Business Patterns

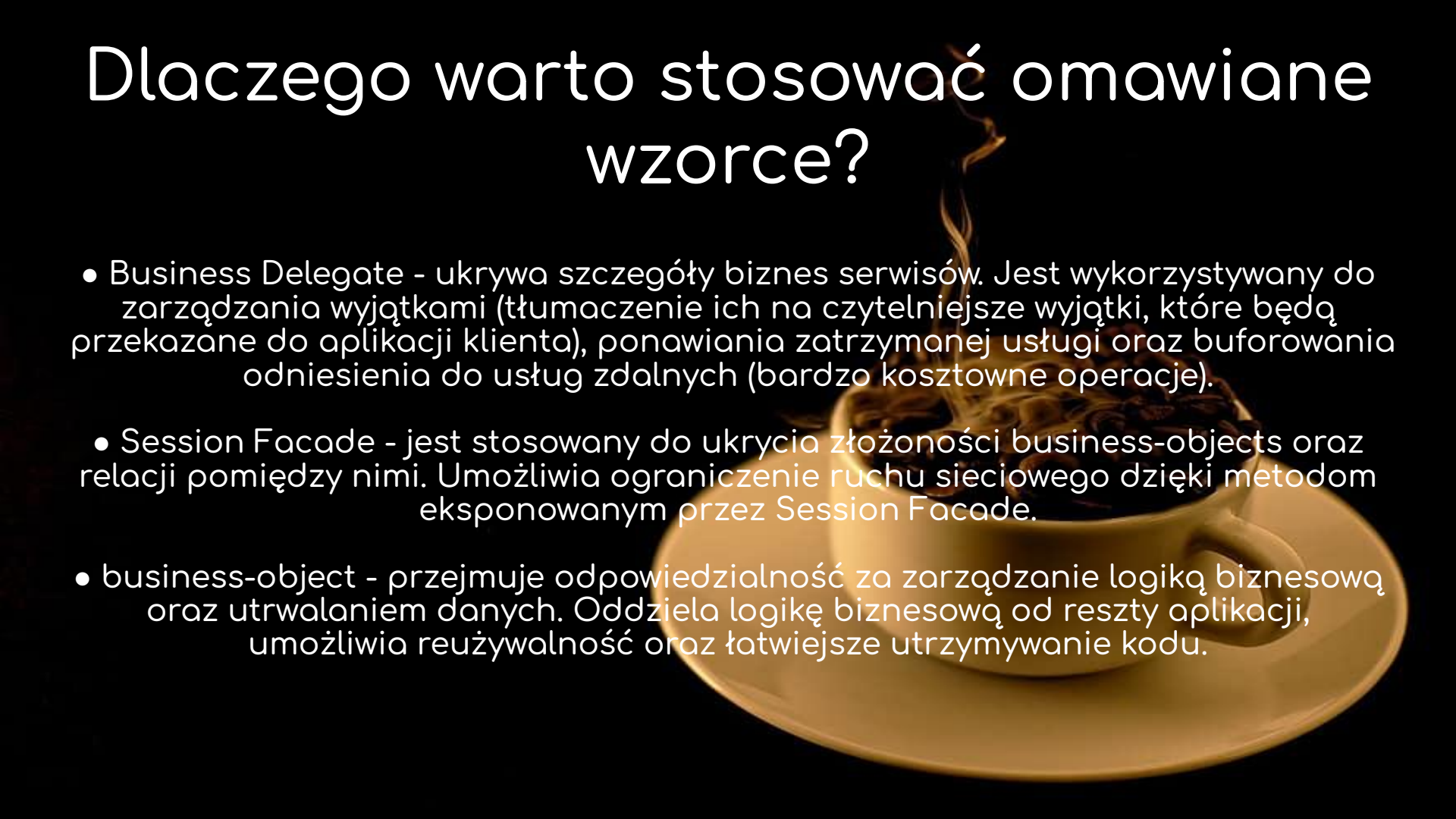
Wzorce warstwy biznesowej

Prezentacja obejmuje wzorce projektowe warstwy biznesowej wraz z ich zastosowaniem w aplikacjach stworzonych z wykorzystaniem Javy EE.

Tematy zawarte w prezentacji:

- Omówienie warstwy biznesowej
- Wzorzec Business Delegate
- Wzorzec Session Facade
- Wzorzec business-object

Dlaczego warto stosować omawiane wzorce?



