PROGRAMOWANIE ASPEKTOWE I WZORCE PROJEKTOWE

ASPECT-ORIENTED PROGRAMMING AND DESIGN PATTERNS

PROGRAMOWANIE ZORIENTOWANE ASPEKTOWO

- Pozwala oddzielić logikę biznesową od kodu, który jest częścią wszystkich aplikacji
- Przykładowo logowanie zdarzeń lub obsługa wyjątków może być oddzielona od kodu biznesowego
- Pozwala dołączyć kod do już istnejącego kodu źródłowego bez potrzeby jego zmiany

KOMPILACJA A WYKONANIE

- AOP polega na modyfikacji kodu klasy podczas kompilacji lub wywoływaniu kodu przed lub po źródłowym kodzie biznesowym
- W Javie do implementacji AOP wykorzystywane są Eclipse AspectJ lub Spring

PROGRAMOWANIE ASPEKTOWE W JAVIE EE - INTERCEPTOR

Interceptory:

- Pozwalają rozdzielić logikę biznesową od dodatkowych zagadnień
- Są częścią Javy EE
- Pozwalają napisać kod wykonywany bezpośrednio przed wywołaniem metody lub (i) bezpośrednio po wywołaniu metody
- Działają jak filtry przechwytujące

Typowe filtry przechwytujące:

- Filtry uwierzytelniania i autoryzacji
- Filtry tworzenia dzienników wywołań i logów audytowych
- Filtry konwersji obrazów
- Filtry kompresji danych
- Filtry szyfrujące
- Filtry tworzenia tokenów

KILKA SŁÓW NA TEMAT CDI I KOMPONENTÓW BEAN

CONTEXT AND DEPENDENCY INJECTION (CDI)

- Wstrzykiwanie danej implementacji w konkretne miejsce
- Filar Javy EE – tworzone usługi zależą od mechanizmów CDI
- Pozwala uniknąć silnych powiązań pomiędzy klasami

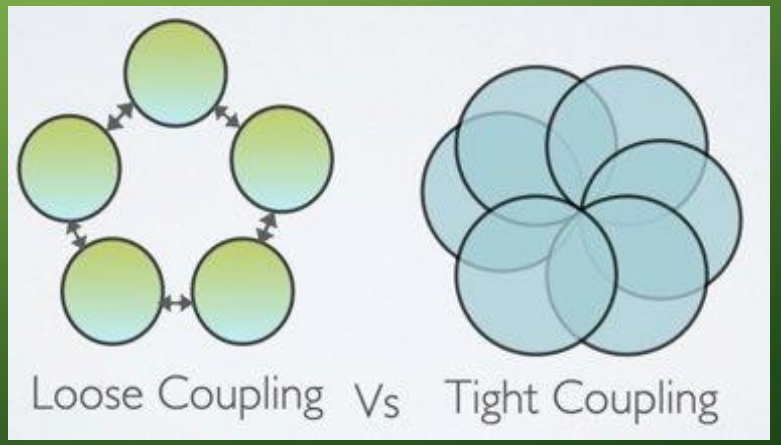
BEAN

- Każdy obiekt o dobrze zdefiniowanym cyklu życia i kontekście
- W Javie EE znany pod postacią tzw. beana zarządzanego; zarządza nim kontener bez dużego udziału programisty, ma dobrze zdefiniowany cykl życia
- Może być wstrzykiwany do innych beanów lub obiektów Javy EE
- W celu zamiany klasy na zarządzany bean używa się adnotacji @ManagedBean

LUŹNE POWIĄZANIE

CZYM JEST LOOSE COUPLING?

- Jest to sytuacja, w której dwie klasy, moduły lub komponenty są w niewielki sposób od siebie zależne. W Javie oznacza to, że dwie klasy są od siebie niezależne. Pozwala ono na wprowadzanie nowych funkcjonalności lub naprawianie kodu w danej warstwie w niewielki sposób wpływając na pozostałe warstwy.



