9. Шаблон Декоратор

ЛЕКЦИОНЕН КУРС: ШАБЛОНИ ЗА ПРОЕКТИРАНЕ

ДОЦ. Д-Р ЕМИЛ ДОЙЧЕВ

Общи сведения

- ✓ **Вид:** структурен за обекти
- ✓ **Цел:** Динамично възлагане на допълнителни отговорности на даден обект. Алтернатива на създаването на подкласове за разширяване на функционалността.
- ✓ **Известен и като:** Обвивка (wrapper)

Мотивация

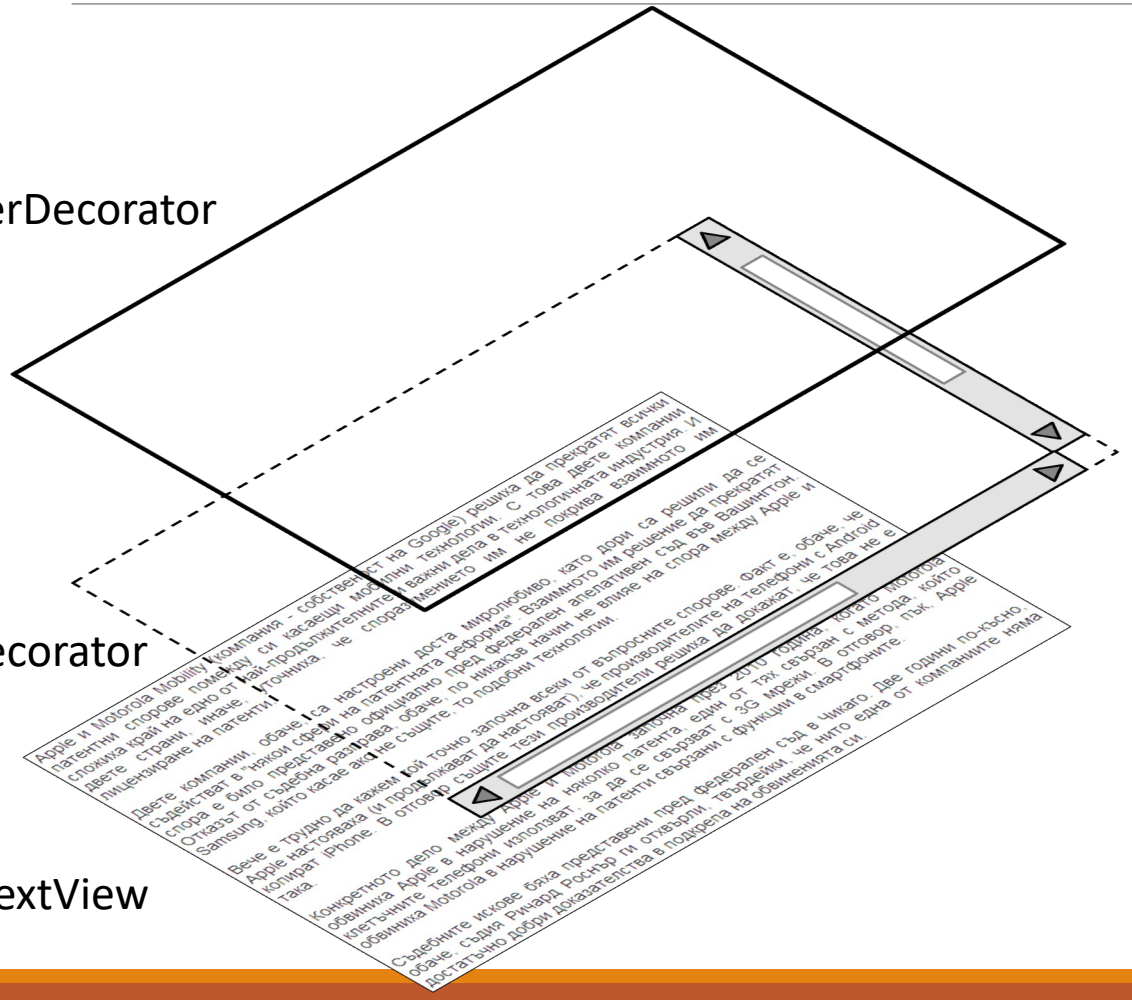
- **Мотивация:** Добавяне на задължения (свойства) към отделни обекти, а не към целия клас.
- Напр. добавяне на атрибути като рамка или scrollbar към графичен компонент.
  - можем да го направим с наследяване, но това ще ограничи гъвкавостта
  - друга възможност е, чрез използване на композиция

