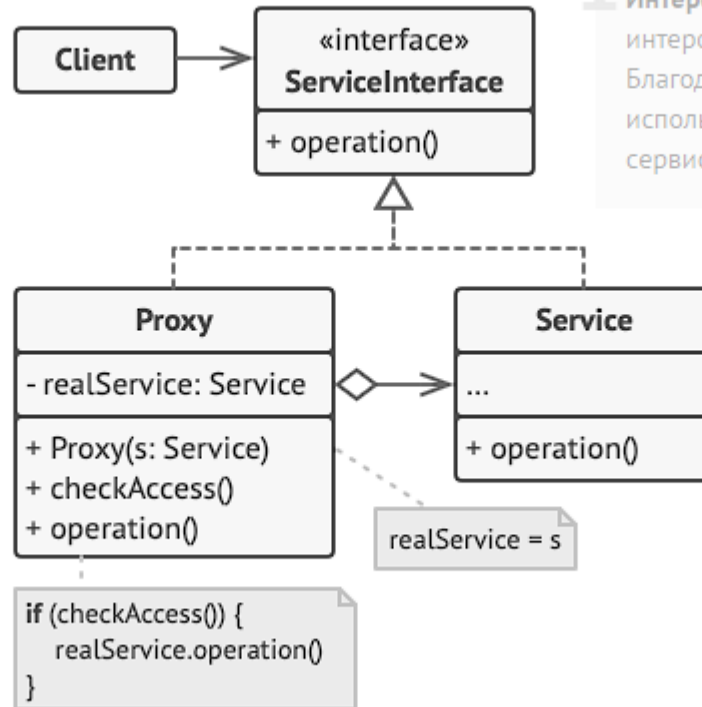
В этих случаях не нужно делать ее тяжелым способом

Proxy (заместитель)

4 Клиент работает с объектами через интерфейс сервиса. Благодаря этому, его можно «одурачить», подменив объект сервиса объектом заместителя.

3 Заместитель хранит ссылку на объект сервиса. После того как заместитель заканчивает свою работу (например, инициализацию, логирование, защиту или другое), он передаёт вызовы вложенному сервису.

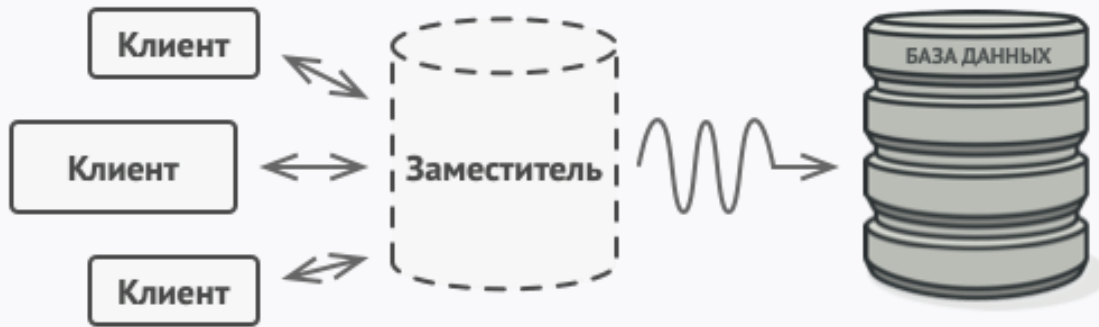
Заместитель может сам отвечать за создание и удаление объекта сервиса.



1 Интерфейс сервиса определяет общий интерфейс для сервиса и заместителя. Благодаря этому, объект заместителя можно использовать там, где ожидается объект сервиса.

2 Сервис содержит полезную бизнес-логику.

Прoxy (заместитель)



- Для каких случаев:
 - Логирование запросов
 - Кэширование ответов
 - Защита доступа
 - ...
- Используется композиция или наследование
- Прокси-класс сам управляет жизнью проксируемого класса

Прoxy (заместитель)

- | | |
|--|--|
| ✓ Позволяет контролировать сервисный объект незаметно для клиента. | ✗ Усложняет код программы из-за введения дополнительных классов. |
| ✓ Может работать, даже если сервисный объект ещё не создан. | ✗ Увеличивает время отклика от сервиса. |
| ✓ Может контролировать жизненный цикл служебного объекта. | |

Проблема:

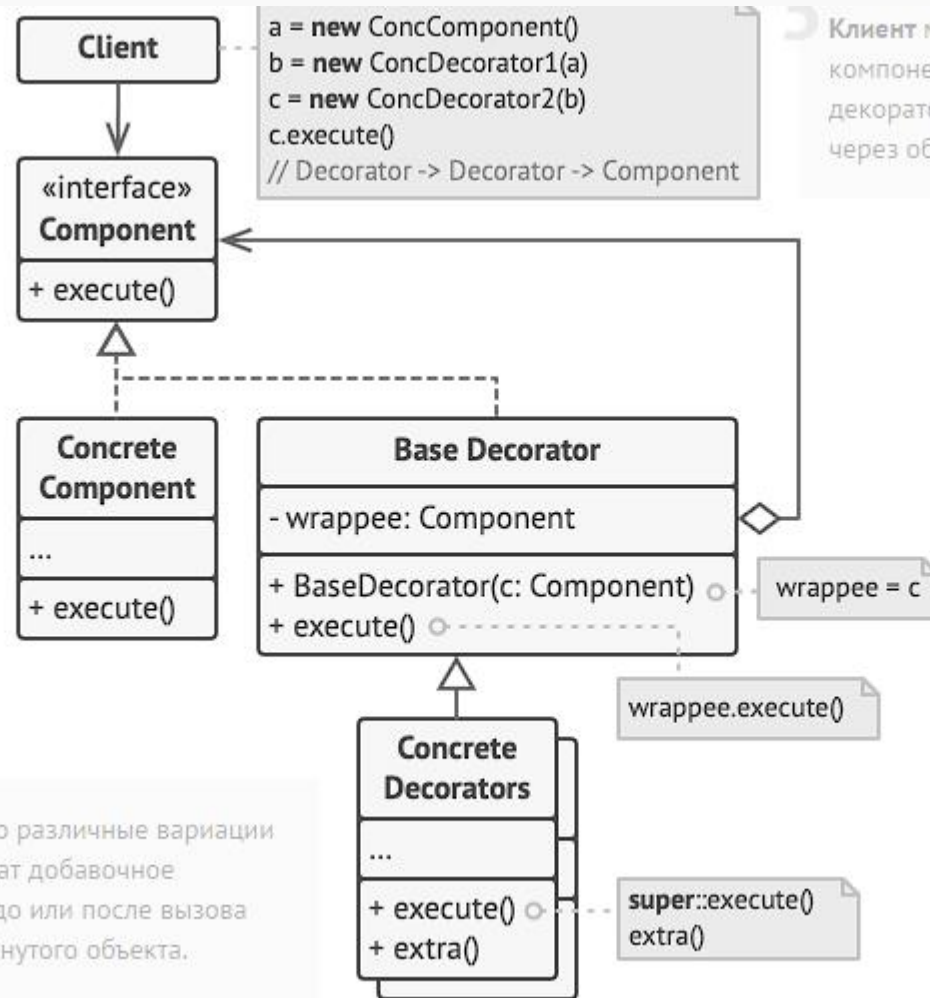
Нужно на лету добавлять объекту типовые обязанности

Decorator / Wrapper (обертка)

1 Компонент задаёт общий интерфейс обёрток и оборачиваемых объектов.

2 Конкретный компонент определяет класс оборачиваемых объектов. Он содержит какое-то базовое поведение, которое потом изменяют декораторы.

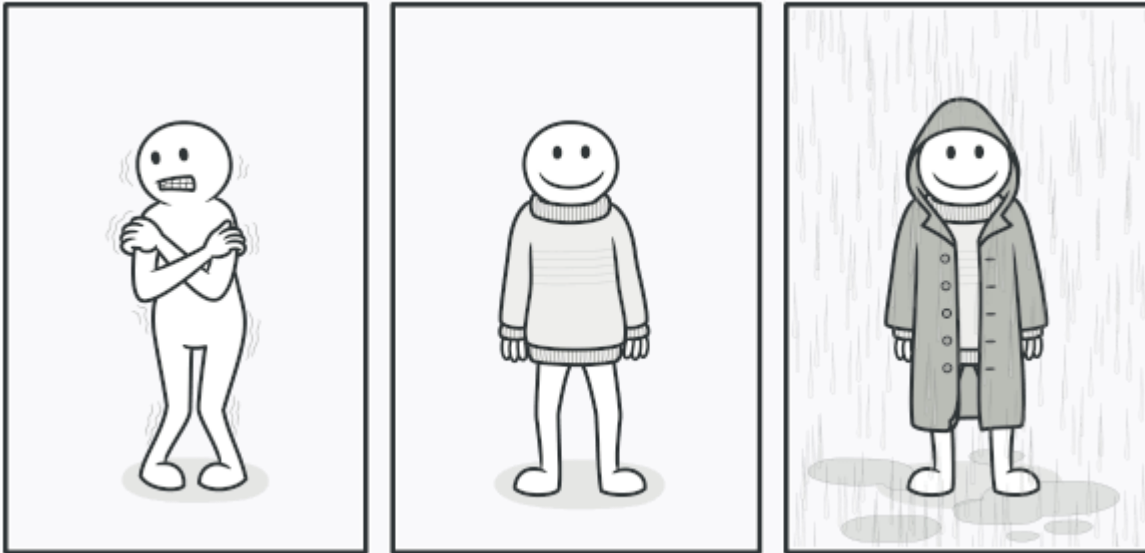
4 Конкретные декораторы — это различные вариации декораторов, которые содержат добавочное поведение. Оно выполняется до или после вызова аналогичного поведения обёрнутого объекта.