- W JEE połączenie zdarzeń, interceptorów i dekoratorów zapewnia dodatkowe luźne powiązanie.
  - Notyfikacje zdarzeń

    Ustanawiany jest decoupling pomiędzy klientami generującymi zdarzenia oraz obserwatoram, które nasłuchują przychodzących zdarzeń
  - Interceptory

    Oddzielają techniczne aspekty od logiki biznesowej
  - Dekoratory

    Umożliwiają rozszerzanie logiki biznesowej

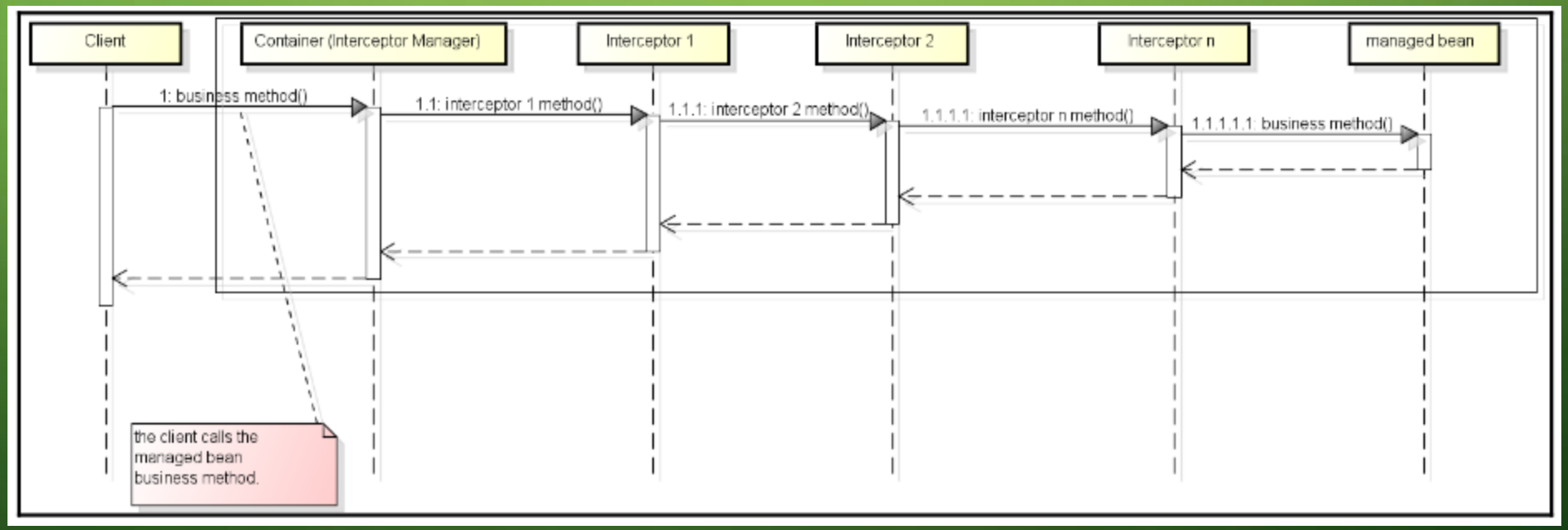
INTERCEPTORY W JEE

IMPLEMENTACJA INTERCEPTOR

- Interceptor może być zdefiniowany w klasie docelowej lub jako osobna klasa, która posiada metody "przechwytujące". Te są wywołane kiedy klasa docelowa jest oznaczana jako "intercepted".
- Każdy element w momencie wywołania wywołuje interceptor. Punktem w którym następuje rozpoczęcie interceptora nazywany jest **pointcut**

WYWOŁANIE INTERCEPTORÓW

Interceptory mogą być związane ze sobą. Wywołanie metody biznesowej, która jest przechwytywana uruchamia interceptor, po nim kolejny aż do interceptora N. Po nim następuje właściwe wykonanie metody biznesowej



