

Enterprise Software Architecture - FAQ Ausarbeitung

Phillip Wo
Benjamin Moser
Daniel Sommer

March 2, 2019

Contents

I	Pattern im Projekt	4
1	Layers Pattern	4
1.1	Erkläre die Funktionsweise + Skizze	4
1.1.1	3 Schichten Architektur:	4
1.2	Beschreibe ein konkretes Anwendungsbeispiel	5
2	Data Access Object (DAO) Pattern	5
2.1	Erkläre die Funktion + Skizze	5
2.2	Nenne die Konsequenzen der Anwendung	6
3	Service Layer Pattern (auch Session Fassade)	6
3.1	Erkläre die Funktion + Skizze	6
3.2	Nenne die Konsequenzen der Anwendung	8
4	Model-View-Controller (MVC) Pattern	9
4.1	Erkläre die Funktion + Skizze	9
4.1.1	Model	9
4.1.2	View	10
4.1.3	Controller	10
4.2	Nenne die Konsequenzen der Anwendung	10
5	Front Controller	10
5.1	Erkläre die Funktion + Skizze	10
5.2	Beschreibe ein konkretes Anwendungsbeispiel	11
5.3	Anwendung im Projekt	11
5.4	Nenne die Konsequenzen der Anwendung	12
6	View Helper	12
6.1	Erkläre die Funktion + Skizze	12
6.2	Nenne die Konsequenzen der Anwendung	13
7	Dependency Injection (don't call us, we'll call you)	13
7.1	Erkläre die Funktion + Skizze	13
7.2	Nenne die Konsequenzen der Anwendung	14
8	Data Transfer Object (DTO) Pattern	14
8.1	Erkläre die Funktion (Skizze - ein Grund für DTO)	14
8.2	Beschreibe ein konkretes Anwendungsbeispiel	15
8.3	Nenne die Konsequenzen der Anwendung	15
9	Page-Object-Pattern	15
10	Remote	15
11	Beschreibe die Unterschiede zwischen lokalem und Remote Interface Design	15

12 Beschreibe drei Situationen wo Multiple Prozesse in Applikationen verwendet werden müssen	16
13 Beschreibe das folgende Diagramm. Was können wir daraus für das Design von Remote Interfaces folgern?	16
14 Exception Handling	16
15 Beschreibe den Unterschied zwischen Checked und Runtime Exceptions in Java (inkl. Klassendiagramm)	16
16 Beschreibe einen Use Case für eine Checked Exceptions in Java	17
17 Beschreibe einen Use Case für eine Runtime Exceptions in Java	17
18 Beschreibe 5 Best Practice Beispiele beim Einsatz von Exceptions	17
19 Beschreibe 5 Exception Handling Anti Pattern	17
20 Logging	18
21 Nenne die Nachteile von debugging mit printf() sowie die Vorteile die Logging Frameworks wie log4j bieten	18
21.1 Nachteile printf()	18
21.2 Vorteile Logging mittels Framework (z.B.: log4j)	18
22 Project Structure	18
23 Toni	18
24 Annotationen	18
24.1 @MappedSuperclass	18
25 Patterns in Practice	20
26 Konfigurationsdateien (pom.xml), (persistence.xml) und noch a bissl mehr Scheiß	20
27 Reihenfolge - Wildfly - Abfolge - einzelne Schritte	20
28 Frageart Prüfung	20
29 Die CONFIG-Files	21
29.1 web.xml	21

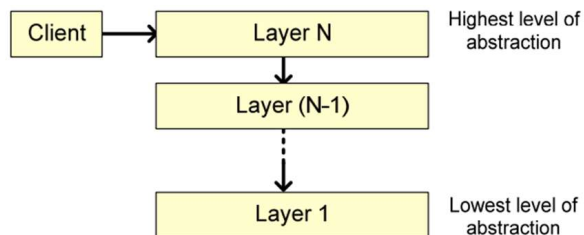
Part I

Pattern im Projekt

1 Layers Pattern

1.1 Erkläre die Funktionsweise + Skizze

- Client schickt eine Anfrage an Layer N
- Layer N reicht da er nicht vollständig alleine beantworten kann, Anfragen an darunterliegenden Layer weiter
- Eine Anfrage kann bei Bedarf auch in mehrere Anfragen an darunterliegende Layer geteilt werden
- dies wird immer weiter fortgesetzt bis Layer 1 erreicht ist
- dabei gehen Abhängigkeiten nur von oben nach unten