5 Клиент может оборачивать простые компоненты и декораторы в другие декораторы, работая со всеми объектами через общий интерфейс компонентов.

3 Базовый декоратор хранит ссылку на вложенный объект-компонент. Им может быть как конкретный компонент, так и один из конкретных декораторов. Базовый декоратор делегирует все свои операции вложенному объекту. Дополнительное поведение будет жить в конкретных декораторах.

Decorator / Wrapper (обертка)



- Если нужно добавить новую бизнес-логику, но наследование использовать не хочется или нельзя
- Если хочется добавлять обязанности объектам «на лету» незаметно для использующего кода



Добавляет новый функционал к розетке
При этом не меняет интерфейс и не
контролирует работу устройства

Decorator / Wrapper (обертка)

Преимущества и недостатки

- ✓ 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.
- ✗ Трудно конфигурировать многократно обернутые объекты.
- ✗ Обилие крошечных классов.

+ Помогает реализовать Open-Close Principle

Сравнение паттернов

TableConnector : IRepository

READ
WRITE
CLEAR_ALL

IRepository:

READ
WRITE
CLEAR_ALL

PROXY

ProxyTableConnector : IRepository

READ
 +проверяю в кэше, если есть беру
 оттуда
 TableConnector.READ
WRITE
 TableConnector.WRITE
CLEAR_ALL
 TableConnector.CLEAR_ALL

Тот же интерфейс,
Композиция или наследование

DECORATOR / WRAPPER

TableConnectorEx : IRepository

READ
 TableConnector.READ
WRITE
 +уведомляю подписчиков
 TableConnector.WRITE
CLEAR_ALL
 +уведомляю подписчиков
 TableConnector.CLEAR_ALL
+UPDATE
 обновляю строку

Может расширять интерфейс,
Агрегация

ADAPTER

TableAsyncConnector

READ_ASYNC
 TableConnector.READ в отдельном потоке
 возвращаю задачу
WRITE_ASYNC
 TableConnector.WRITE в отдельном потоке
 возвращаю задачу
CLEAR_ALL_ASYNC
 TableConnector.CLEAR_ALL в отдельном потоке
 возвращаю задачу

Альтернативный интерфейс
Агрегация

FACADE

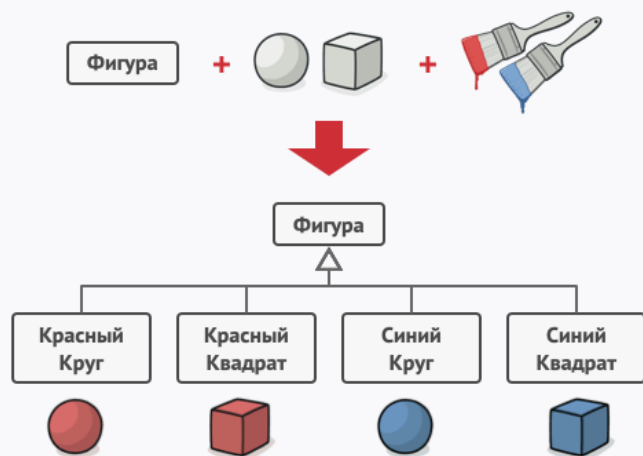
TableModifier

READ
 TableConnector.READ
CHANGE
 TableConnector.CLEAR_ALL
 TableConnector.WRITE

Упрощенный интерфейс
Агрегация или композиция

Проблема:

Есть две разных иерархии, которые могут соединяться в разных комбинациях
Нужно реализовать их комбинации



Bridge (мост)

1 Абстракция содержит управляющую логику. Код абстракции делегирует реальную работу связанному объекту реализации.