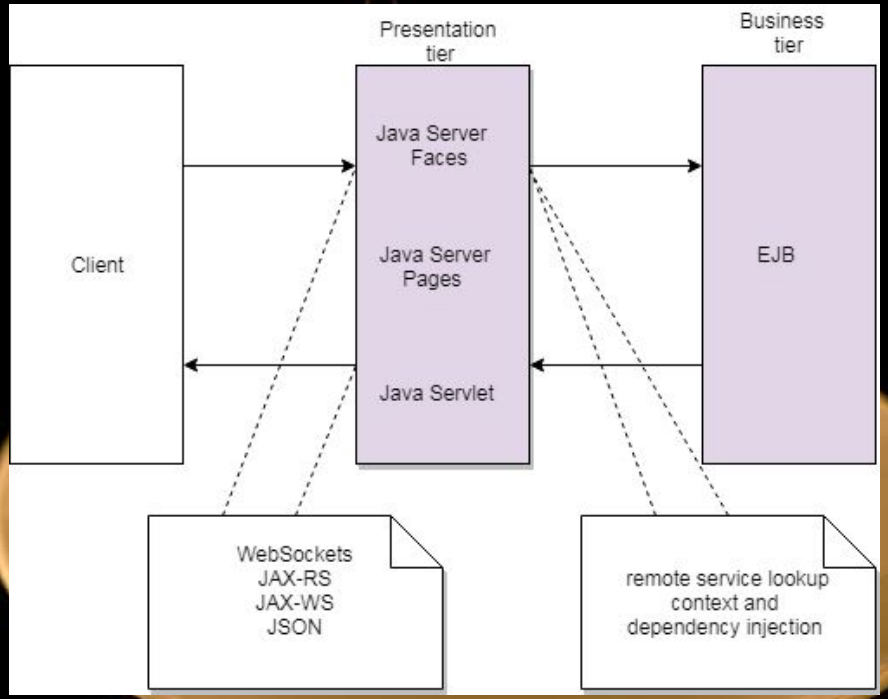
- Business Delegate - ukrywa szczegóły biznes serwisów. Jest wykorzystywany do zarządzania wyjątkami (tłumaczenie ich na czytelniejsze wyjątki, które będą przekazane do aplikacji klienta), ponawiania zatrzymanej usługi oraz buforowania odniesienia do usług zdalnych (bardzo kosztowne operacje).

- Session Facade - jest stosowany do ukrycia złożoności business-objects oraz relacji pomiędzy nimi. Umożliwia ograniczenie ruchu sieciowego dzięki metodom eksponowanym przez Session Facade.

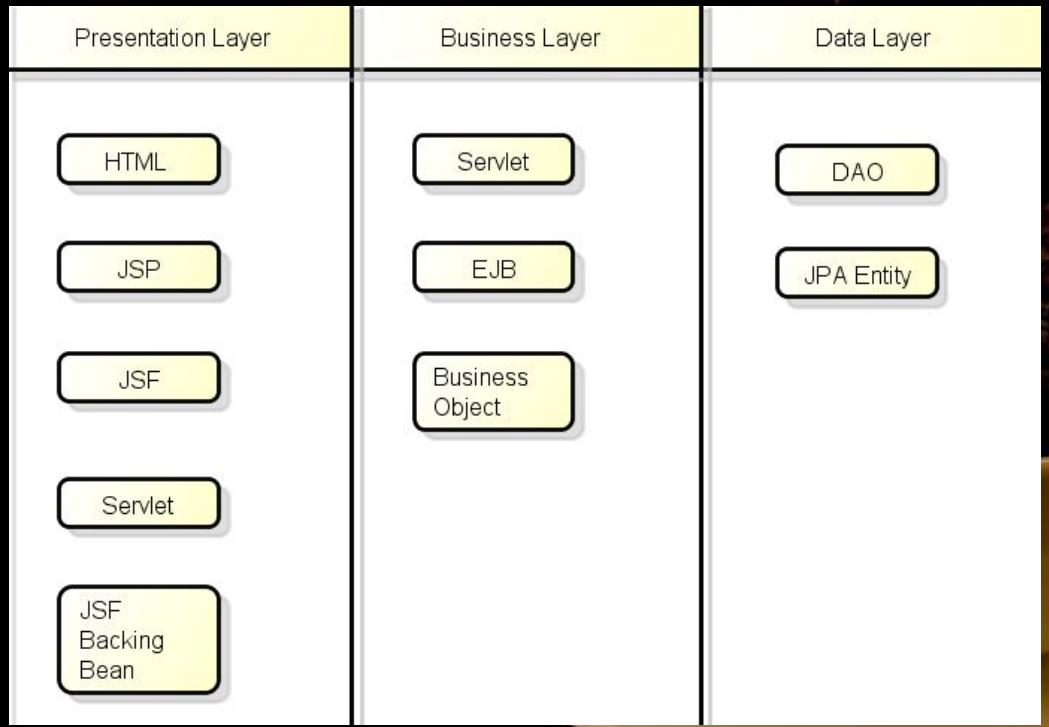
- business-object - przejmuje odpowiedzialność za zarządzanie logiką biznesową oraz utrwalaniem danych. Oddziela logikę biznesową od reszty aplikacji, umożliwia reużywalność oraz łatwiejsze utrzymywanie kodu.