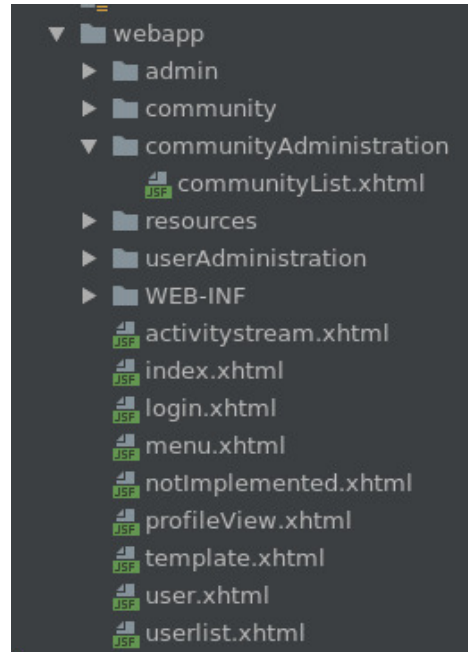
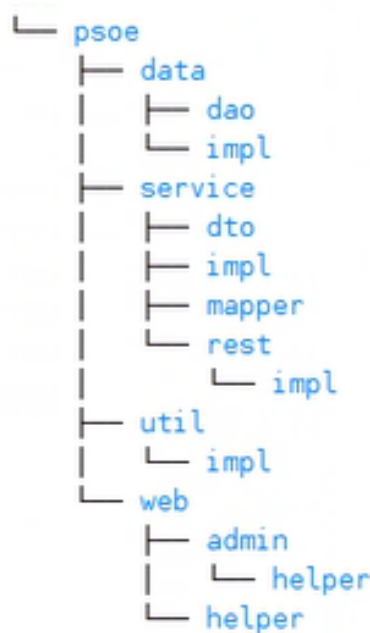


1.1.1 3 Schichten Architektur:

- Data Source Layer (data): Zugriff auf Daten, kümmert sich um Kommunikation mit anderen Systemen (z.B.: Datenbank)
 - beinhaltet DAO und DAOImpl » DocumentDAO, DocumentlibraryDAO
- Domain Layer(service): enthält Business Logik (Berechnungen, Datenvalidierung, ...)
 - beinhaltet
 - * **Service Layer Pattern** (aka Session Fassade - siehe 3)
 - * DTO » DocumentDTO
 - * Mapper » DocumentMapper

```
public static Document toEntity(DocumentDTO documentDTO, Document
↔ document){};
public static DocumentDTO toDTO(Document document){};
```

- Presentation Layer(web): serverseitig, kümmert sich um Benutzerinteraktion
 - Controller (ViewHelper) » DocumentController, DocumentListController
 - View (WebApp)



1.2 Beschreibe ein konkretes Anwendungsbeispiel

- Java Enterprise Edition (JEE) Architektur bestehend aus
 - Client Tier (Client Machine)
 - Web Tier, Business Tier (Java EE Server)
 - EIS Tier (Database Server)

2 Data Access Object (DAO) Pattern

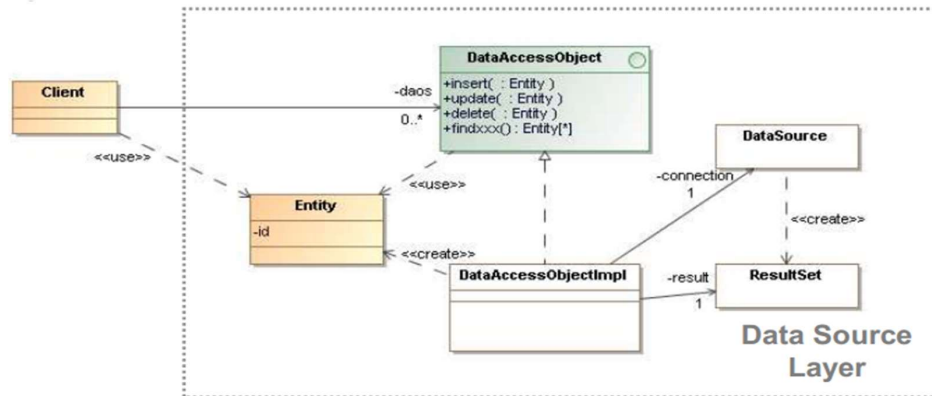
Befindet sich im Projekt in data und damit innerhalb des Data Layer.