ADNOTACJE

- **AroundConstruct** – wywołana zwrotnie przez konstruktor klasy docelowej
- **AroundInvoke** - wywołana kiedy metoda oznaczona jako przechwytywana jest wywołana
- **PostConstruct** - wywołana po wsztrzyknięciu zależności
- **PreDestroy** - wywołana do sygnalizacji, że instancja jest usuwana

IMPLEMENTACJA INTERCEPTORA W EJB

IMPLEMENTACJA INTERCEPTORA W EJB

Załóżmy, że istnieje klasa ***AcademicFascadeImpl***, która zawiera metodę przechwytywaną ***requestTestReview*** *wysyłającą żądanie*. Chcąc wiedzieć który użytkownik wykonał najwięcej żądań można wykorzystać interceptor , aby oddzielić logikę biznesową od kodu

```
@Stateless
@LocalBean
public class AcademicFacadeImpl {
    ...
    ...
    // this method will be intercepted for some statistical
    // interceptor:
    public void requestTestReview (@Observes TestRevisionTO
   testRevisionTO) {
       System.out.println("enrollment : " +
       testRevisionTO.getEnrollment());
       LocalDateTime dateTime = scheduleTestReview (testRevisionTO);
       // send an email with the schedule date for review:
       sendEmail (testRevisionTO, dateTime);
   }
}
```

```
@Stateless
public class AcademicFacadeImpl {
    ...
    ...
    @AroundInvoke
  public Object statisticMethod (InvocationContext invocationContext)
throws Exception{
    ...
}

}
```