Wybrane patterny

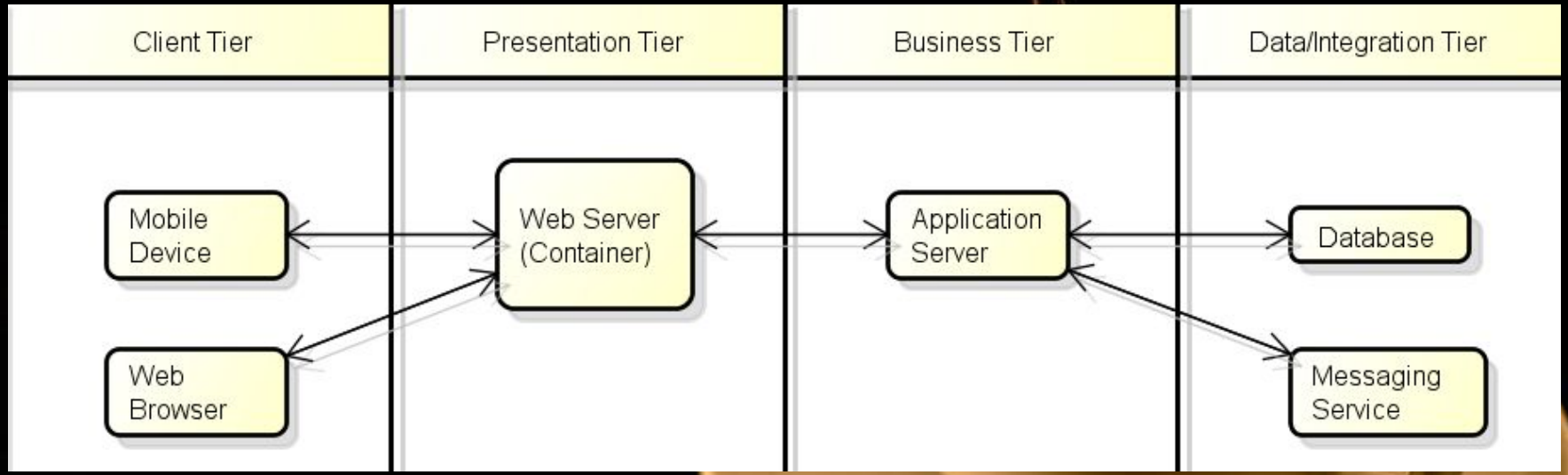
- Business Delegate pattern
- Session Façade pattern
- Business-object pattern



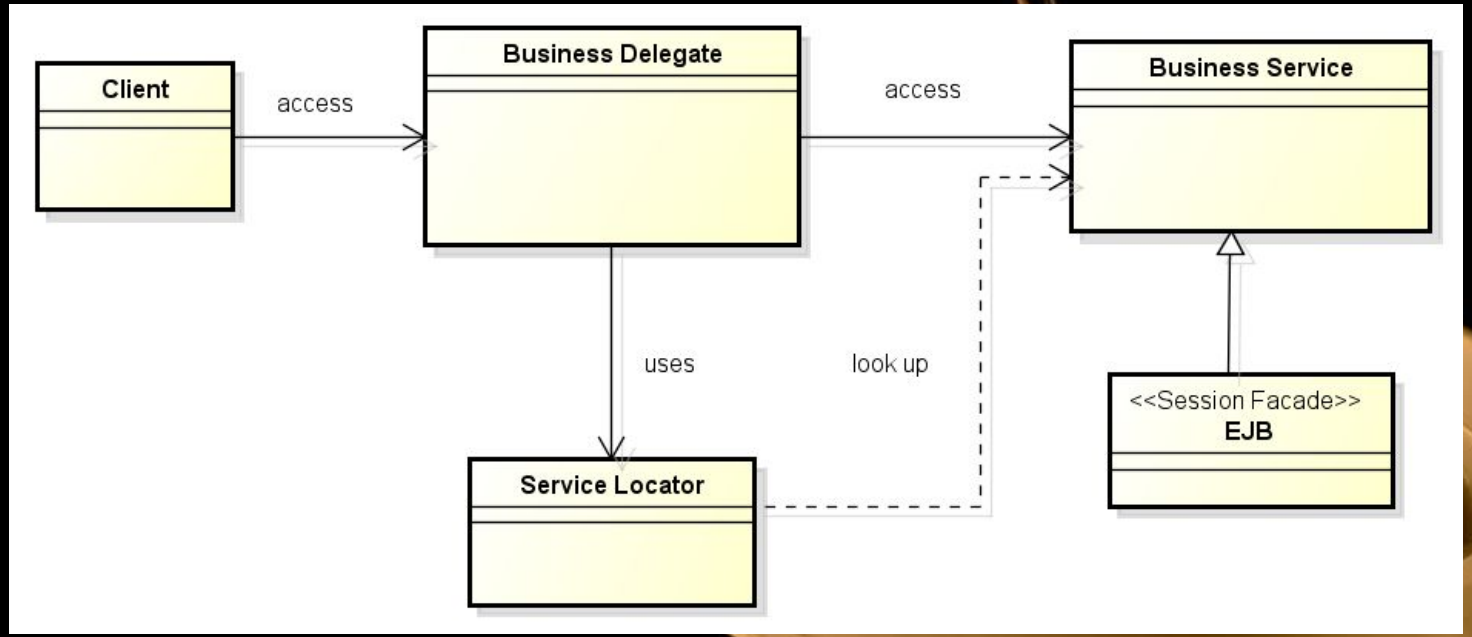
Warstwy


Presentation Layer
Business Layer
Data Layer
HTML
JSP
JSF
Servlet
JSF Backing Bean
Servlet
EJB
Business Object
DAO
JPA Entity