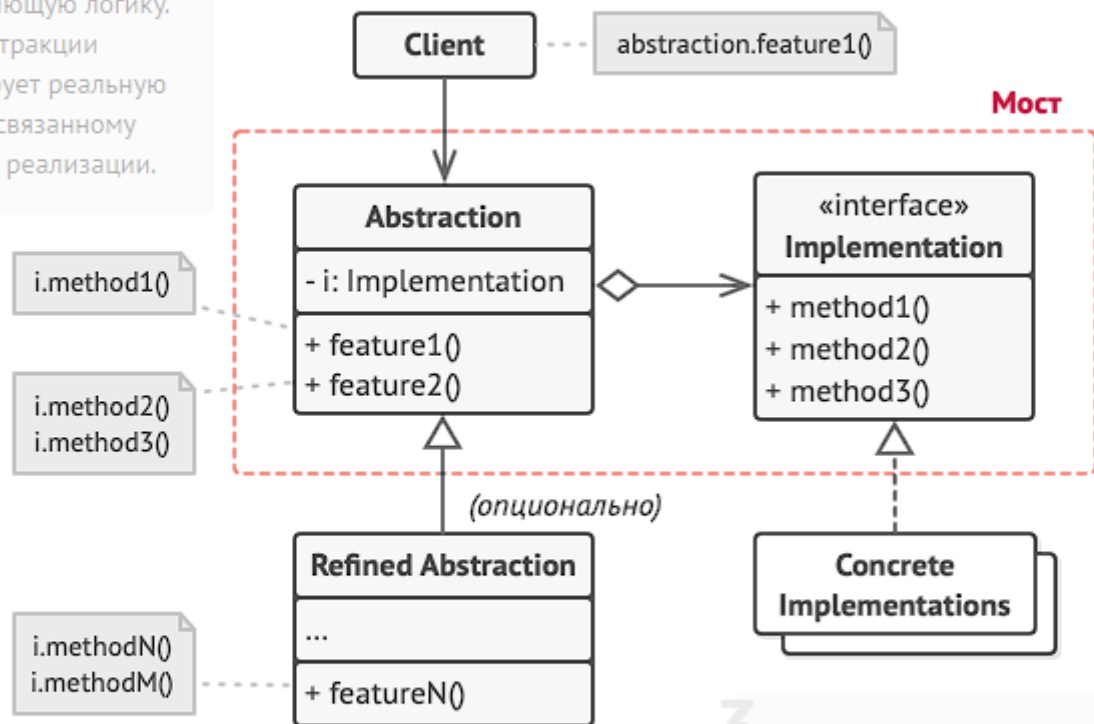
Клиент работает только с объектами абстракции. Не считая начального связывания абстракции с одной из реализаций, клиентский код не имеет прямого доступа к объектам реализации.

2 Реализация задаёт общий интерфейс для всех реализаций. Все методы, которые здесь описаны, будут доступны из класса абстракции и его подклассов.

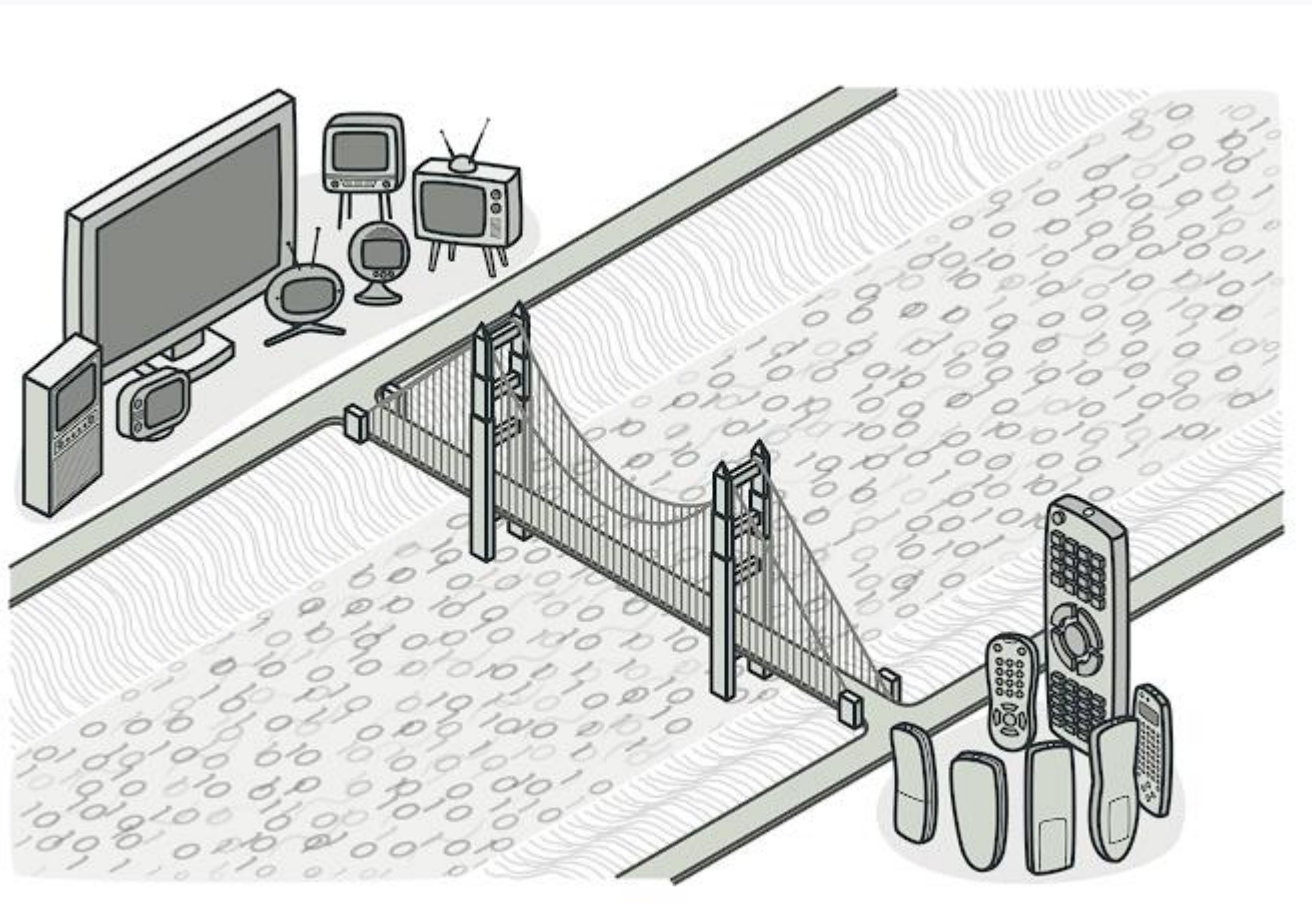
Интерфейсы абстракции и реализации могут как совпадать, так и быть совершенно разными. Но обычно в реализации живут базовые операции, на которых строятся сложные операции абстракции.