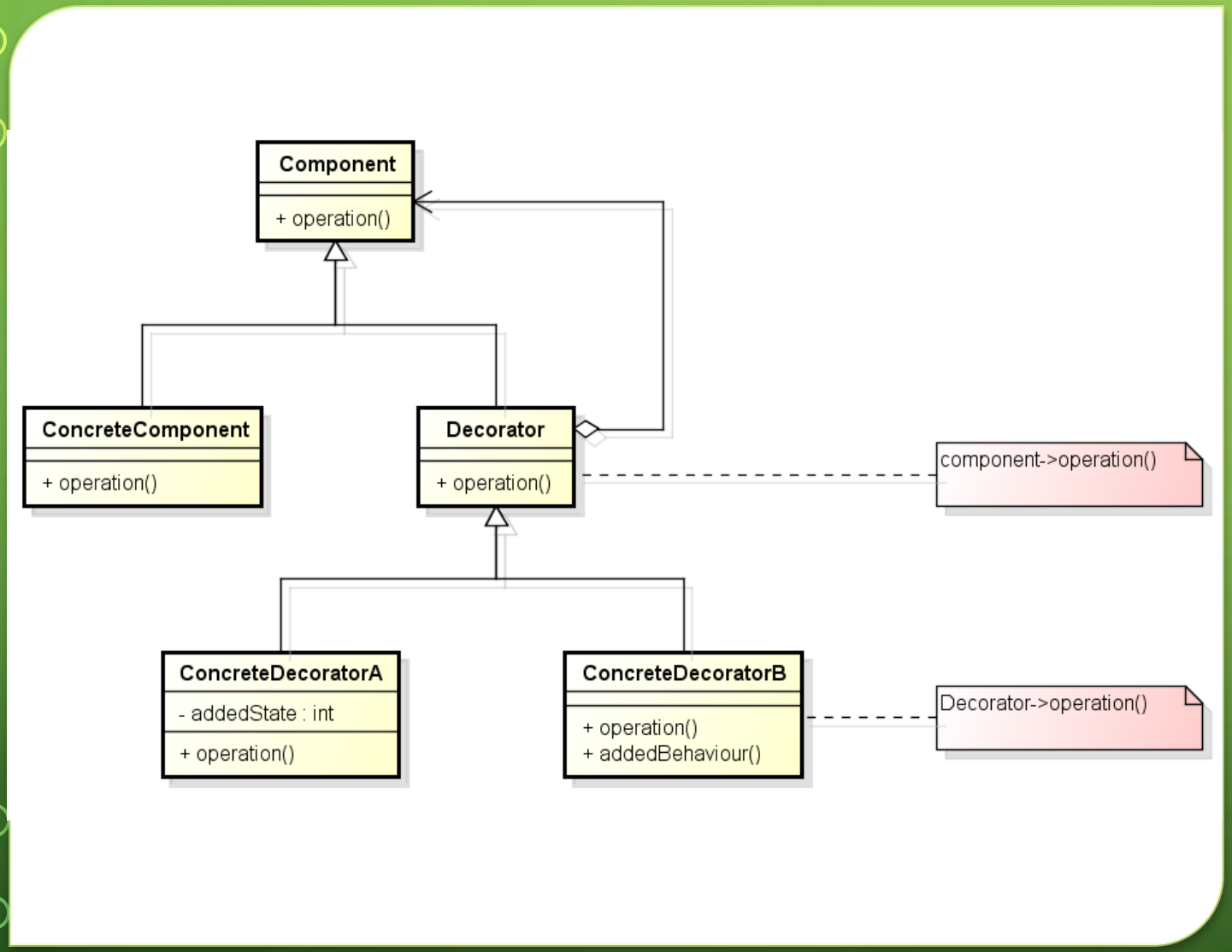
RP

DEKORATOR

CZYM JEST DEKORATOR?

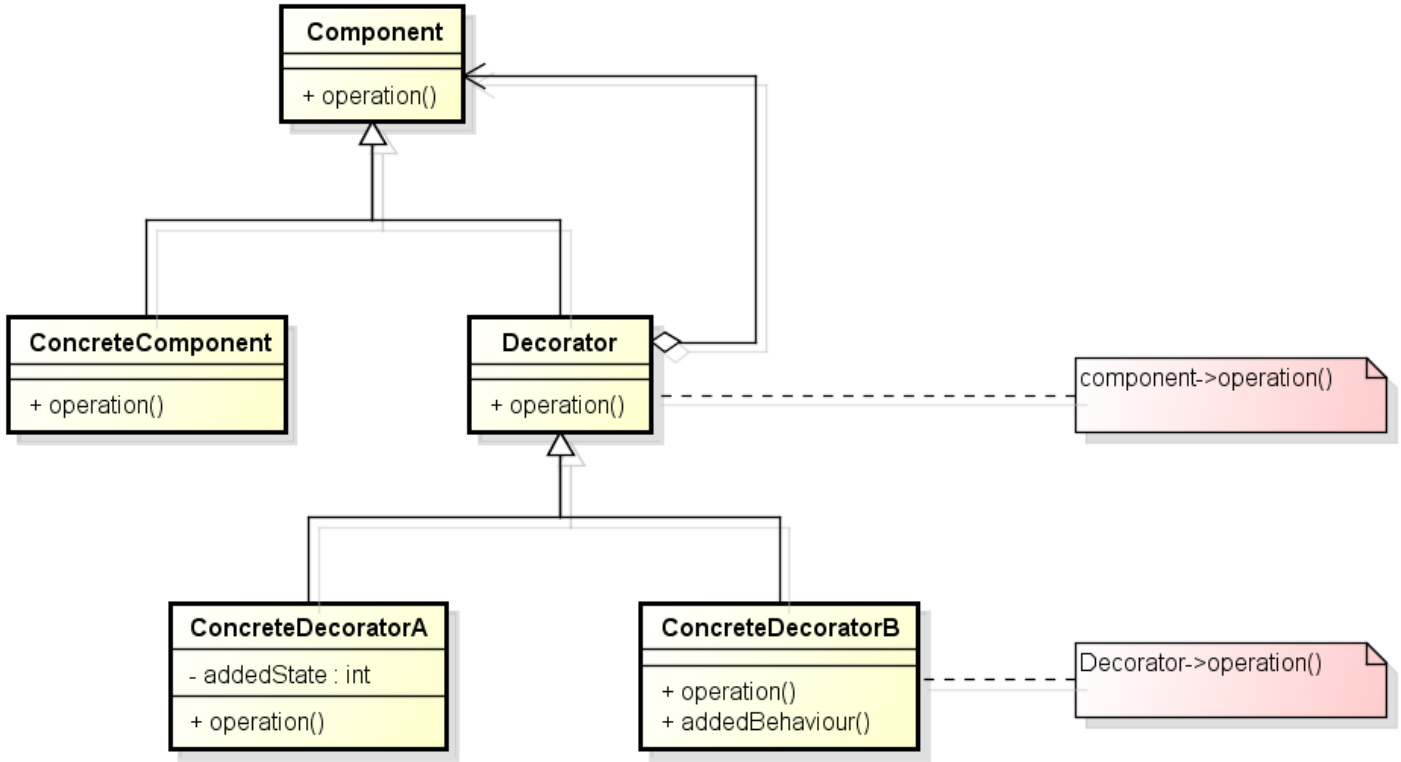
- Dekorator to wzorzec projektowy, w którym logika biznesowa aplikacji jest rozszerzana o poszczególne funkcjonalności dopiero w czasie działania programu.
- Oryginalny obiekt jest „opakowywany” w nowy. Metody które istnieją w oryginalnym obiekcie działają bez zmian, jednak dostępne są również nowe, dodatkowe metody.