Tiers


Client Tier
Presentation Tier
Business Tier
Data/Integration Tier
Mobile Device
Web Browser
Web Server (Container)
Application Server
Database
Messaging Service

Business Delegate pattern scenario



Korzyści wynikające z zastosowania Session Façade pattern:

- centralizuje logikę biznesową, ograniczając eksponowanie złożonych interakcji
- angażuje obiekty biznesowe po stronie klienta
- hermetyzuje komponenty warstwy biznesowej
- klienci uzyskują dostęp do Session Façade zamiast do komponentów biznesowych

Implementacja Session Façade pattern in JEE

Session Façade pattern implementowany jest przez bezstanowy lub stanowy EJB. EJB może używać lub łączyć w sobie obiekty biznesowe, EJB lub POJO. Podczas implementacji należy uważać, aby stworzyć zbyt wielu niepotrzebnych warstw w celu uniknięcia łańcucha wywołań niepotrzebnej ilości coraz bardziej wewnętrznych obiektów EJB. Usługi muszą być odpowiednio zaprojektowane i zmapowane.

Ponieważ Session Façade pattern jest zaimplementowany przez EBJ w usługach takich jak transaction-control, security-management to w tej warstwie zwykle mamy kontrolę nad transakcjami większości obiektów wewnętrznych.

The classic Session Façade pattern scenario

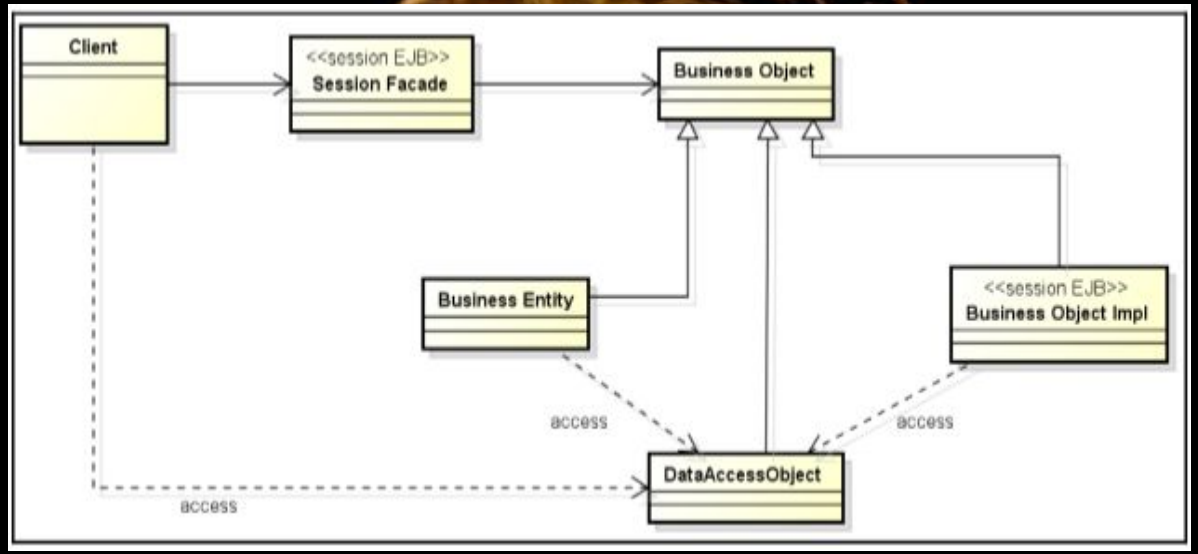
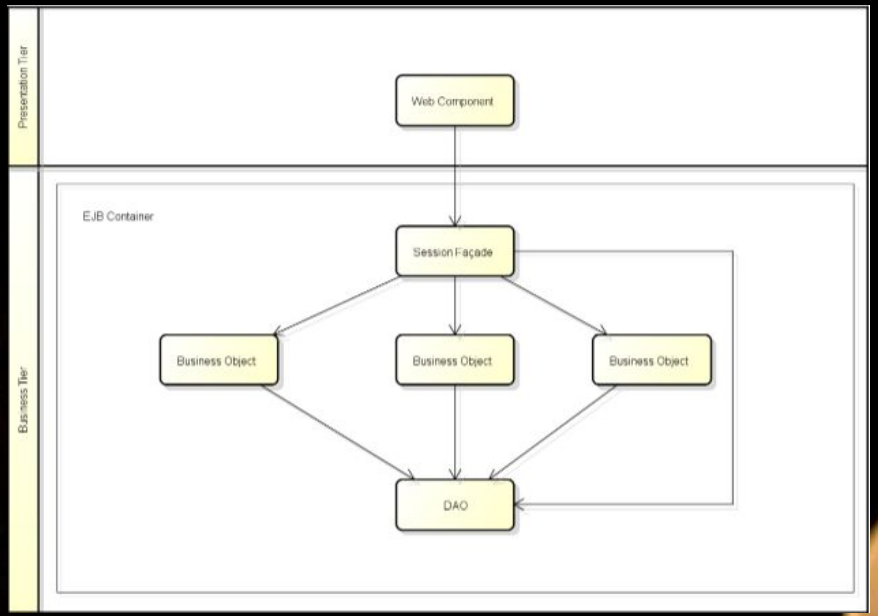
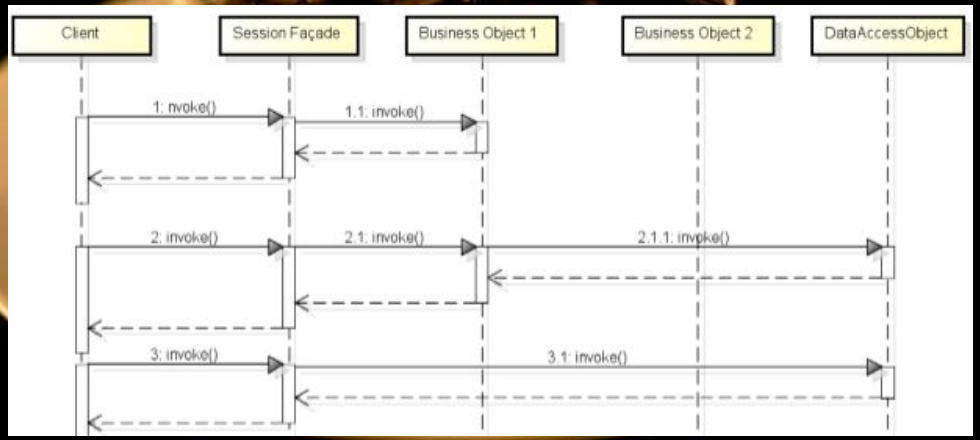


