



НОВОГОДНЯЯ РАСПРОДАЖА!



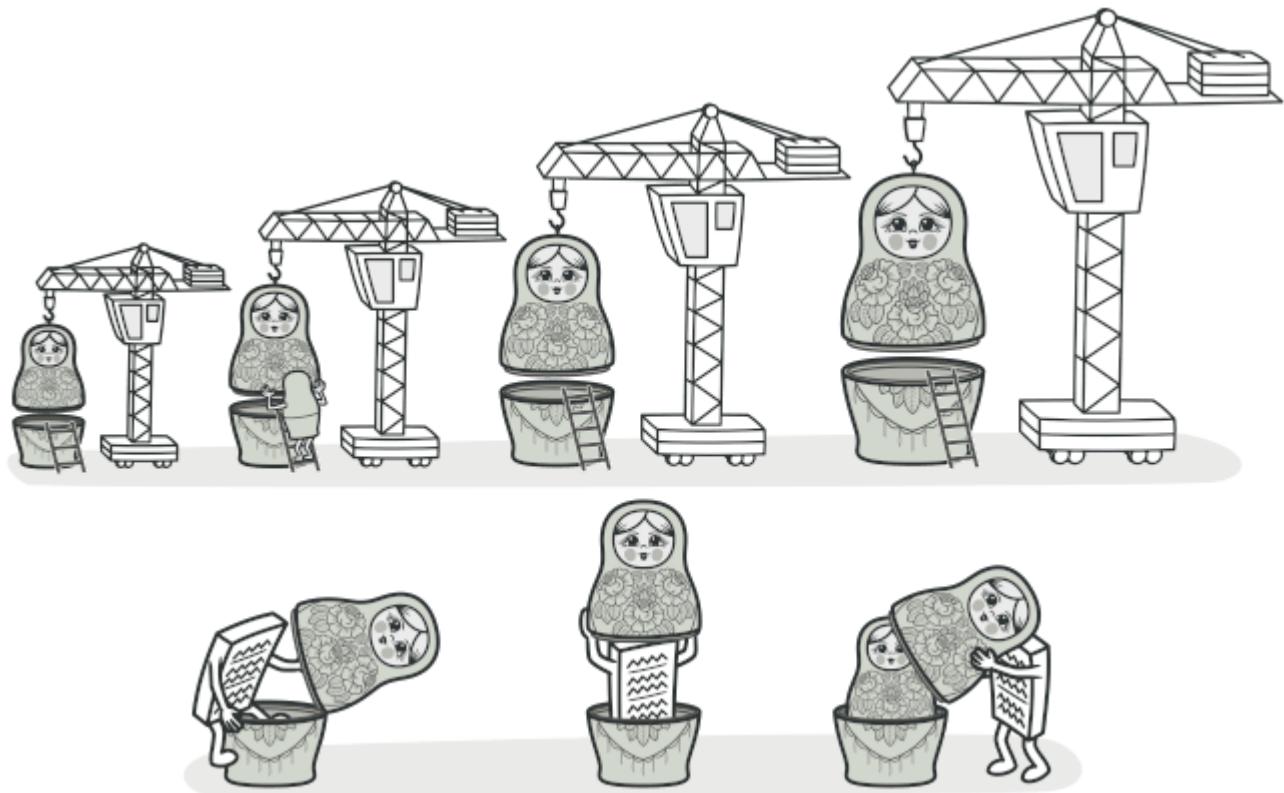
[Home](#) / Паттерны проектирования / Структурные паттерны

