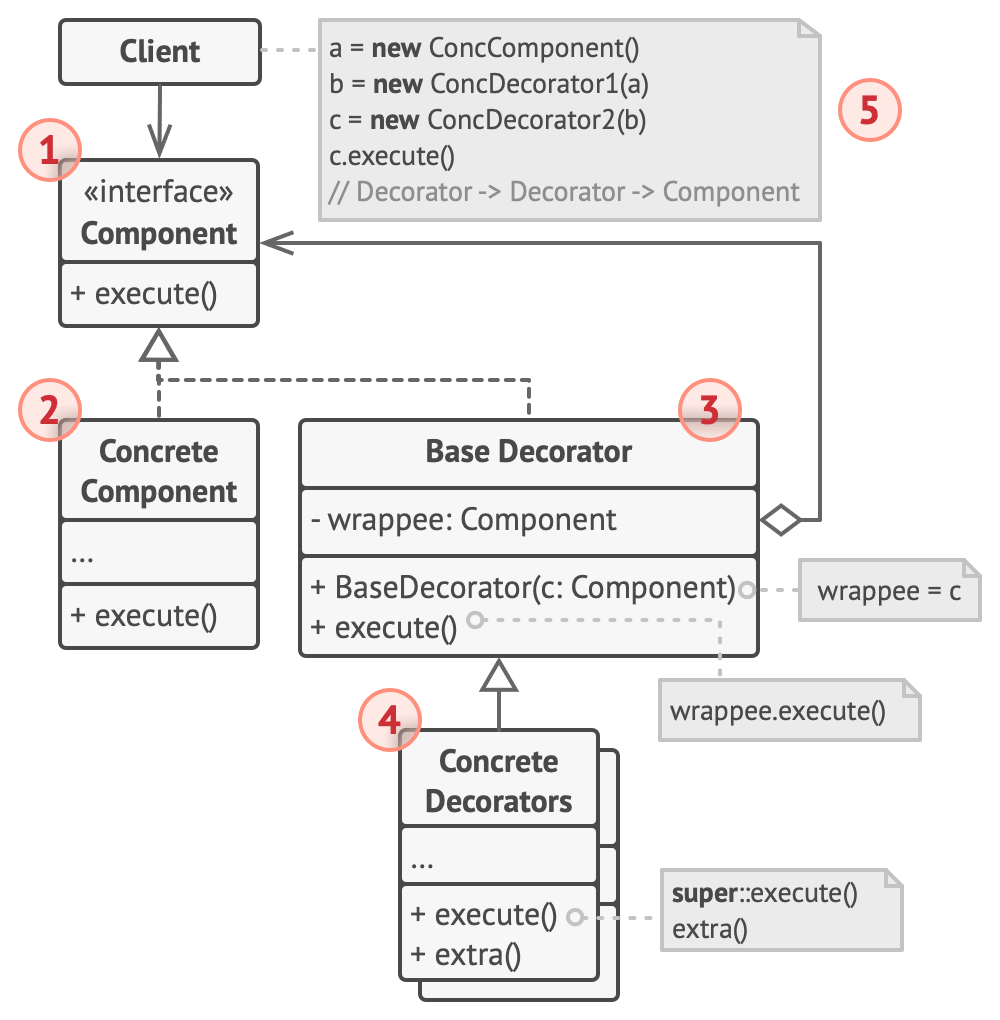
Коллоквиум от 24.10.2023 по предмету ПП

Часть 1

1.3

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

Любая одежда — это аналог Декоратора. Применяя Декоратор, вы не меняете первоначальный класс и не создаёте дочерних классов. Так и с одеждой — надевая свитер, вы не перестаёте быть собой, но получаете новое свойство — защиту от холода. Вы можете пойти дальше и надеть сверху ещё один декоратор — плащ, чтобы защититься и от дождя.



Выше представлена UML-диаграмма, которая демонстрирует преимущества использования такого паттерна, как декоратор.

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.
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.
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.

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.

Оба объекта имеют общий интерфейс, поэтому для пользователя нет никакой разницы, с каким объектом работать — чистым или обёрнутым. Вы можете использовать несколько разных обёрток одновременно — результат будет иметь объединённое поведение всех обёрток сразу.