Diagram aktywności z warstwami komponentów zagnieżdżonymi w Session Façade pattern



Schemat sekwencyjny Session Façade pattern



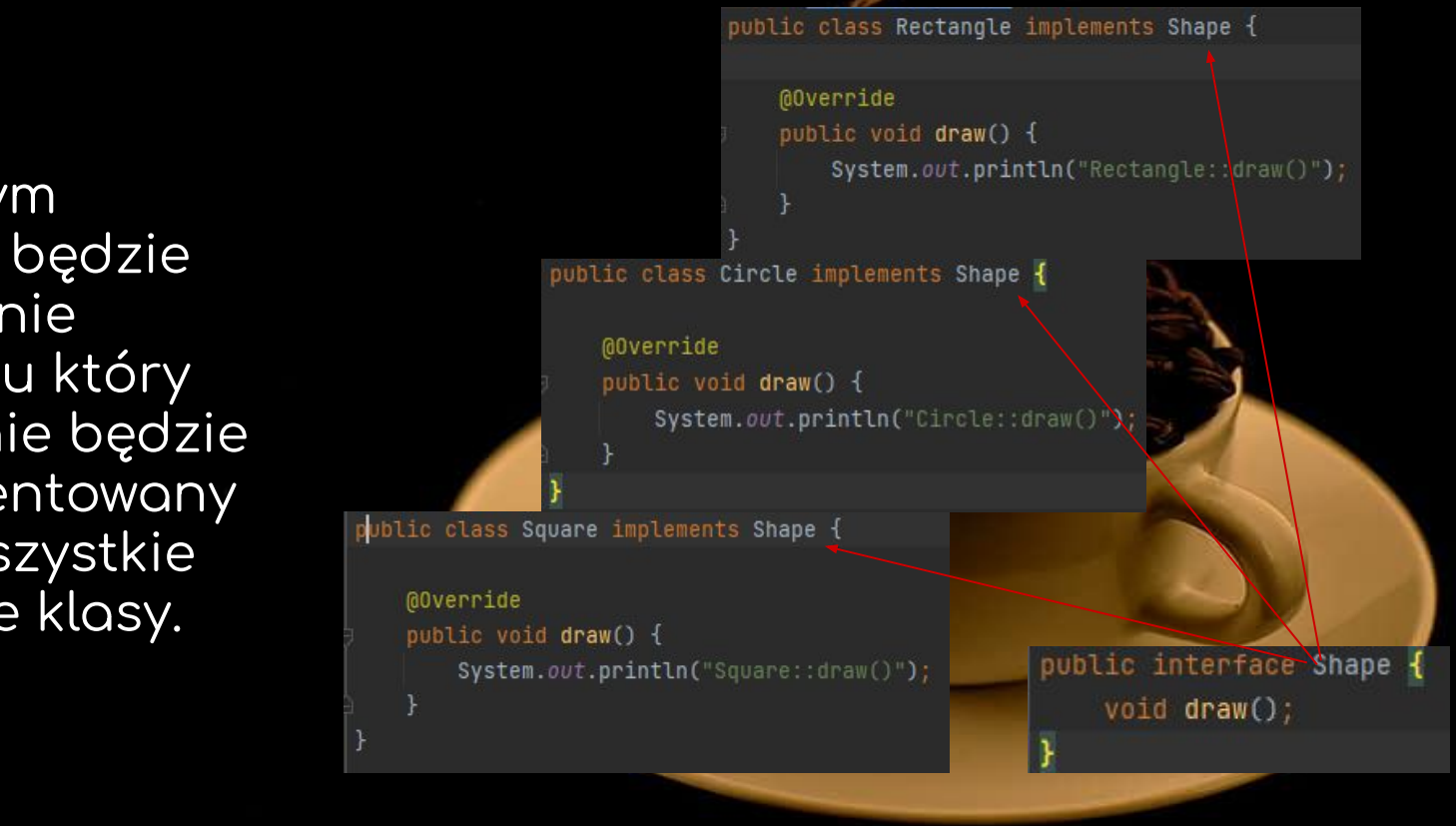
Implementacja Façade Design Pattern

Implementacja wyżej wymienionego wzorca składa się z:

- utworzenia interfejsu
- utworzenie concrete klas implementującej utworzony wcześniej interfejs
- utworzenie klasy fasady
- użycie fasady
- weryfikacja outputu

Utworzenie interfejsu oraz concrete klas

Pierwszym krokiem będzie utworzenie interfejsu który następnie będzie implementowany przez wszystkie concrete klasy.



Utworzenie klasy fasady oraz użycie jej

```java
public class ShapeMaker {
    private final Shape circle;
    private final Shape rectangle;
    private final Shape square;

    public ShapeMaker() {
        circle = new Circle();
        rectangle = new Rectangle();
        square = new Square();
    }

    public void drawCircle(){
        circle.draw();
    }
    public void drawRectangle(){
        rectangle.draw();
    }
    public void drawSquare(){
        square.draw();
    }
}
```

Za pomocą klasy ‘ShapeMaker’ kierujemy użytkownika do concrete klasy która została zawołana

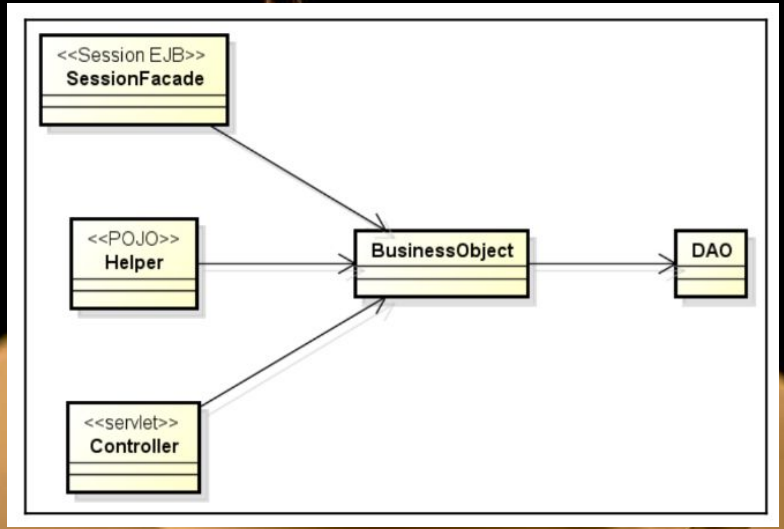
```java
public class FacadePatternDemo {
    public static void main(String[] args) {