Декоратор

Также известен как: Wrapper, Обёртка, Decorator

💬 Суть паттерна

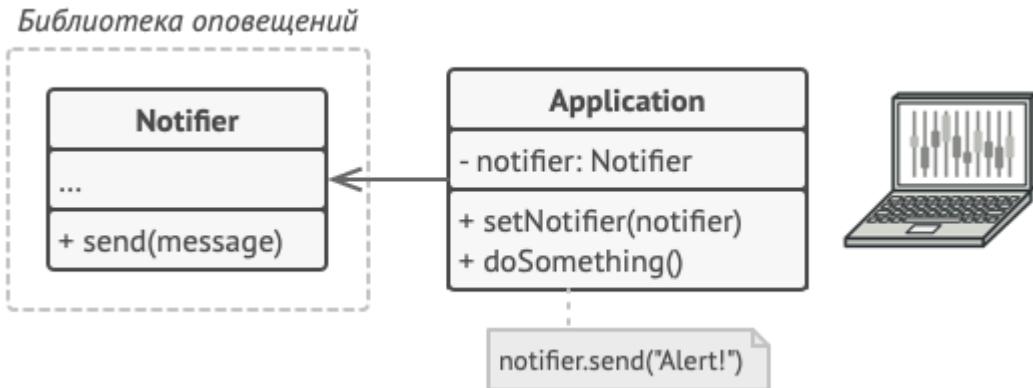
Декоратор – это структурный паттерн проектирования, который позволяет динамически добавлять объектам новую функциональность, оборачивая их в полезные «обёртки».



😢 Проблема

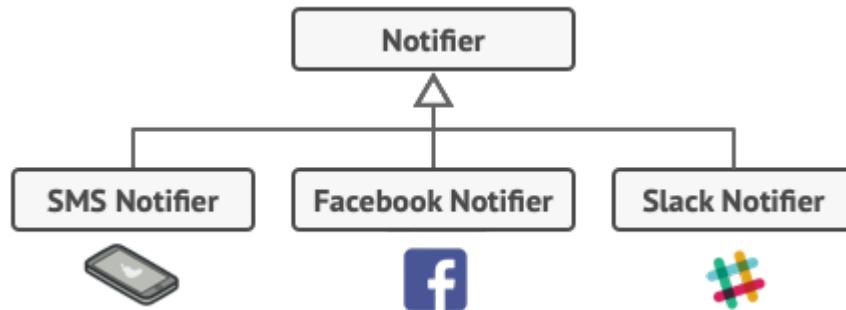
Вы работаете над библиотекой оповещений, которую можно подключать к разнообразным программам, чтобы получать уведомления о важных событиях.

 **НОВОГОДНЯЯ РАСПРОДАЖА!** 
 Структура сообщения и высылает ее всем администраторам по электронной почте. Сторонняя программа должна создать и настроить этот объект, указав кому отправлять оповещения, а затем использовать его каждый раз, когда что-то случается.



Сторонние программы используют главный класс оповещений.

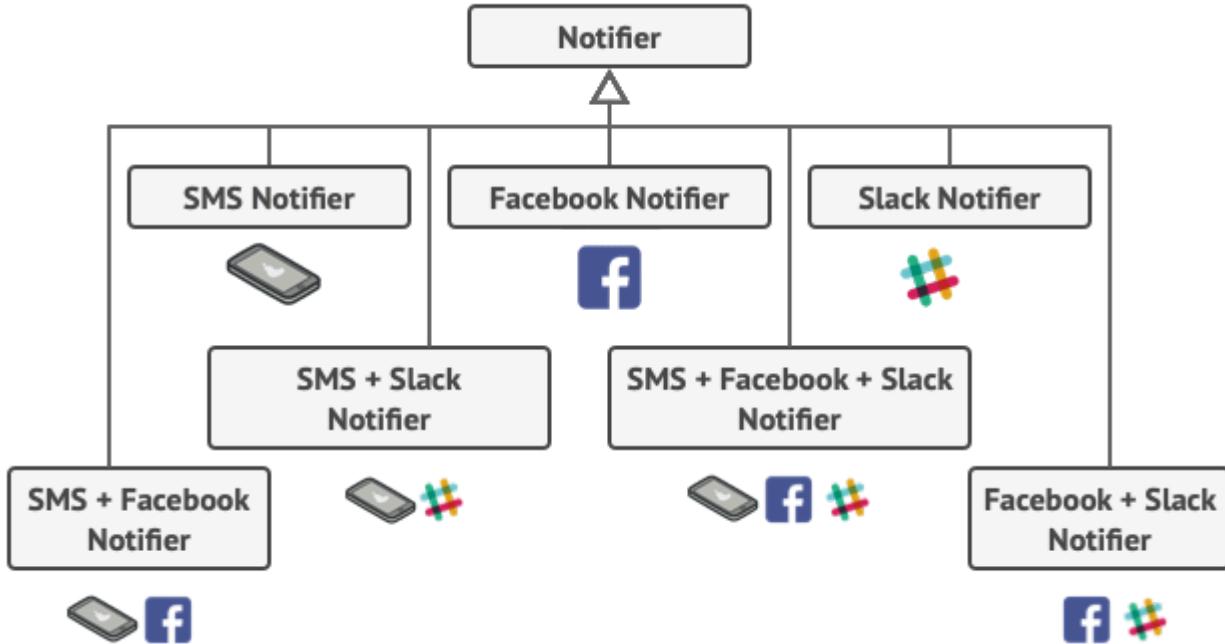
В какой-то момент стало понятно, что одних email-оповещений пользователям мало. Некоторые из них хотели бы получать извещения о критических проблемах через SMS. Другие хотели бы получать их в виде сообщений Facebook. Корпоративные пользователи хотели бы видеть сообщения в Slack.