2.1 Erkläre die Funktion + Skizze

- Client erstellt ein DAO Object und kann nach Entitäten suchen, einfügen, löschen, etc.
- das DAO selbst soll keine spezifischen Elemente enthalten (Entity Manager, SQL Exception -> stattdessen DAOException)
- dadurch entsteht eine Kapselung bei der die DAOImpl ohne den Client zu verändern ausgetauscht werden kann

```

@ApplicationScoped
public class DocumentDAOImpl implements DocumentDAO, Serializable {
    private static final long serialVersionUID = 1L;
    private static final Logger logger =
        ↪ LoggerFactory.getLogger(DocumentDAOImpl.class);
    @PersistenceContext
  
```



```

private EntityManager entityManager;

@Override
public List<Document> findByCommunity(Community community) {...}

@Override
public List<Document> findByUser(User user) {...}

@Override
public void insert(Document document) {...}

@Override
public void delete(Document document) {...}

@Override
public Document findById(Long id) {...}
}

```

2.2 Nenne die Konsequenzen der Anwendung

- Zugriff auf persistenten Speicher wird abstrahiert
- Details des Speichers werden versteckt
- ermöglicht einheitlichen Zugriff auf Daten
- entkoppelt Implementierung von Persistierung (Datenbank,...)
- ermöglicht Objektorientierte Ansicht des Speichers

3 Service Layer Pattern (auch Session Fassade)

3.1 Erkläre die Funktion + Skizze

- Der Service Layer delegiert auf die Business Logik (Zeile 68 community.setDocumentlibrary) und zum DAO (z.B. Zeile 66)
- Bei wenig Logik wird zumindest Transaktions (Zeile 40), Error (ab Zeile 42) und Validierungshandling (ab Zeile 23) im Service erledigt

```

1  @Local(DocumentService.class)
2  @Remote(DocumentServiceRemote.class)
3  @Stateless
4  public class DocumentServiceImpl implements DocumentService,
   ↳ DocumentServiceRemote, Serializable {
5      private static final long serialVersionUID = -1L;
6      private static final Logger logger =
   ↳ LoggerFactory.getLogger(DocumentServiceImpl.class);
7
8      @Inject
9      private DocumentDAO documentDAO;
10     @Inject
11     private DocumentlibraryDAO documentlibraryDAO;
12     @Inject
13     private CommunityDAO communityDAO;
14     @Inject
15     private UserDAO userDAO;
16     @Inject
17     private MessageDAO messageDAO;
18     @Override
19     public DocumentDTO addDocument(Long communityID, String userID, byte[]
   ↳ data, String filename) {
20         Document addedDocument;
21         User user;
22
23         // Validierungshandling gefolgt von Error Handling
24         try {
25             if (communityID <= 0) throw new
   ↳ IllegalArgumentException("community must not be empty");
26             if (userID == null) throw new IllegalArgumentException("user must
   ↳ not be empty");
27             if (data == null) throw new IllegalArgumentException("uploaded
   ↳ file must not be empty");
28             if (filename == null) throw new
   ↳ IllegalArgumentException("filename must not be empty");
29
30             Documentlibrary documentlibrary =
   ↳ documentlibraryDAO.findByCommunityId(communityID);
31
32             //create a document library, if there isn't already one in the
   ↳ database
33             if (documentlibrary == null) {
34                 documentlibrary = addDocumentlibrary(communityID);
35             }
36
37             user = userDAO.getByUserId(userID);
38
39             addedDocument = new Document(documentlibrary, user, filename,
   ↳ data);
40             documentDAO.insert(addedDocument); // Transaktionshandling
41             logger.info(String.format("Document %s saved in database",
   ↳ filename));
42             // Error Handling