        ShapeMaker shapeMaker = new ShapeMaker();

        shapeMaker.drawCircle();
        shapeMaker.drawRectangle();
        shapeMaker.drawSquare();
    }
}
```

Czym jest business-object pattern?

- służy do reprezentowania obiektów typu konta bankowe, studenci, pracownicy, zamówienia, itp.
- przydatny w aplikacjach z bardziej złożoną logiką biznesową
- przechowuje dane o danym obiekcie, jak i o obiektach powiązanych relacjami

Business-object pattern - wykorzystanie

Business Object jest “pośrednikiem”, który oddziela warstwę danych od warstwy logiki biznesowej.



Business-object pattern - zalety

- wykorzystanie tego wzorca prowadzi do większej reużywalności kodu
- dostęp do BO z wielu miejsc gwarantuje jednolitość zachowania
- oddziela logikę biznesową od pozostałych elementów aplikacji
- zwiększa spójność kodu (separation of responsibilities - rozdział obowiązków)

Implementacja wzorca business-object

Teraz wprowadzimy trochę kodu aby zilustrować wzorzec business-object. Musimy jednak zwrócić uwagę na to, że prawdopodobnie istnieje inne podejście do uzyskania wyników. Przykładowo możemy użyć mapowania O-R (JPA lub Hibernate) do mapowania encji. Na przykład, encja Profesor ma relację n do n
z encją Discipline co odbywa się za pomocą adnotacji JPA.

