

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



Apple и Motorola Mobility (компания - собственост на Google) решиха да прекратят всички патентни спорове помежду си касаещи мобилни технологии. С това двете компании сложиха край на едно от най-продължителните и важни дела в технологичната индустрия. И двете страни, иначе, уточниха, че споразумението им не покрива взаимното им лицензиране на патенти.

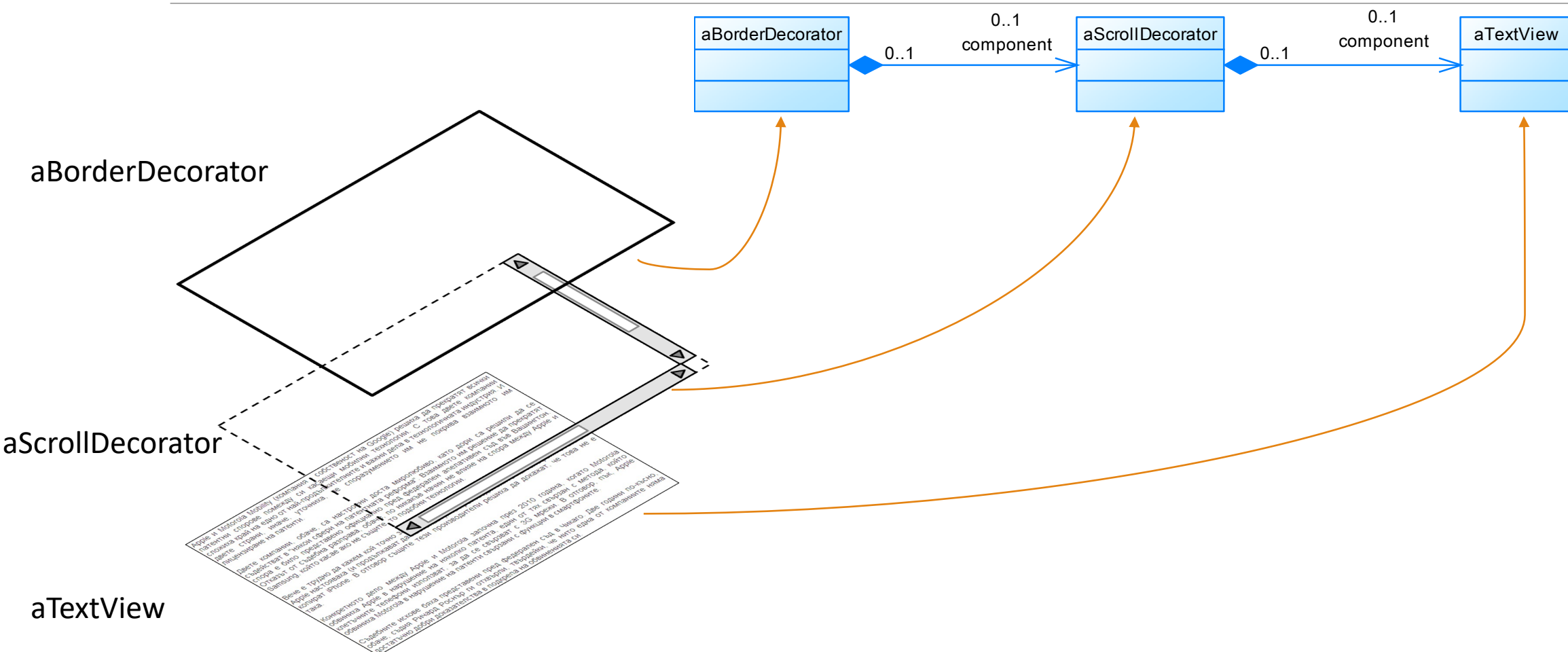
Двете компании, обаче, са настроени доста миролюбиво, като дори са решили да се съдействат в "някои сфери на патентната реформа". Взаимното им решение да прекратят спора е било представено официално пред федерален апелативен съд във Вашингтон. Отказът от съдебна разправа, обаче, по никакъв начин не влияе на спора между Apple и Samsung, който касае ако не същите, то подобни технологии.

Вече е трудно да кажем кой точно започна всеки от въпросните спорове. Факт е, обаче, че Apple настояваха (и продължават да настояват), че производителите на телефони с Android копират iPhone. В отговор същите тези производители решиха да докажат, че това не е така.