3 Конкретные реализации содержат платформо-зависимый код.

4 Расширенные абстракции содержат различные вариации управляющей логики. Как и родитель, работает с реализациями только через общий интерфейс реализации.



Bridge (мост)



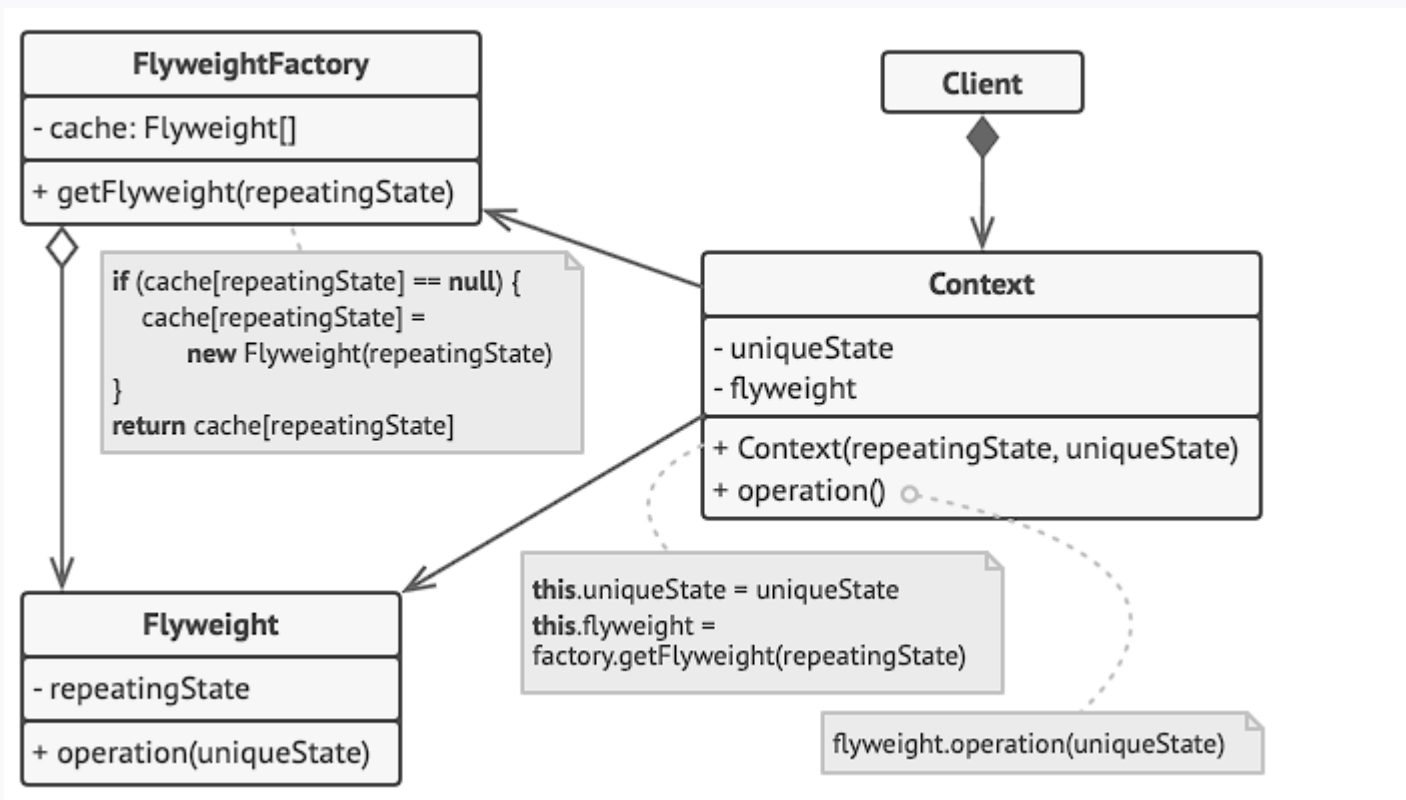
Bridge (мост)