My użyjemy ProfessorBO, Professor, Discipline, ProfessorDAO oraz DisciplineDAO. Skorzystamy również z klas pokazanych poniżej. Została wprowadzona zmiana w klasie AcadmicFacedelmpl. Teraz, ta fasada sesji używa BO o nazwie ProfessorBO do obsługi biznesu związanego z Professorem.

Przyjrzyjmy się klasie PrefossorBO:

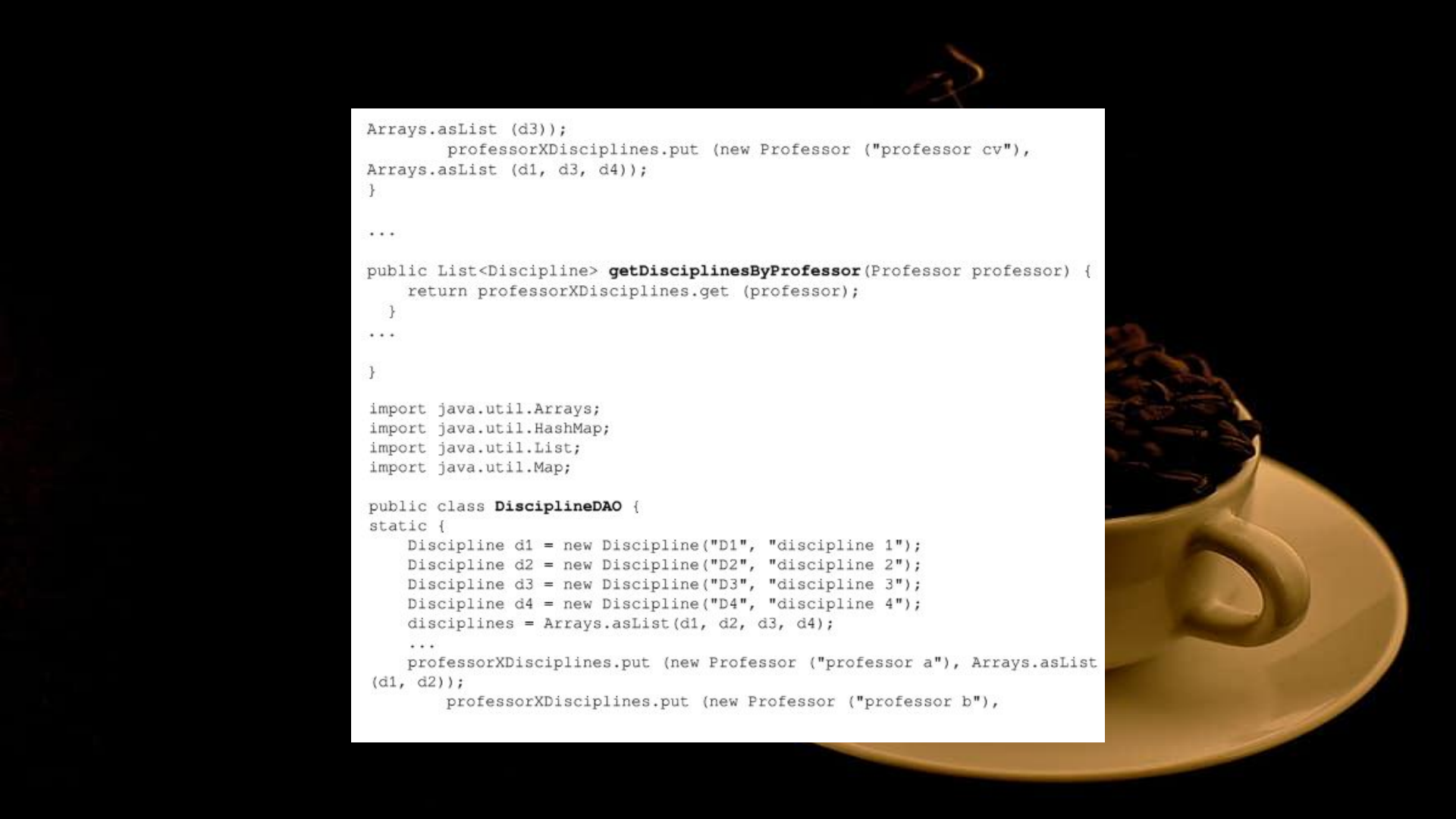
```java
import java.time.LocalDate;
import java.util.List;
import javax.inject.Inject;

public class ProfessorBO {
    private Professor professor;
    private List<Discipline> disciplines;
    @Inject
    private ProfessorDAO professorDAO;
    @Inject
    private DisciplineDAO disciplineDAO;

    public void setProfessor(Professor professor) {
        this.professor = professorDAO.findByName(professor.getName());
    }

    public boolean canTeachDiscipline(Discipline discipline) {
        if (disciplines == null) {
            disciplines = disciplineDAO.getDisciplinesByProfessor(professor);
        }
        return disciplines.stream().anyMatch(d -> d.equals(discipline));
//return disciplines.contains(discipline);
    }

    public LocalDate getInitDate() {
        return professor.getInitDate();
    }

    public String getName() {
        return professor.getName();
    }
}
```