## Применимость

* **Когда вам нужно добавлять обязанности объектам на лету, незаметно для кода, который их использует.**

Объекты помещают в обёртки, имеющие дополнительные поведения. Обёртки и сами объекты имеют одинаковый интерфейс, поэтому клиентам без разницы, с чем работать — с обычным объектом данных или с обёрнутым.

* **Когда нельзя расширить обязанности объекта с помощью наследования.**

Во многих языках программирования есть ключевое слово final, которое может заблокировать наследование класса. Расширить такие классы можно только с помощью Декоратора.

Часть 2

2.2

Объектно-ориентированная декомпозиция предписывает разделение модели предметной области на элементы, выбирая в качестве критерия декомпозиции принадлежность этих элементов к различным абстракциям данной предметной области. Прежде чем разделять задачу на шаги, необходимо определить объекты предметной области.

Наиболее очевидным примером обсуждаемого метода, удовлетворяющим критерию декомпозиции, является метод нисходящего (сверху вниз) проектирования (top-down design). В соответствии с этим методом разработчик должен начать с наиболее абстрактного описания функции, выполняемой системой. Затем последовательными шагами детализировать это представление, разбивая на каждом шаге каждую подсистему на небольшое число более простых подсистем до тех пор, пока не будут получены элементы с настолько низким уровнем абстракции, что становится возможной их непосредственная реализация.

2.5

Агрегация это когда один объект *содержит* ссылку на другой и делегирует ему работу, вместо того чтобы самому *наследовать* его поведение.

Механизм наследования имеет несколько досадных проблем.

* Он статичен. Вы не можете изменить поведение существующего объекта. Для этого вам надо создать новый объект, выбрав другой подкласс.
* Он не разрешает наследовать поведение нескольких классов одновременно. Из-за этого вам приходится создавать множество подклассов-комбинаций для получения совмещённого поведения.

Одним из способов обойти эти проблемы является замена наследования *агрегацией*. Как раз на этом принципе построен паттерн Декоратор.

Следует использовать наследование, если:

* производный класс имеет тот же тип, что и базовый класс;
* производный класс расширяет функциональность базового класса;
* производный класс представляет уникальный случай базового класса и может использовать и переопределять его методы.

Следует использовать агрегацию, если:

* объект нуждается в более сложной структуре данных, которая состоит из нескольких других объектов;
* это позволяет упростить код и сделать его более модульным;
* объекты могут быть использованы другими объектами и должны быть независимыми.

2.7

При статическом варианте развития событий метод можно вызывать при компиляции кода, происходит это при помощи перегрузки методов. Говоря проще, у них будет одинаковое название, но разная функциональная начинка (тип возвращаемых данных, свойств и т. д.). В таком случае Java позволяет пользователю свободно определять функциям идентичные названия, если они будут отличны по типу и параметрам.

При динамическом полиморфизме вызвать переопределеный метод можно при выполнении программы. Скопировать метод родительского класса и обозначить его в дочернем можно с помощью ссылочной переменной родительского класса, при этом вызванный метод будет определяться по объекту, на который она ссылается. Такую операцию еще называют Upcasting.

2.10

Ситуация типична для любого новаторства в искусстве или технике: автор знает, что именно нового он внес, в чем были его инсайты в процессе создания и какие проблемы были решены. А гуру прежних поколений и наблюдатели со стороны склонны занижать и обесценивать новаторство, объясняя, что всё это уже известно. И сейчас любой IT-шник может это наблюдать в обсуждении новых языков, фреймворков, сред работы или даже новых таск-трекеров и досок.

Часть 3

3.1

Пирамида тестирования, также часто говорят уровни тестирования, это группировка тестов по уровню детализации и их назначению. Эту абстракцию придумал Майк Кон и описал в книге «Scrum: гибкая разработка ПО» (Succeeding With Agile. Software Development Using Scrum).

Пирамиду разбивают на 4 уровня (снизу вверх), например, по ISTQB:

* модульное тестирование (юнит);
* интеграционное тестирование;
* системное тестирования;
* приемочное тестирование.

## Компонентный уровень

Чаще всего называют **юнит тестированием**. Реже называют модульным тестированием. На этом уровне тестируют атомарные части кода. Это могут быть классы, функции или методы классов.

*Пример: твоя компания разрабатывает приложение "Калькулятор", которое умеет складывать и вычитать. Каждая операция это одна функция. Проверка каждой функции, которая не зависит от других, является юнит тестированием.*

На модульном уровне разработчик (или автотестер) использует **метод белого ящика**.Он знает что принимает и отдает минимальная единица кода, и как она работает.