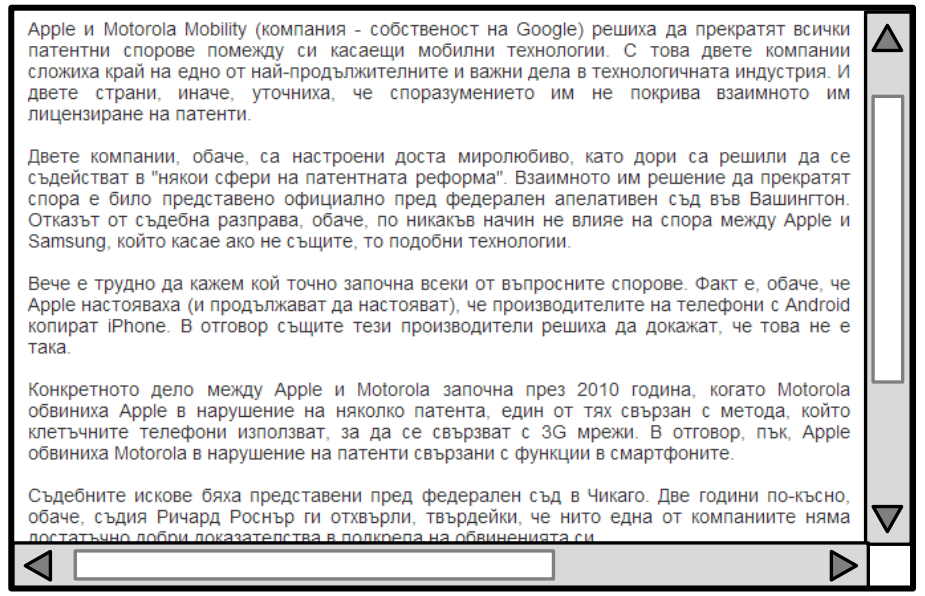
Мотивация

aBorderDecorator

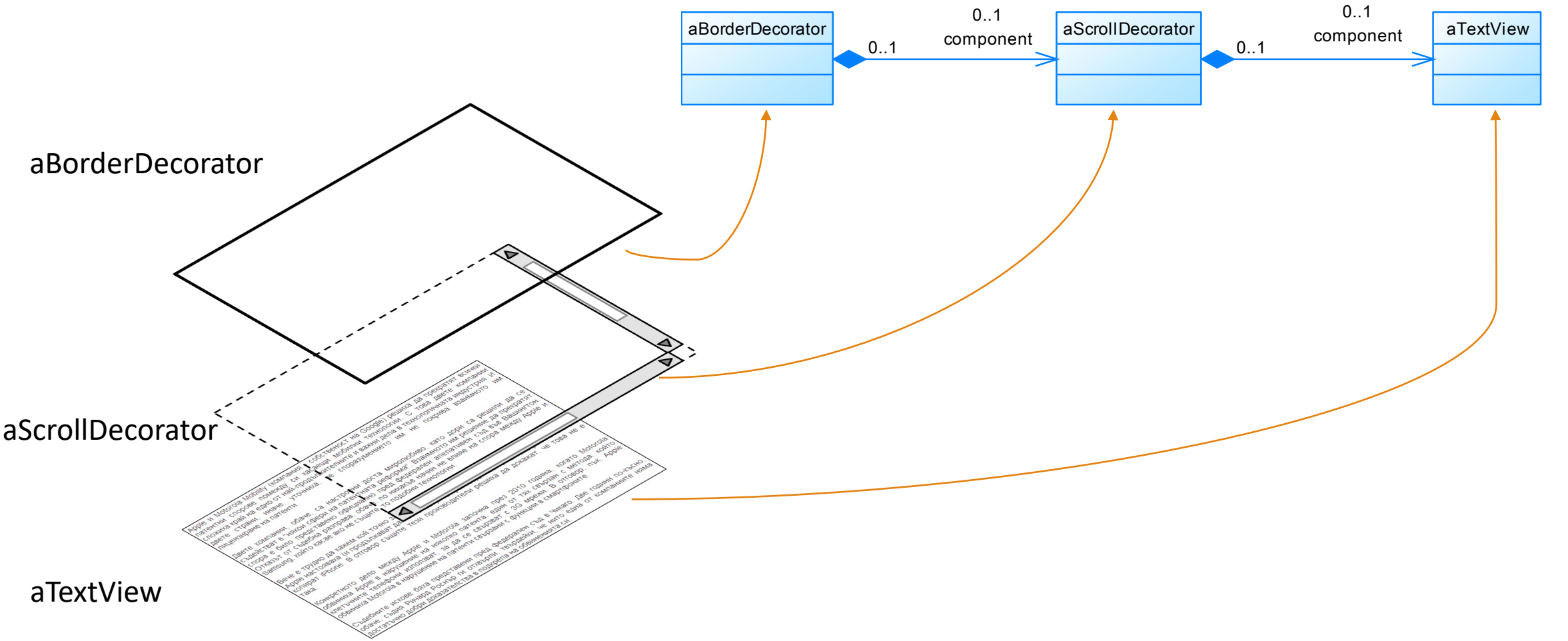
aScrollDecorator

aTextView

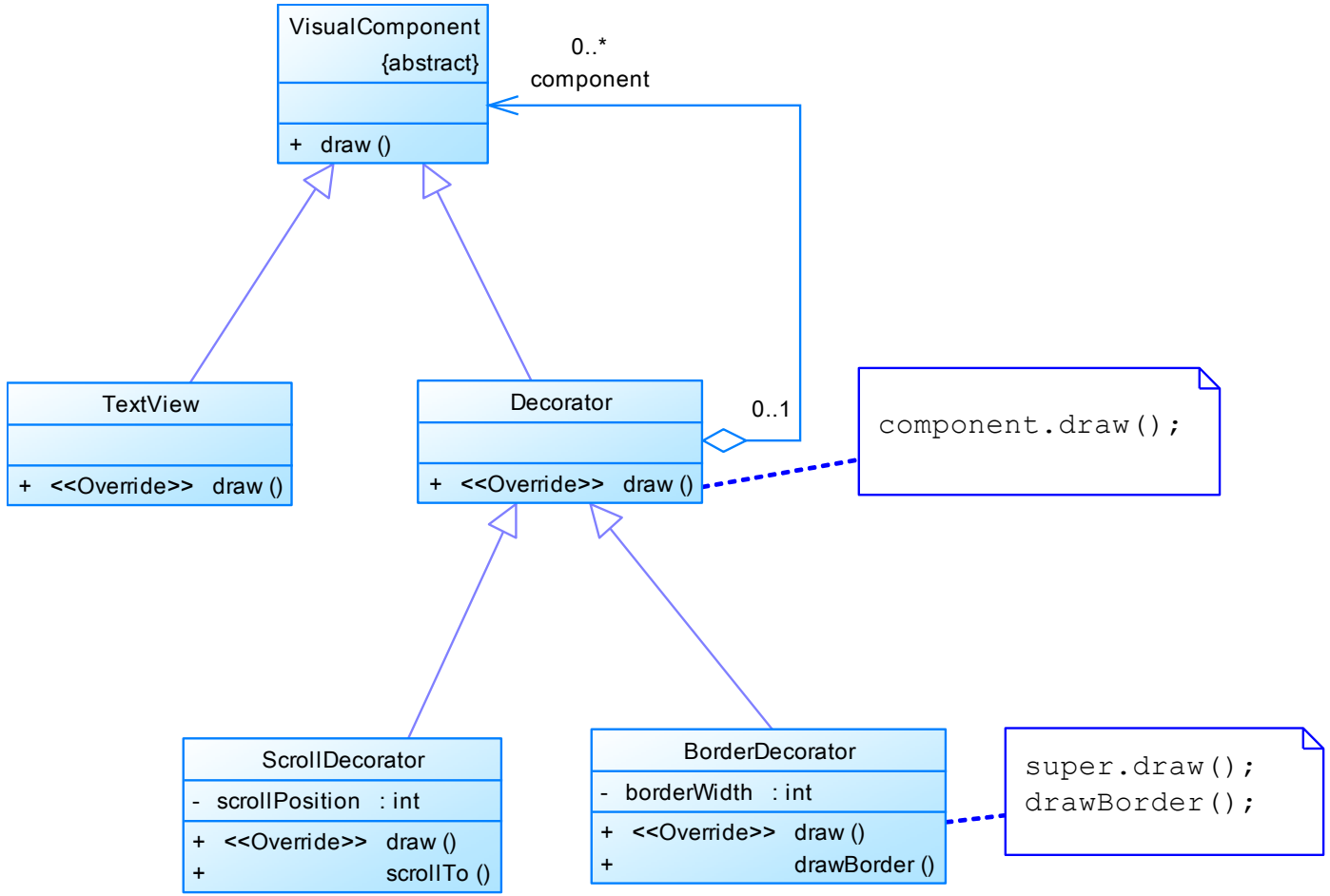




Мотивация


aBorderDecorator
aScrollDecorator
aTextView
aBorderDecorator
0..1
0..1
component
aScrollDecorator
0..1
0..1
component
aTextView