Каждый тип оповещения живёт в собственном подклассе.

Сначала вы добавили каждый из этих типов оповещений в программу, унаследовав их от базового класса `Notifier`. Теперь пользователь выбирал один из типов оповещений, который и использовался в дальнейшем.

Но затем кто-то резонно спросил, почему нельзя выбрать несколько типов оповещений сразу? Ведь если вдруг в вашем доме начался пожар, вы бы хотели получить оповещения по всем каналам, не так ли?



Комбинаторный взрыв подклассов при совмещении типов оповещений.

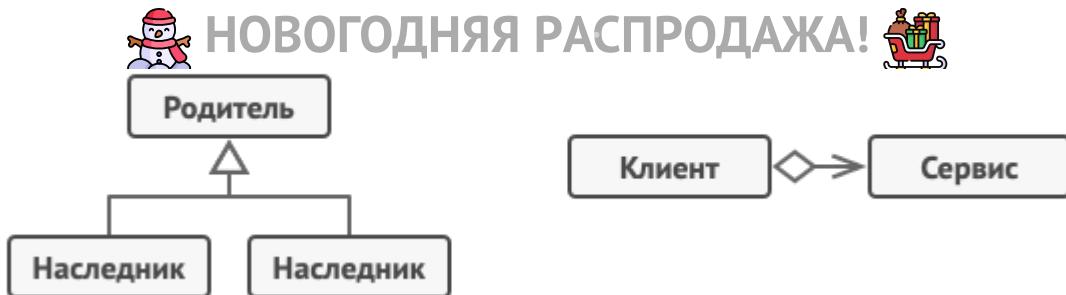
Итак, нужен какой-то другой способ комбинирования поведения объектов, который не приводит к взрыву количества подклассов.

😊 Решение

Наследование – это первое, что приходит в голову многим программистам, когда нужно расширить какое-то существующее поведение. Но 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 придется создавать множество подклассов-комбинаций для получения совмешённого поведения.

Одним из способов обойти эти проблемы является замена наследования *агрегацией* либо *композицией* ⓘ . Это когда один объект *содержит* ссылку на другой и делегирует ему работу, вместо того чтобы самому *наследовать* его поведение. Как раз на этом принципе построен паттерн Декоратор.