## Интеграционный уровень

Проверят взаимосвязь компоненты, которую проверяли на модульном уровне, с другой или другими компонентами, а также интеграцию компоненты с системой *(проверка работы с ОС, сервисами и службами, базами данных, железом и т.д.)*. Часто в английских статьях называют service test или API test.

В случае с интеграционными тестами редко когда требуется наличие UI, чтобы его проверить. Компоненты ПО или системы взаимодействуют с тестируемым модулем с помощью интерфейсов. Тут начинается участие тестирования. Это проверки API, работы сервисов *(проверка логов на сервере, записи в БД)* и т.п.

На этом уровне используется либо серый, либо черный ящик.

В интеграционном тестировании есть 3 основных способа тестирования

* **Снизу вверх** (Bottom Up Integration): все мелкие части модуля собираются в один модуль и тестируются. Далее собираются следующие мелкие модули в один большой и тестируется с предыдущим и т.д. *Например, функция публикации фото в соц. профиле состоит из 2 модулей: загрузчик и публикатор. Загрузчик, в свою очередь, состоит из модуля компрессии и отправки на сервер. Публикатор состоит из верификатора (проверяет подлинность) и управления доступом к фотографии. В интеграционном тестировании соберем модули загрузчика и проверим, потом соберем модули публикатора, проверим и протестируем взаимодействие загрузчика и публикатор.*
* **Сверху вниз** (Top Down Integration): сначала проверяем работу крупных модулей, спускаясь ниже добавляем модули уровнем ниже. На этапе проверки уровней выше данные, необходимые от уровней ниже, симулируются.*Например, проверяем работу загрузчика и публикатора. Руками (создаем функцию-заглушку) передаем от загрузчика публикатору фото, которое якобы было обработано компрессором.*
* **Большой взрыв** ("Big Bang" Integration): собираем все реализованные модули всех уровней, интегрируем в систему и тестируем. Если что-то не работает или недоработали, то фиксим или дорабатываем.

Системный уровень

На системном уровне выявляются такие дефекты, как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д.

На этом уровне используют черный ящик.

## Приемочное тестирование

Также часто называют E2E тестами (End-2-End) или сквозными. На этом уровне происходит валидация требований *(проверка работы ПО в целом, не только по прописанным требованиям, что проверили на системном уровне).*

Проверка требований производится на наборе приемочных тестов. Они разрабатываются на основе требований и возможных способах использования ПО.

Отмечу, что приемочные тесты проводят, когда (1) продукт достиг необходимо уровня качества и (2) заказчик ПО ознакомлен с планом приемки *(в нем описан набор сценариев и тестов, дата проведения и т.п.).*

Приемку проводит либо внутреннее тестирование *(необязательно тестировщики)* или внешнее тестирование *(сам заказчик и необязательно тестировщик).*

Важно помнить, что E2E тесты автоматизируются сложнее, дольше, стоят дороже, сложнее поддерживаются и трудно выполняются при регрессе. Значит таких тестов должно быть меньше.

3.6

**Test Runner** – это приложение, используемое для запуска тестов, проверки выходных данных, а также предоставления инструментов, необходимых для **диагностики и отладки тестов**.

3.8

Mockito — фреймворк для тестирования приложений, который позволяет легко и быстро подменять реальные объекты программы «пустышками». Такие фиктивные объекты часто называют «моками» (Mock — подражать).

Mock можно создать и для тех классов, новый экземпляр которых вообще-то так просто не создашь, в частности, классов с исключительно приватными конструкторами типа синглтонов и утилитных классов, а при минимальной настройке фреймворка — и перечислений (enums).

3.11

Аннотации представляют из себя дескрипторы, включаемые в текст программы, и используются для хранения метаданных программного кода, необходимых на разных этапах жизненного цикла программы.  
Информация, хранимая в аннотациях, может использоваться соответствующими обработчиками для создания необходимых вспомогательных файлов или для маркировки классов, полей и т.д.

При написании API, Библиотек или программ, в которых возможно повторение однотипного (но не совсем одинакового) кода, аннотации - бесспорно оптимальное решение.

Часть 4, Часть 5, Часть 6 - [Ссылка на github-репозиторий](https://github.com/Mariaastashkevich/First-Colloquim.git)