- + Помогает реализовать Open-Close Principle
- Дополнительные классы

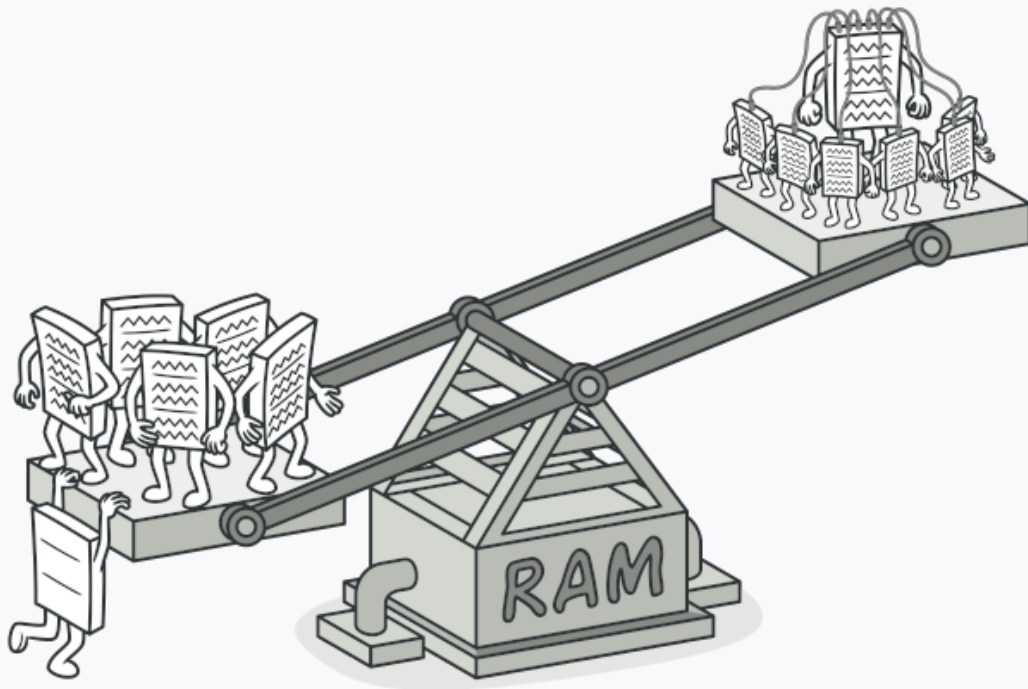
Проблема:

Существует класс, у которого предполагается большое количество экземпляров. Нужно оптимизировать данные в этом классе

Flyweight (легковес)



Flyweight (легковес)



- Если нужно сэкономить оперативную память при большом количестве объектов

Flyweight (легковес)

✓ Экономит оперативную память.

✗ Расходует процессорное время на поиск/вычисление контекста.

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

Проблема:

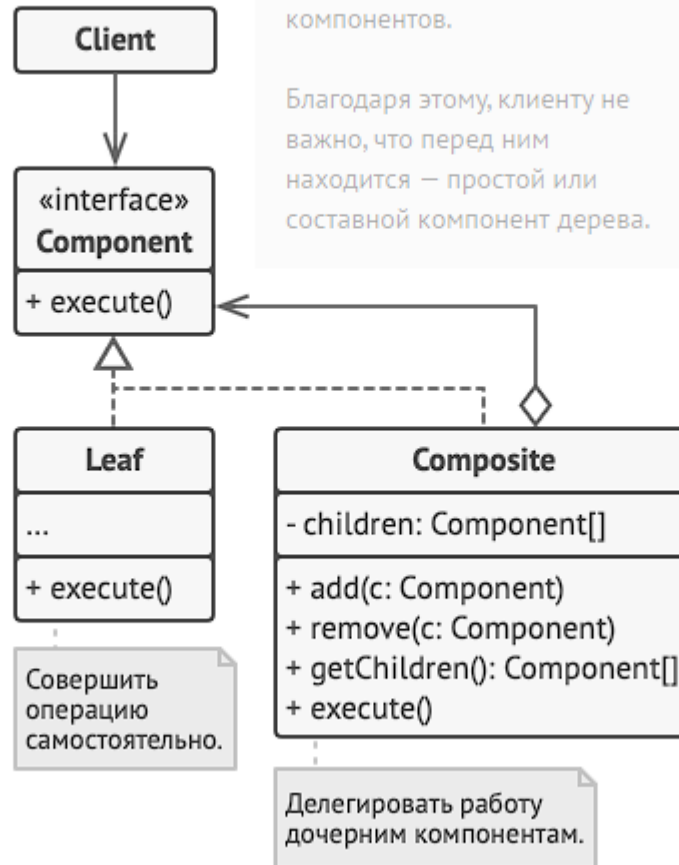
Существуют однотипные объекты, у каждого из которых реализована операция. Нужно уметь выполнить эту операцию над любой иерархией этих объектов

Composite (компоновщик)

1 **Компонент** определяет общий интерфейс для простых и составных компонентов дерева.