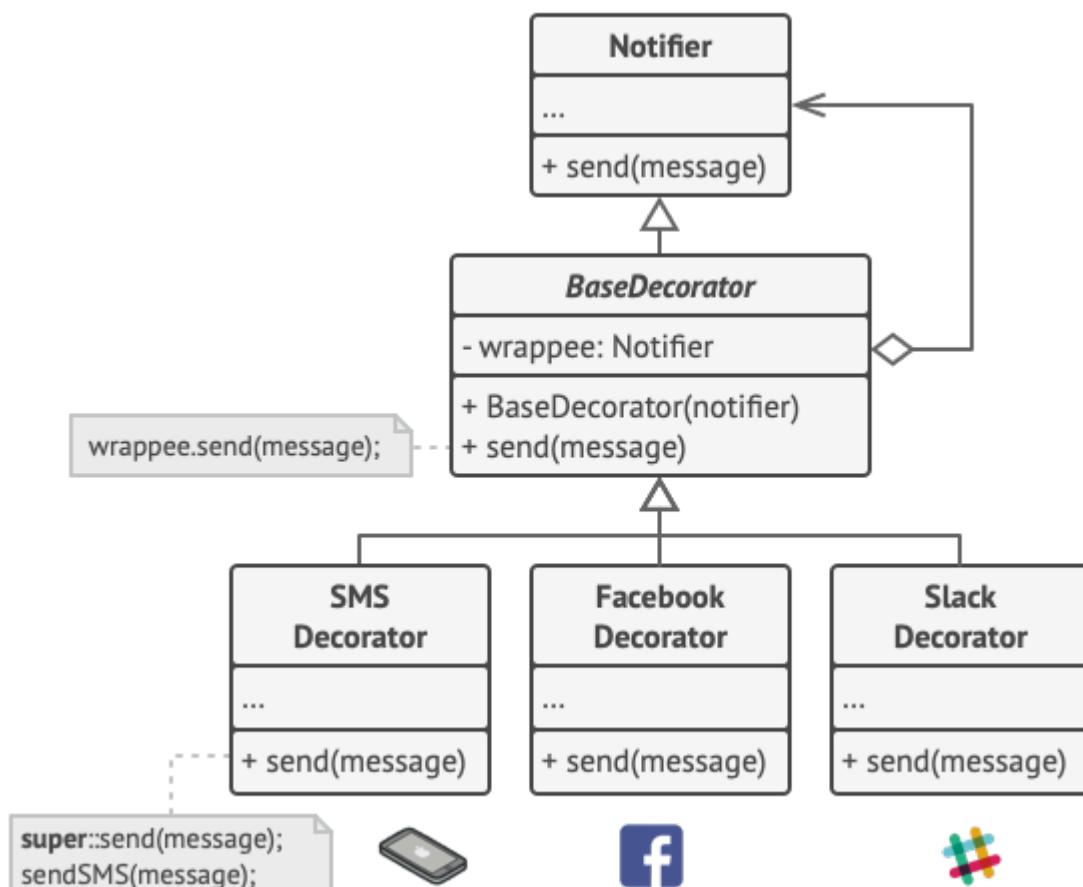


Наследование против Агрегации.

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.

В примере с оповещениями мы оставим в базовом классе простую отправку по электронной почте, а расширенные способы отправки сделаем декораторами.



Расширенные способы оповещения становятся декораторами.

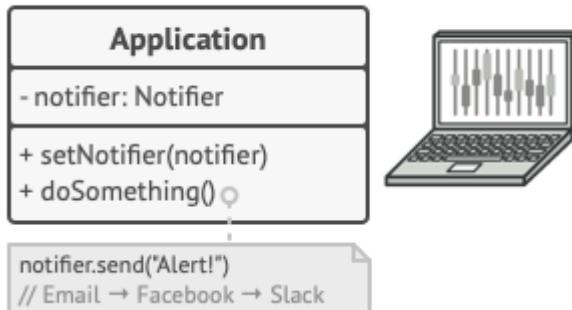
НОВОГОДНЯЯ РАСПРОДАЖА!