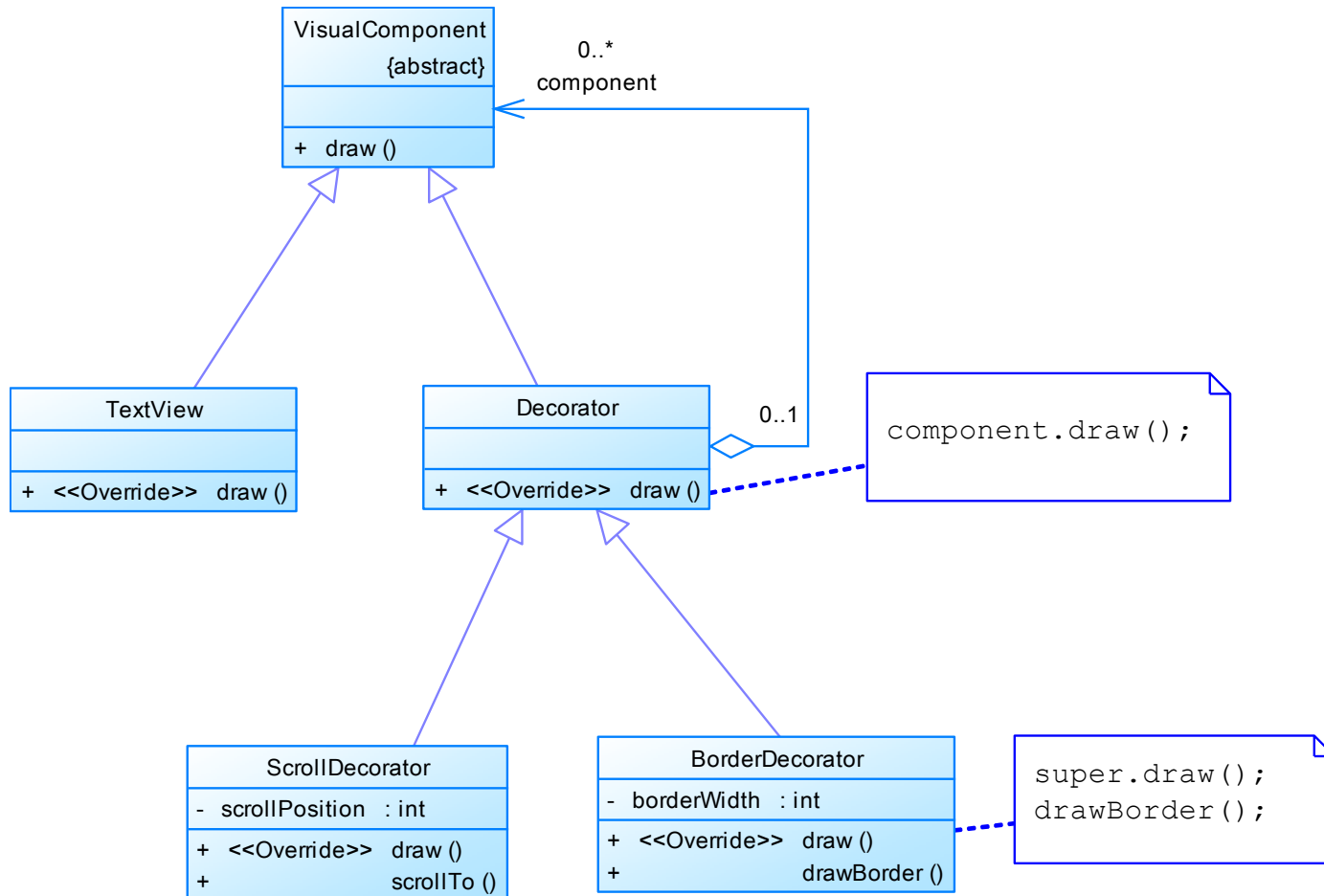
Конкретното дело между Apple и Motorola започна през 2010 година, когато Motorola обвиниха Apple в нарушение на няколко патента, един от тях свързан с метода, който клетъчните телефони използват, за да се свързват с 3G мрежи. В отговор, пък, Apple обвиниха Motorola в нарушение на патенти свързани с функции в смартфоните.

Съдебните искове бяха представени пред федерален съд в Чикаго. Две години по-късно, обаче, съдия Ричард Роснър ги отхвърли, твърдейки, че нито една от компаниите няма достатъчно добри доказателства в подкрепа на обвиненията си.