2 **Лист** — это простой компонент дерева, не имеющий ответвлений.

Из-за того, что им некому больше передавать выполнение, классы листьев будут содержать большую часть полезного кода.



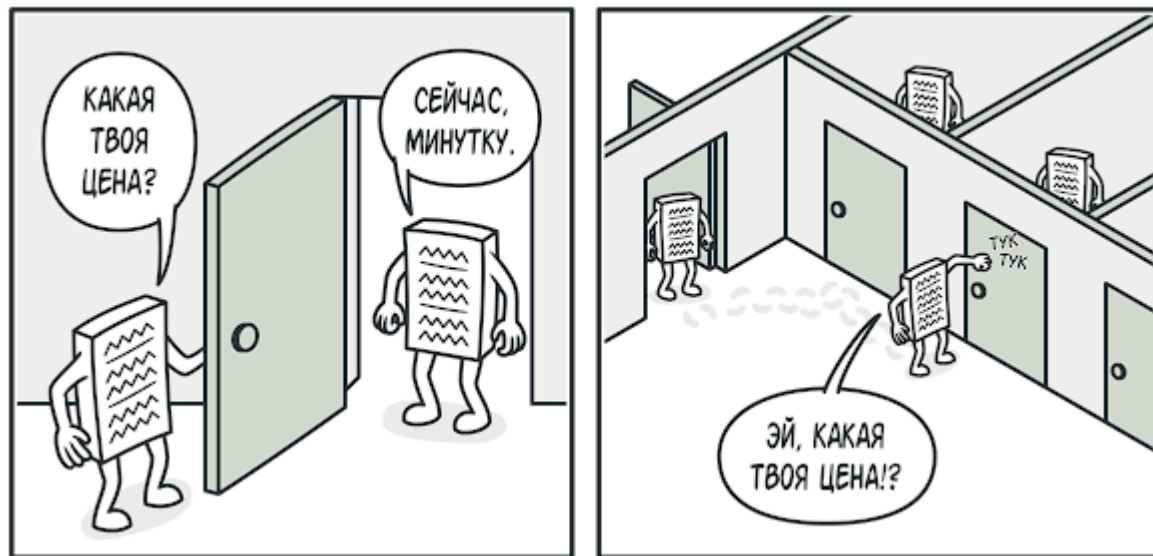
Клиент работает с деревом через общий интерфейс компонентов.

Благодаря этому, клиенту не важно, что перед ним находится — простой или составной компонент дерева.

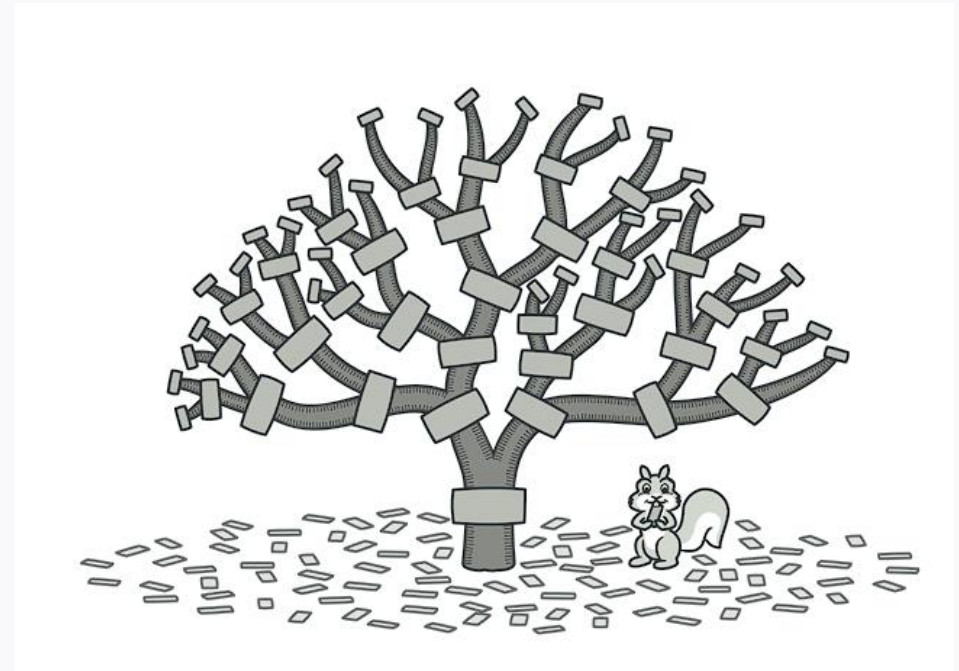
3 **Контейнер** (или *композит*) — это составной компонент дерева. Он содержит набор дочерних компонентов, но ничего не знает об их типах. Это могут быть как простые компоненты-листья, так и другие компоненты-контейнеры. Но это не является проблемой, если все дочерние компоненты следуют единому интерфейсу.