- W uproszczeniu: dekorator dodaje nowe funkcjonalności do istniejących obiektów, bez potrzeby używania dziedziczenia.
- Zachowanie obiektu jest zmieniane podczas działania programu, a nie podczas kompilacji, jak w przypadku dziedziczenia.



WZORZEC DEKORATORA

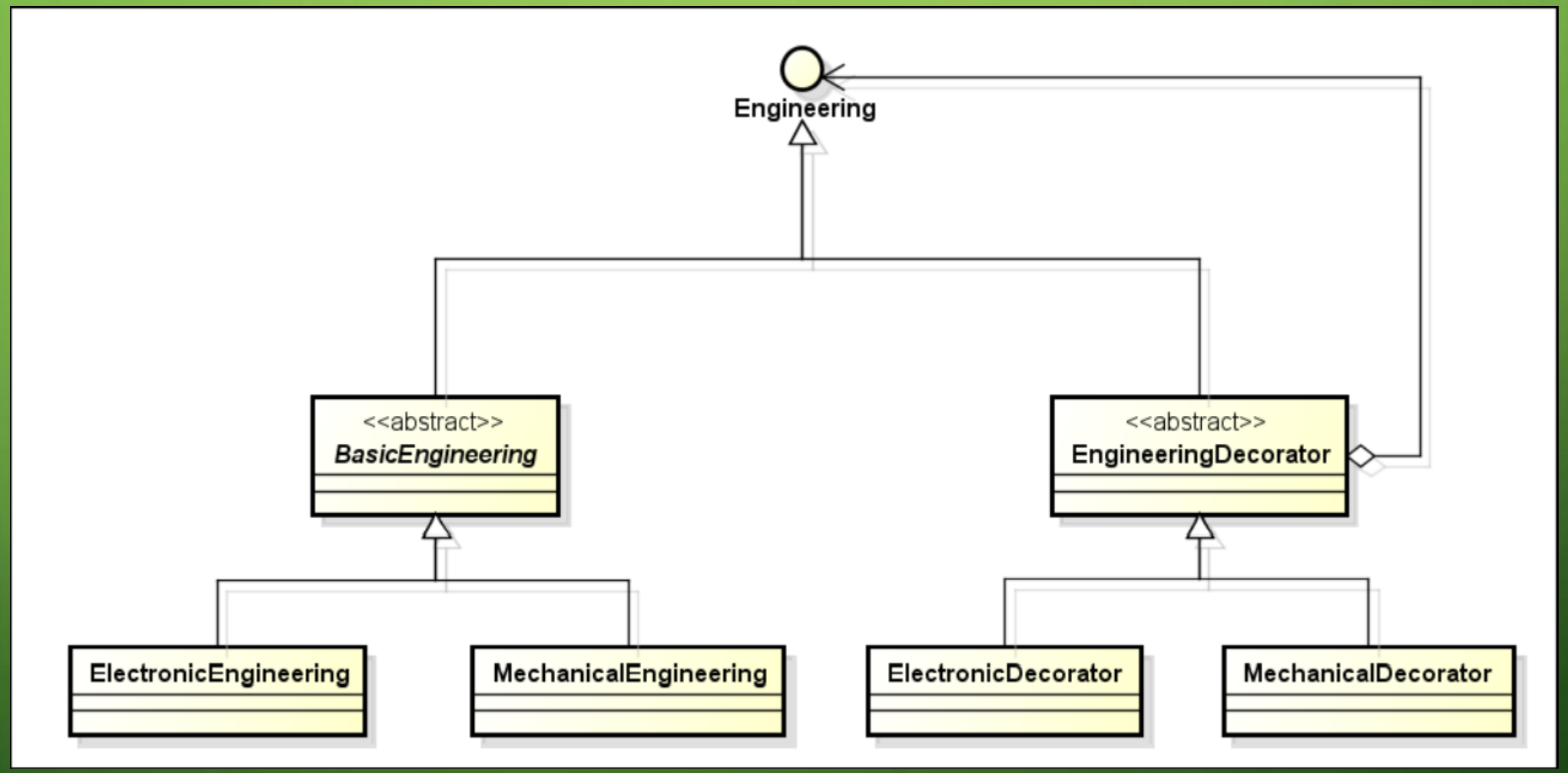
- Component – interfejs
- ConcreteComponent – klasa korzystająca z interfejsu Component. Mogą być do niej dynamicznie dodawane nowe funkcjonalności.



