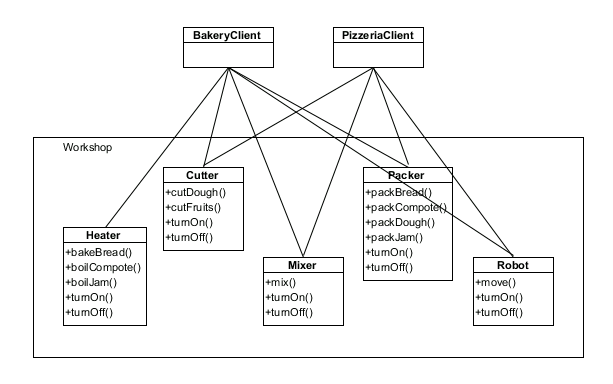
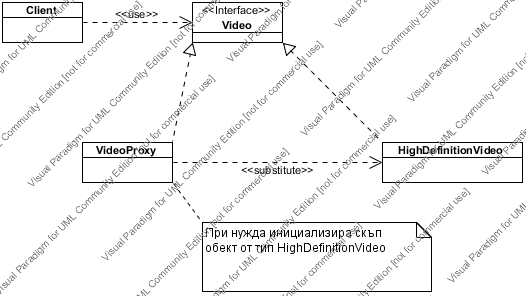
**Façade**

1. *Цел на шаблона* - предоставяне на интерфейс за достъп до сложна системата от класове.
2. *Структура:*



1. *Приложения*: - Уеб услугите (web services) могат да се разглеждат като реализация на фасада, когато нямат собствена бизнес логика, а са дефинирани за предоставяне на достъп до дадена система.
2. *Свързани шаблони* – “Abstract Factory” и “Sigleton”

**Proxy**

1. *Цел на шаблона* - предоставяне на заместник или контейнер за даден реален обект, позволявайки контролиран достъп до него.
2. *Структура:* 
3. *Приложения*: - има множество приложения – примерно: Aspect - Oriented Programming – AOP, Java RMI и т.н.
4. *Свързани шаблони –* „Adaptor” и “Decorator”

**Decorator**

**Decorator pattern** се използва за добавянето на дадена функционалност/

отговорност към вече съществуваш обект. При неговото използване се запазва open-close

принципа. Кодът ни остава затворен за промени, но едновременно с това отворен за

допълнения и extentions.

Един от начините да се добави функционалност е статичният, класически метод

чрез наследяване. При наличието на много обекти, това не е оптимален вариан, тъй като

ако искаме даден клас да има дадена функионалност, а да му липсва друга, ние сме

принудени да декларираме нов клас.!

Решението, което Decorator pattern-a предлага е да се работи на абстрактно ниво

чрез вкарването на нашият обект в друг(наричащ се декоратор), който да му добавя

дадената функционалност. Декораторът имплементира интерфейса на обекта, който

приема и по този начин клиентите дори не разбират за неговото съществуване.

**Кога да го използваме:!**

1. Когато искаме динамично да добавяме различни функционалности/отговорности към

даден обект!

2. За да добавим възможността run-time да премахваме различни функционалности/

отговорности!

3. Когато имплементацията чрез екстендване на base класовете не е практичка

**Какво получаваме:**

1. По-гъвкава имплементация, съответветно по-малко статична

2. Не пренатоварваме base класовете с ненужни полета и методи

3. Множество от малки класове, което може да доведе до по-трудно разбиране на

логиката в програмата и по-трудно дебъгване

4. Тъй като декорираният обект и основният такъв не са еднакви, трябва да се

имплементира начин за различаването им

**Примерна имплементация:**

1. Решавате кой клас/обект евентуално би имал нужда от добавяне на поведение/

функционалности. За целите на примера ще използвам FastFood. Ще имаме няколко

типа FastFood - пици и сандвичи.

2. Създавам абстрактен клас със същото име - FastFood. Други два класа, Pizza и

Sandwich, го наследяват.

3. Създаваме абстрактен Decorator class на име Ingredient, наследяващ FastFood.

Неговият конструктор ще приема обект от тип FastFood и ще го пази в променлива.!

4. Нека създадем сандвич с домати, шунка и кашкавал: Първо добавяме 3 конкретни

имплементации на добавките чрез екстендване на абстрактния клас Ingredient: public

class Tomato : Ingredient {…}, Ham, Cheese;

5. След като сме имплементирали горната логика можем директно да създадем сандвич

или пица по следния начин:

FastFood sandwich = new Tomato(new Ham(new Cheese(new Sandwich)));

По този начин първо се създава сандвич, който се пуска към Cheese декоратор,

който го обработва и му добавя нови функции. Например, ако ние продаваме нашия

сандвич и цената му зависи от продуктите, които слагаме вътре, то добавянето на

кашкавал би увеличило тази цена.

Ако имаме много различни видове сандвичи и пици за създаване, бихме могли да

комбинираме този pattern с Builder pattern за по-оптимално решение

**Свързани патърни:**

Composite, Adapter, Strategy