Заворачивайтъ своеу оповещенин в ти обёртки, котрые соответствуютъ желаемому стилюу оповещения.

```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```



Программа может составлять составные объекты из декораторов.

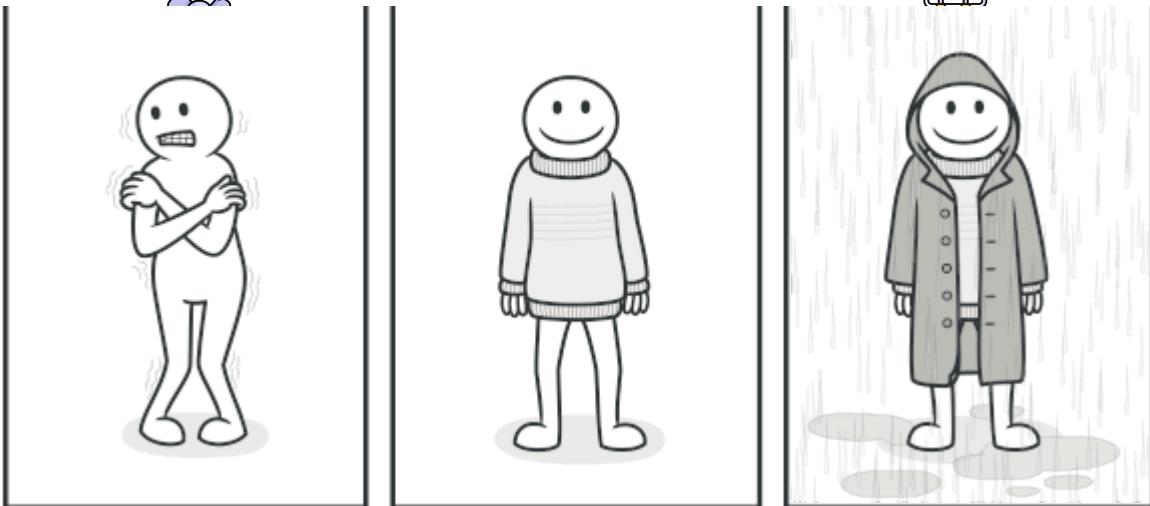
Последняя обёртка в списке и будет тем объектом, с которым клиент будет работать в остальное время. Для остального клиентского кода, по сути, ничего не изменится, ведь все обёртки имеют точно такой же интерфейс, что и базовый класс оповещений.

Таким же образом можно изменять не только способ доставки оповещений, но и форматирование, список адресатов и так далее. К тому же клиент может «дообернуть» объект любыми другими обёртками, когда ему захочется.

Аналогия из жизни



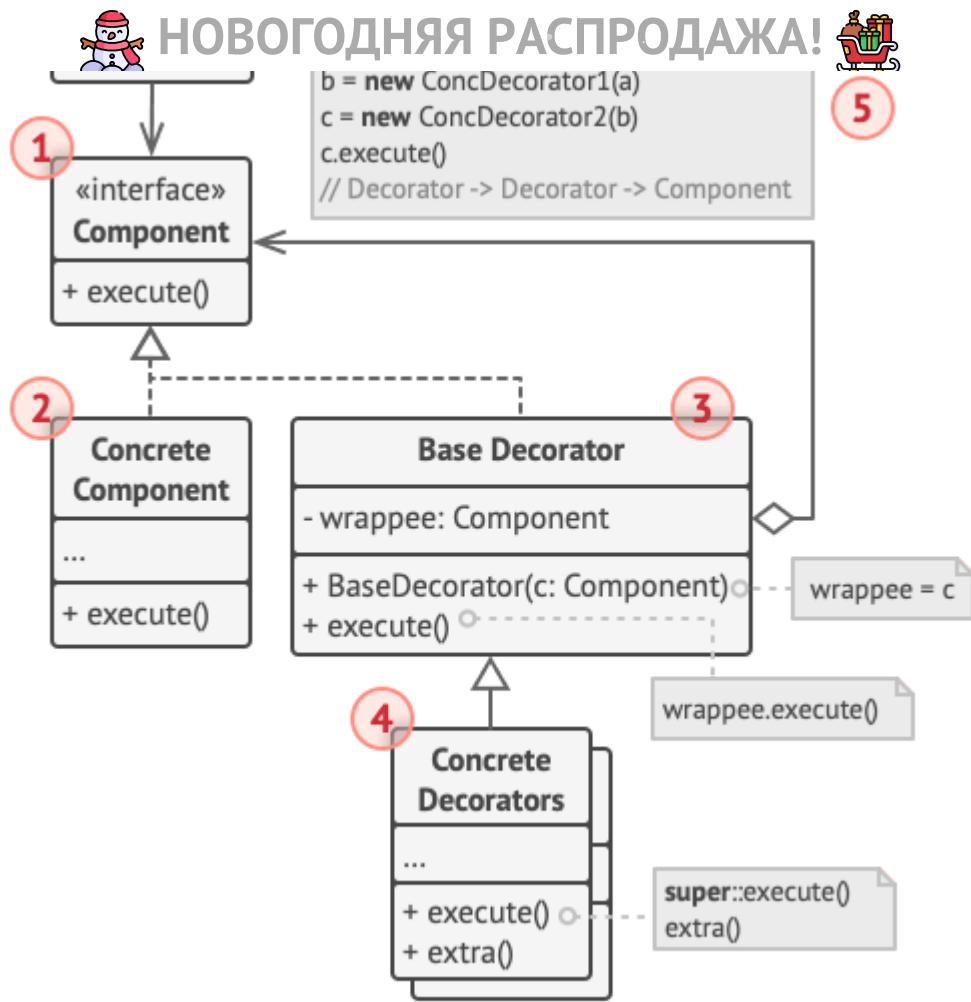
НОВОГОДНЯЯ РАСПРОДАЖА!



Одежда можно надевать слоями, получая комбинированный эффект.

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.

STRUCTURE

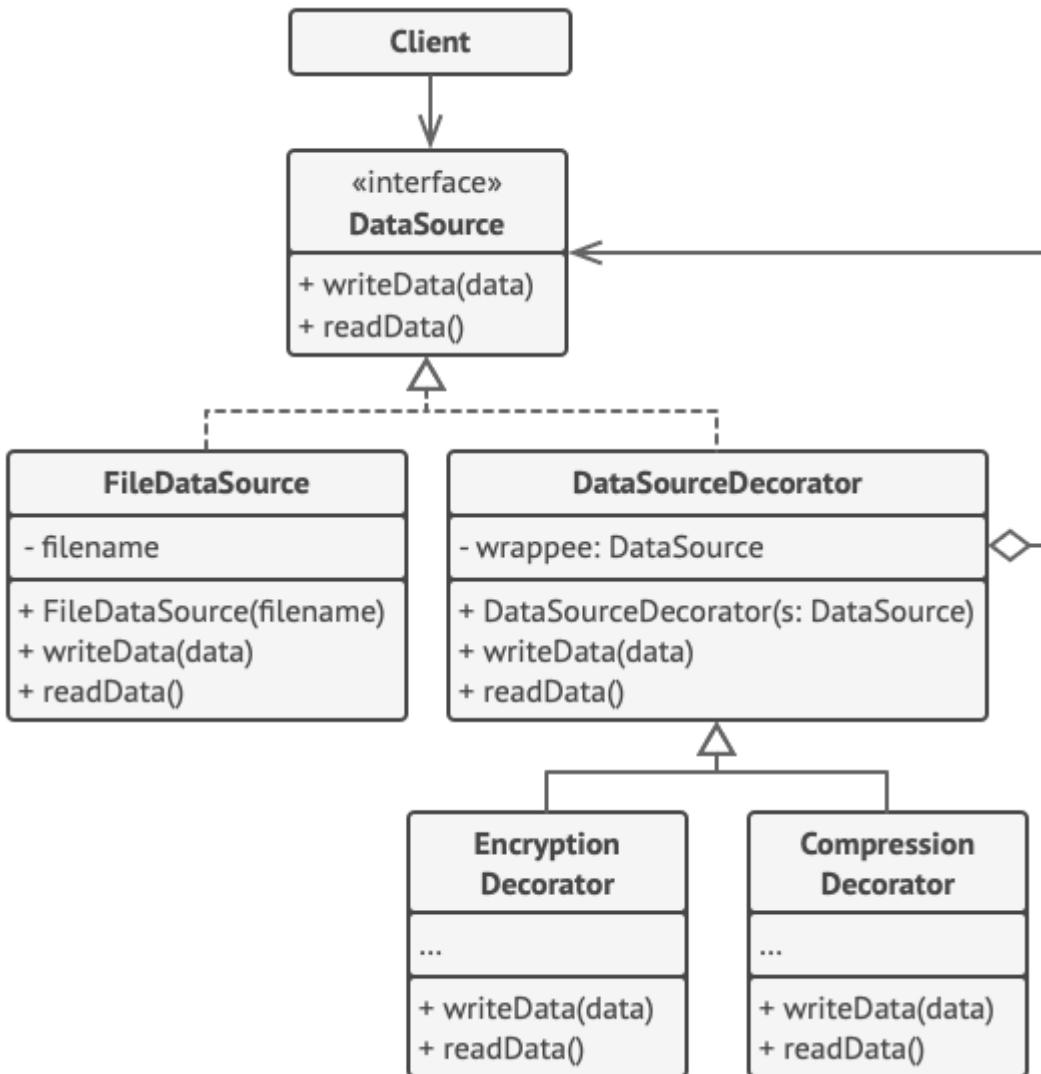


1. **Компонент** задаёт общий интерфейс обёрток и обываемых объектов.
2. **Конкретный компонент** определяет класс обываемых объектов. Он содержит какое-то базовое поведение, которое потом изменяют декораторы.
3. **Базовый декоратор** хранит ссылку на вложенный объект-компонент. Им может быть как конкретный 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 обрабатывать простые компоненты и декораторы в другие декораторы, работая со всеми объектами через общий интерфейс компонентов.

Псевдокод

НОВОГОДНЯЯ РАСПРОДАЖА!

Безопасности призываю для кода, который их использует.



Пример шифрования и компрессии данных с помощью обёрток.

Приложение оборачивает класс данных в шифрующую и сжимающую обёртки, которые при чтении выдают оригинальные данные, а при записи – зашифрованные и сжатые.

Декораторы, как и сам класс данных, имеют общий интерфейс. Поэтому клиентскому коду не важно, с чем работать – с «чистым» объектом данных или с «обёрнутым».