Мотивация

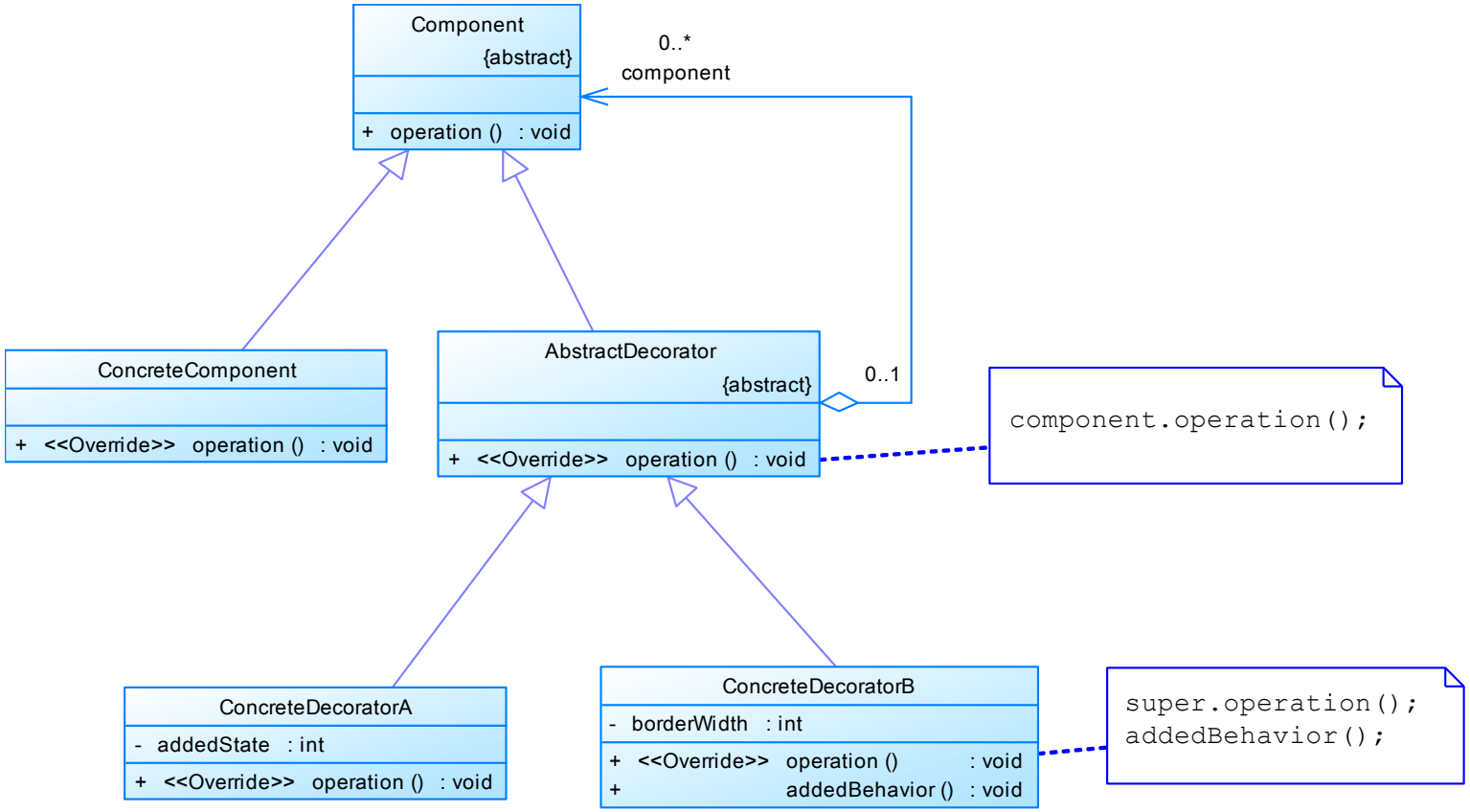


- ✓ Декораторът се придържа към интерфейса на обекта, който декорира, така че присъствието му да е прозрачно за клиентите на компонента.
- ✓ Декораторът прехвърля заявки към компонента и може да извършва допълнителни действия.
- ✓ Прозрачността дава възможност за рекурсивно наслагване на декоратори – т.е. възможно е добавяне на безкраен брой допълнителни задължения/свойства към компонента.
- ✓ Декораторите могат да бъдат навсякъде, където може да бъде VisualComponent.