WZORZEC DEKORATORA

- Decorator – interfejs przechowujący referencję do obiektu z interfejsem Component. Musi zapewniać kompatybilność z Component.
- ConcreteDecorator… – Klasy dodające nowe funkcjonalności – zarówno pola jak i metody.

DIAGRAM PRZYKŁADOWEJ IMPLEMENTACJI



IMPLEMENTACJA INTERFEJSU I KLAS

```
public interface Engineering {
 List<String> getDisciplines ();
}
public class BasicEngineering implements Engineering {

 @Override
 public List<String> getDisciplines() {
 return Arrays.asList("d7", "d3");
 }
}
@Electronic
public class ElectronicEngineering extends BasicEngineering {
 ...
}
@Mechanical
public class MechanicalEngineering extends BasicEngineering {
 ...
}
```

IMPLEMENTACJA DEKORATORA

```
@Decorator
public abstract class EngineeringDecorator implements Engineering {
    @Electronic
    @Any
    @Inject
    @Delegate
    Engineering engineering;

    @Override
    public List<String> getDisciplines() {
        System.out.println("Decorating Electronic");
        List<String> disciplines = new ArrayList<>
        (engineering.getDisciplines());
        disciplines.addAll (Arrays.asList("d21", "d27"
        "d22"));
        return disciplines;
     }
}
```

```
@Decorator
public abstract class MechanicalDecorator implements Engineering {
    @Mechanical
    @Any
    @Inject
    @Delegate
    Engineering engineering;

    @Override
    public List<String> getDisciplines() {
        System.out.println("Decorating Mechanical Engineering");
        List<String> disciplines = new ArrayList<>
        (engineering.getDisciplines());
        disciplines.addAll (Arrays.asList("d31", "d37", "d33", "d34",
        "d32"));
        return disciplines;
     }
}
```

PYTANIA?

DZIĘKUJEMY ZA UWAGĘ 😄

Marzena Pepera

Rafał Pokrywka

Karolina Banyś

Konrad Czechowicz

Bartłomiej Czupta

Paweł Mąsior

Paweł Niziałek