```

// Общий интерфейс компонентов.
interface DataSource is
    method writeData(data)
    method readData():data

// Один из конкретных компонентов реализует базовую
// функциональность.
class FileDataSource implements DataSource is
  
```



НОВОГОДНЯЯ РАСПРОДАЖА!



```

method writeData(data) is
    // Записать данные в файл.

method readData():data is
    // Прочитать данные из файла.

// Родитель всех декораторов содержит код обёртывания.
class DataSourceDecorator implements DataSource is
    protected field wrappee: DataSource

constructor DataSourceDecorator(source: DataSource) is
    wrappee = source

method writeData(data) is
    wrappee.writeData(data)

method readData():data is
    return wrappee.readData()

// Конкретные декораторы добавляют что-то своё к базовому
// поведению обёрнутого компонента.
class EncryptionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Зашифровать поданные данные.
        // 2. Передать зашифрованные данные в метод writeData
        // обёрнутого объекта (wrappee).

    method readData():data is
        // 1. Получить данные из метода readData обёрнутого
        // объекта (wrappee).
        // 2. Расшифровать их, если они зашифрованы.
        // 3. Вернуть результат.

// Декорировать можно не только базовые компоненты, но и уже
// обёрнутые объекты.
class CompressionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Запаковать поданные данные.
        // 2. Передать запакованные данные в метод writeData
        // обёрнутого объекта (wrappee).

    method readData():data is
        // 1. Получить данные из метода readData обёрнутого
        // объекта (wrappee).
        // 2. Распаковать их, если они запакованы.
        // 3. Вернуть результат.

// Вариант 1. Простой пример сборки и использования декораторов.
class Application is

```



НОВОГОДНЯЯ РАСПРОДАЖА!



```

source.writeData(salaryRecords)
// В файл были записаны чистые данные.

source = new CompressionDecorator(source)
source.writeData(salaryRecords)
// В файл были записаны сжатые данные.

source = new EncryptionDecorator(source)
// Сейчас в source находится связка из трёх объектов:
// Encryption > Compression > FileDataSource

source.writeData(salaryRecords)
// В файл были записаны сжатые и зашифрованные данные.

// Вариант 2. Клиентский код, использующий внешний источник
// данных. Класс SalaryManager ничего не знает о том, как именно
// будут считаны и записаны данные. Он получает уже готовый
// источник данных.

class SalaryManager is
    field source: DataSource

    constructor SalaryManager(source: DataSource) { ... }

    method load() is
        return source.readData()

    method save() is
        source.writeData(salaryRecords)
    // ...Остальные полезные методы...