Мотивация



Мотивация

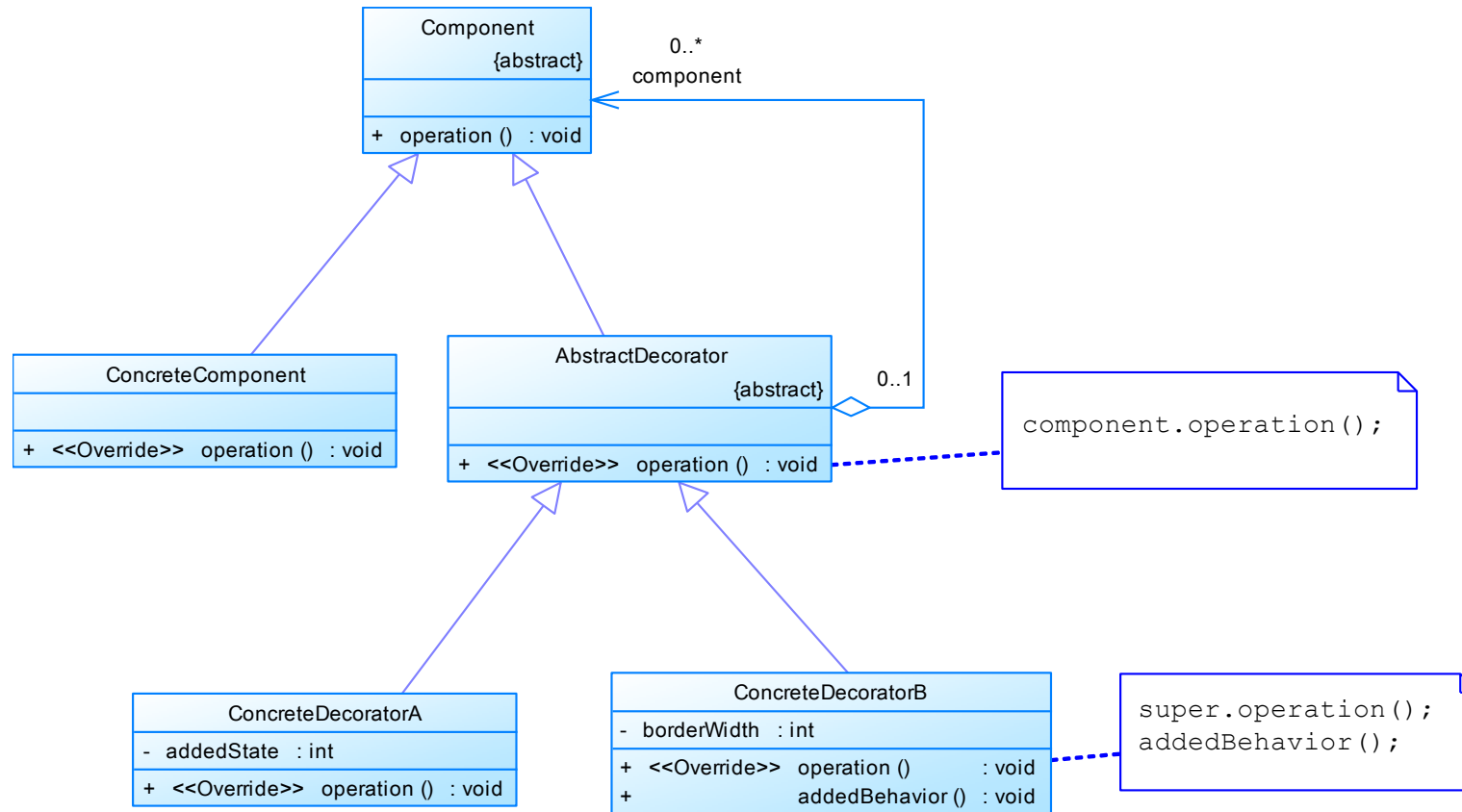


- ✓ Декораторът се придържа към интерфейса на обекта, който декорира, така че присъствието му да е прозрачно за клиентите на компонента.
- ✓ Декораторът прехвърля заявки към компонента и може да извършва допълнителни действия.
- ✓ Прозрачността дава възможност за рекурсивно наслагване на декоратори – 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



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

- ✓ Някои 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 pddbin = new PushbackInputStream(dbin);  
    }  
}
```

Край: Шаблон Декоратор

ЛЕКЦИОНЕН КУРС: ШАБЛОНИ ЗА ПРОЕКТИРАНЕ