Методы контейнера переадресуют основную работу своим дочерним компонентам, хотя и могут добавлять что-то своё к результату.

Composite (компоновщик)



- Если нужна работа с группой объектов как с единым объектом (через древовидную структуру)



Composite (компоновщик)

- + Упрощает архитектуру клиента
- + Облегчает добавление новых видов компонентов
- Все компоненты должны реализовывать общий интерфейс

Итоги занятия

1

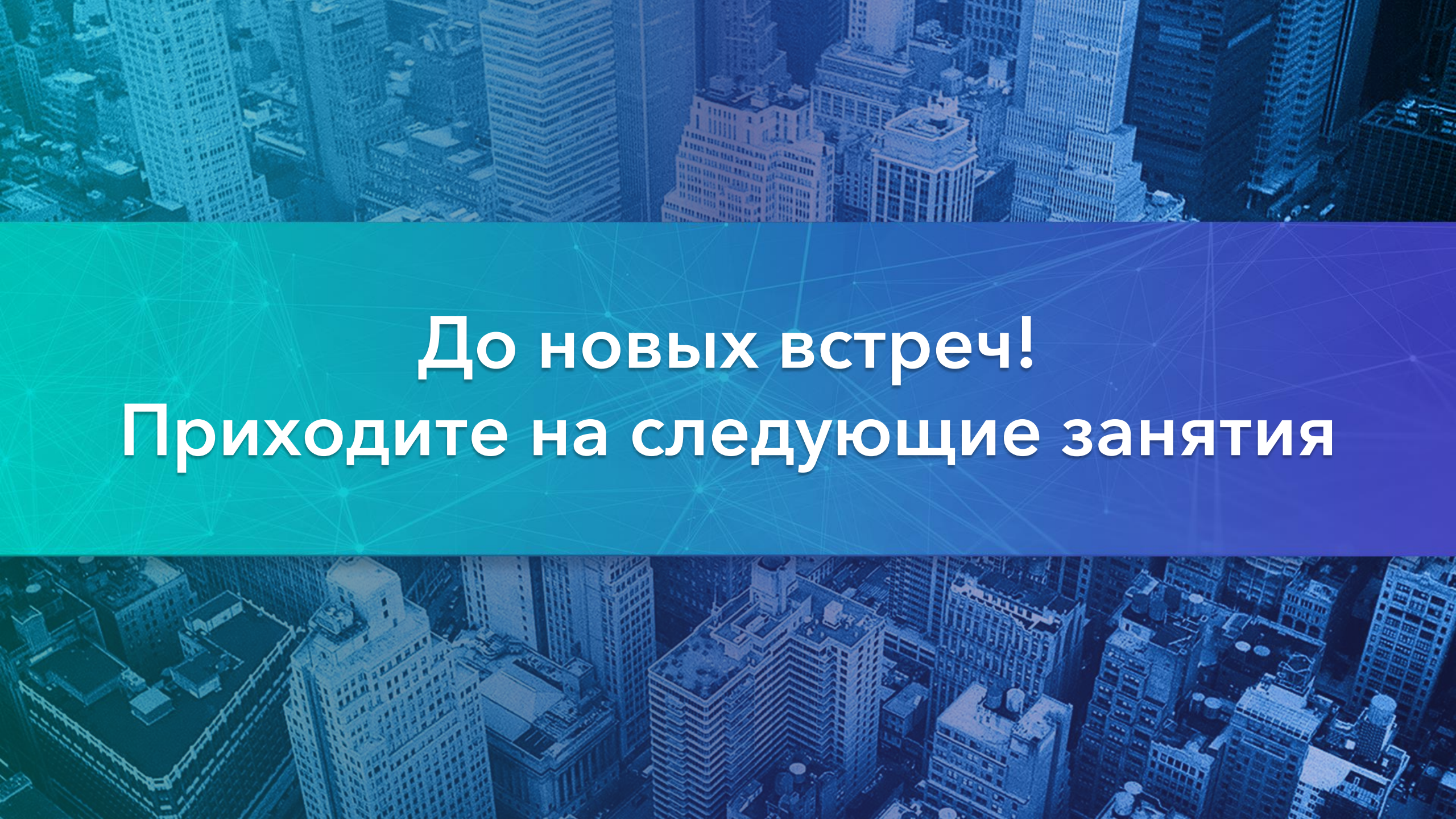
Научились выбирать подходящие
структурные шаблоны проектирования

2

Реализовали популярные
структурные шаблоны на C#



Заполните, пожалуйста,
опрос о занятии по ссылке в чате



До новых встреч!
Приходите на следующие занятия