// Приложение может по-разному собирать декорируемые объекты, в
// зависимости от условий использования.

class ApplicationConfigurator is
    method configurationExample() is
        source = new FileDataSource("salary.dat")
        if (enabledEncryption)
            source = new EncryptionDecorator(source)
        if (enabledCompression)
            source = new CompressionDecorator(source)

        logger = new SalaryManager(source)
        salary = logger.load()
    // ...

```

Применимость



НОВОГОДНЯЯ РАСПРОДАЖА!



которыми их используют.

-  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.

Шаги реализации

1. Убедитесь, что в вашей задаче есть один основной компонент и несколько опциональных дополнений или надстроек над ним.
2. Создайте интерфейс компонента, который описывал бы общие методы как для основного компонента, так и для его дополнений.
3. Создайте класс конкретного компонента и поместите в него основную бизнес-логику.
4. Создайте базовый класс декораторов. Он должен иметь поле для хранения ссылки на вложенный объект-компонент. Все методы базового декоратора должны делегировать действие вложенному объекту.
5. И конкретный компонент, и базовый декоратор должны следовать одному и тому же интерфейсу компонента.
6. Теперь создайте классы конкретных декораторов, наследуя их от базового декоратора. Конкретный декоратор должен выполнять свою добавочную функцию, а затем (или перед этим) вызывать эту же операцию обёрнутого объекта.
7. Клиент берёт на себя ответственность за конфигурацию и порядок обёртывания объектов.

Преимущества и недостатки



НОВОГОДНЯЯ РАСПРОДАЖА!

ВСЕРНУТЫЕ ОБЪЕКТЫ.



- ✓ Позволяет добавлять обязанности на лету.
- ✗ Обилие крошечных классов.
- ✓ Можно добавлять несколько новых обязанностей сразу.
- ✓ Позволяет иметь несколько мелких объектов вместо одного объекта на все случаи жизни.

↔ Отношения с другими паттернами

- **Адаптер** предоставляет совершенно другой интерфейс для доступа к существующему объекту. С другой стороны, при использовании паттерна **Декоратор** интерфейс либо остается прежним, либо расширяется. Причём *Декоратор* поддерживает рекурсивную вложенность, чего не скажешь об *Адаптере*.
- С **Адаптером** вы получаете доступ к существующему объекту через другой интерфейс. Используя **Заместитель**, интерфейс остается неизменным. Используя **Декоратор**, вы получаете доступ к объекту через расширенный интерфейс.
- **Цепочка обязанностей** и **Декоратор** имеют очень похожие структуры. Оба паттерна базируются на принципе рекурсивного выполнения операции через серию связанных объектов. Но есть и несколько важных отличий.

Обработчики в *Цепочке обязанностей* могут выполнять произвольные действия, независимые друг от друга, а также в любой момент прерывать дальнейшую передачу по цепочке. С другой стороны *Декораторы* расширяют какое-то определённое действие, не ломая интерфейс базовой операции и не прерывая выполнение остальных декораторов.

- **Компоновщик** и **Декоратор** имеют похожие структуры классов из-за того, что оба построены на рекурсивной вложенности. Она позволяет связать в одну структуру бесконечное количество объектов.

Декоратор оборачивает только один объект, а узел *Компоновщика* может иметь много детей. *Декоратор* добавляет вложенному объекту новую функциональность, а *Компоновщик* не добавляет ничего нового, но «суммирует» результаты всех своих детей.

Но они могут и сотрудничать: *Компоновщик* может использовать *Декоратор*, чтобы переопределить функции отдельных частей дерева компонентов.



- **Стратегия** меняет поведение объекта «изнутри», а **Декоратор** изменяет его «снаружи».
- **Декоратор** и **Заместитель** имеют схожие структуры, но разные назначения. Они похожи тем, что оба построены на принципе композиции и делегируют работу другим объектам. Паттерны отличаются тем, что **Заместитель** сам управляет жизнью сервисного объекта, а обёртывание **Декораторов** контролируется клиентом.

</> Примеры реализации паттерна



Не втыкай в транспорте

Лучше почитай нашу книгу о паттернах проектирования.

Теперь это удобно делать даже во время поездок в общественном транспорте.

Узнать больше...