Przyjrzyjmy się również klasie AcademicFacadeImpl:

```
@Stateless
@LocalBean
public class AcademicFacadeImpl implements AcademicFacadeRemote,
        AcademicFacadeLocal {
...
        ...
    @Inject
    private ProfessorBO professorBO;
    @Override
    public List<Professor> getProfessorsByDiscipline(Discipline discipline) {
        return disciplineDAO.getProfessorByDiscipline(discipline);
    }
    public boolean canProfessorTeachDiscipline (Professor professor,
                                                Discipline discipline) {
/*return disciplineDAO.getDisciplinesByProfessor
(professor).contains(discipline);*/
        professorBO.setProfessor (professor);
        return professorBO.canTeachDiscipline(discipline);
    }
}
```

Jak widać w poprzednim bloku kodu, fasada AcademicFacadeImpl Session Façade wywołuje metodę canTeachDiscipline z wstrzykniętego beana ProfessorBO. ProfessorBO korzysta następnie z ProfessorDAO i DisciplineDAO.

Teraz przyjrzyjmy się części kodu DisciplineDAO, która używana jest przez ProfessorBO:



```
Arrays.asList (d3));
        professorXDisciplines.put (new Professor ("professor cv"),
Arrays.asList (d1, d3, d4));
}

...

public List<Discipline> getDisciplinesByProfessor(Professor professor) {
    return professorXDisciplines.get (professor);
  }
...

}

import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class DisciplineDAO {
static {
    Discipline d1 = new Discipline("D1", "discipline 1");
    Discipline d2 = new Discipline("D2", "discipline 2");
    Discipline d3 = new Discipline("D3", "discipline 3");
    Discipline d4 = new Discipline("D4", "discipline 4");
    disciplines = Arrays.asList(d1, d2, d3, d4);
    ...
    professorXDisciplines.put (new Professor ("professor a"), Arrays.asList
(d1, d2));
        professorXDisciplines.put (new Professor ("professor b"),
```

Oraz dość ważny fragment używany przez ProfessorDTO:

```
public class ProfessorDAO {
  private static Set<Professor> professors;
  static {
   Professor p1 = new Professor ("professor a", LocalDate.of (2001, 03,
   22)),
      p2 = new Professor ("professor b", LocalDate.of (1994, 07, 05)),
      p3 = new Professor ("professor c", LocalDate.of (1985, 10, 12)),
      p4 = new Professor ("professor cv", LocalDate.of (2005, 07, 17));
    professors = Arrays
          .stream (new Professor[]{p1, p2, p3, p4})
            .collect (Collectors.toSet());
  }
  public Professor findByName (String name) {
    return professors
        .stream()
        .filter(p->p.getName().equals(name))
        .findAny()
        .get();
  }

}
```

Finalny kod dla Professor BO:

```
import java.time.LocalDate;
import java.util.List;
import javax.inject.Inject;

public class ProfessorBO {
  private Professor professor;
  private List<Discipline> disciplines;
  @Inject

   private ProfessorDAO professorDAO;
   @Inject
   private DisciplineDAO disciplineDAO;
   public void setProfessor (Professor professor ) {
     this.professor = professorDAO.findByName (professor.getName());
   }

   public boolean canTeachDiscipline (Discipline discipline) {
     if (disciplines == null) {
       disciplines = disciplineDAO.getDisciplinesByProfessor
       (professor);
     }
     return disciplines.stream().anyMatch(d->d.equals(discipline));
     //return disciplines.contains(discipline);
   }
   public LocalDate getInitDate () {
     return professor.getInitDate();
   }
   public String getName () {
     return professor.getName();
   }
 }
```

Podsumowanie

Głównym celem Delegate Business jest ukrycie szczegółów implementacji usług pod warstwą prezentacji. CDI - to technologia odpowiedzialna za wstrzykiwanie komponentów do aplikacji w bezpieczny sposób i istnieją sytuacje w których ta technologia była użyteczna.

Business Delegate jest nadal szeroko stosowany w bardziej technicznych wyjątkach - na przykład, gdy ma zdalne wywołania EJB. Ponadto delegat ochrania opisywaną wyżej warstwę prezentacji przed ewentualnymi niepożądanymi zmianami w warstwie usługi i odwrotnie w momencie gdy klienci są inne niż przeglądarka, przez co mamy łatwiejszy dostęp do usług.

Dziękujemy za uwagę

Krzysztof Szczęśniak
Andrzej woźniacki
Aleksander Radwan-Pragłowski
Mateusz Klimczyk
Mateusz Nawrocki
Paweł Pytlowski