Приложимост

- ✓ **Приложимост:** Шаблонът Декоратор се използва в следните случаи:
  - За да се добавят динамично и прозрачно задължения (функционалност, свойства) към отделни обекти, без да се засягат другите обекти от същия клас.
  - Когато разширяването чрез създаване на подкласове не е практично. Понякога е възможен голям брой независими разширения и това води до експлозия на подкласове, които трябва да поддържат всяка възможна комбинация.

Структура


Component
{abstract}
+ operation () : void
0..*
component
ConcreteComponent
+ <<Override>> operation () : void
AbstractDecorator
{abstract}
+ <<Override>> operation () : void
0..1
component.operation();
ConcreteDecoratorA
- addedState : int
+ <<Override>> operation () : void
ConcreteDecoratorB
- borderWidth : int
+ <<Override>> operation () : void
+ addedBehavior () : void
super.operation();
addedBehavior();

Участници

- ✓ **Component** (VisualComponent) – дефинира интерфейс за обекти, към които могат да се добавят динамични задължения.
- ✓ **ConcreteComponent** (TextView) – дефинира обект, към който могат да се добавят допълнителни задължения.
- ✓ **AbstractDecorator** (Decorator) – поддържа референция към обект от тип Component и дефинира интерфейс, отговарящ на интерфейса на Component.
- ✓ **ConcreteDecorator** (BorderDecorator, ScrollDecorator) – добавя задължения към компонента

Взаимодействия

- ✓ AbstractDecorator (и наследниците му) предава заявки към своя обект Component.
- ✓ Има възможност за извършване на допълнителни операции преди и след подаването на заявката.

Следствия

- ✓ **Предимства**
  - *Предлага по-голяма гъвкавост от статичното наследяване.* Задълженията могат да се добавят и премахват по време на изпълнението просто чрез закачането и откачането им. Възможно е и добавянето на свойства повече от един път.
  - *Избягва претоварването на класовете нагоре в йерархията.* Вместо да се поддържат всички предвидими възможности в един сложен клас с възможност за настройка, се дефинира един прост клас и постепенно се добавя функционалност чрез декоратори.
- ✓ **Недостатъци**
  - *Декораторът и неговият компонент не са идентични.* Декораторът служи като прозрачна опаковка, но от гледна точка на идентичността на обекта, един декориран компонент не е идентичен със самия компонент. Т.е. не може да се разчита на идентичността на компонента.
  - *Множество малки обекти.* Дизайнът с шаблона Декоратор често води до системи, композирани от множество малки обекти, които си приличат. Обектите се различават само по начина, по който са взаимосвързани, а не по класа си или стойностите на техните атрибути. Тези системи могат да са трудни за изучаване и откриване на грешки.

Пример

- ✓ Класовете в Java I/O библиотеката използват шаблона Декоратор.
- ✓ Основните I/O класове са: InputStream, OutputStream, Reader и Writer. Тези класове имат само базова функционалност.
- ✓ Можем да добавяме допълнителни поведения към един съществуващ поток, например:
  - Buffered Stream – добавя буфериране за потока
  - Data Stream – позволява вход/изход на примитивни типове данни в Java
  - Pushback Stream – позволява операция за възстановяване (undo)
- ✓ Не бихме искали да модифицираме основните I/O класове за да постигнем тази функционалност, затова използваме класове декоратори (които в Java се наричат филтри) за да добавим нужните свойства чрез композиция.

Пример

- ✓ Някой примери за декоратор (филтър) класове са:
  - BufferedInputStream
  - DataInputStream
  - PushbackInputStream
- ✓ Конструктора на тези класове приема обект от тип InputStream

Пример

```
public class DecoratorDemo {

    /**
     * @param args
     * @throws FileNotFoundException
     */
    public static void main(String[] args) throws FileNotFoundException {

        // Отваряне на InputStream.
        FileInputStream in = new FileInputStream("test.dat");

        // Създаване на буфериран InputStream.
        BufferedInputStream bin = new BufferedInputStream(in);

        // Създаване на буфериран InputStream за данни
        DataInputStream dbin = new DataInputStream(bin);

        // Създаване на небуфериран InputStream за данни
        DataInputStream din = new DataInputStream(in);

        // Създаване на буфериран, pushback InputStream за данни
        PushbackInputStream pbdbin = new PushbackInputStream(dbin);
    }

}
```

Край: Шаблон Декоратор

ЛЕКЦИОНЕН КУРС: ШАБЛОНИ ЗА ПРОЕКТИРАНЕ