```

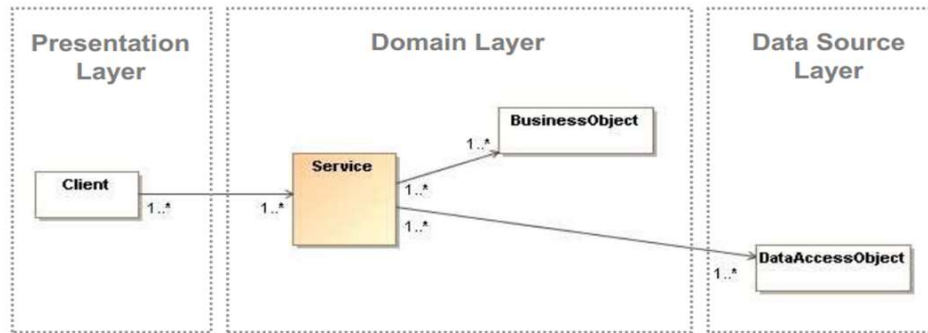
```

43     } catch (IllegalArgumentException iaex) {
44         String errorMsg = "Uploading file failed (illegal argument)";
45         logger.error(errorMsg, iaex);
46         throw new ServiceException(errorMsg);
47
48     } catch (Exception ex) {
49         String errorMsg = String.format("Uploading file %s failed.",
50             ↪ filename);
51         logger.error(errorMsg, ex);
52         throw new ServiceException(errorMsg);
53     }
54
55     String msgText = "Uploaded Document " + filename + " by user " +
56     ↪ user.getUserId();
57     addMessageToStream(communityID, user, msgText, addedDocument);
58     return DocumentMapper.toDTO(addedDocument);
59 }
60
61 private void addMessageToStream(Long communityID, User user, String text,
62     ↪ Document document) {...}
63
64 private Documentlibrary addDocumentlibrary(Long communityID) {
65     logger.info("Create missing documentlibrary");
66     Community community;
67     Documentlibrary documentlibrary = new Documentlibrary();
68     documentlibraryDAO.insert(documentlibrary); // Delegation zum DAO
69     community = communityDAO.findById(communityID); // Delegation zum
70     ↪ DAO
71     community.setDocumentlibrary(documentlibrary); // Delegation zur
72     ↪ Business Logik (Entity)
73     communityDAO.update(community); // Delegation zum DAO
74     return documentlibrary;
75 }
76
77 @Override
78 public List<DocumentDTO> getDocumentsFromCommunity(Long communityID)
79     ↪ {...}
80
81 @Override
82 public List<DocumentDTO> getDocumentsFromUser(String userID) {...}
83
84 @Override
85 public void removeDocument(Long documentID) {...}
86
87 @Override
88 public DocumentDTO getDocumentById(Long documentID) {...}
89 }

```

3.2 Nenne die Konsequenzen der Anwendung

- Reduzierung der Abhängigkeiten zwischen Presentation und Domain Layer



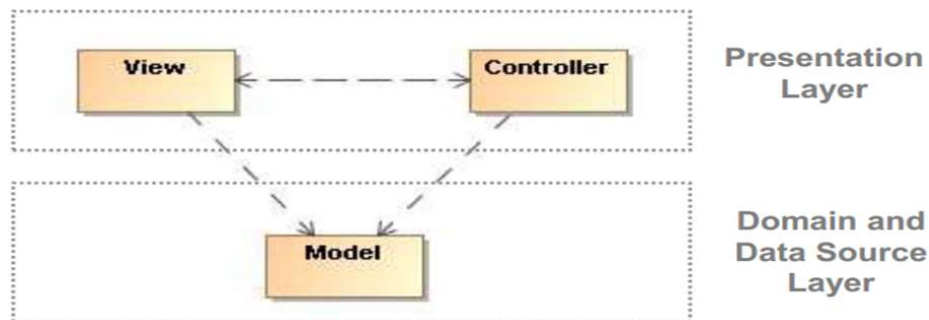
- Zentralisiertes Sicherheits und Transaktionshandling
- verbirgt vor Client Komplexität der Business Logik
- stellt Client ein grobkörniges Interface zur Verfügung
- gut für Remote Aufrufe geeignet (weniger Aufrufe)

4 Model-View-Controller (MVC) Pattern

4.1 Erkläre die Funktion + Skizze

MVC unterteilt eine interaktive Applikation in drei Teile: Model, View und Controller.

- Controller und View befinden sich im Presentation Layer und haben gegenseitig Abhängigkeiten
- Das Model darf keine Abhängigkeiten haben (Controller und View hängen vom Model ab)



4.1.1 Model

- Es befinden sich Teile im Domain und Data Source Layer.
- Das Model enthält die Kernfunktionalität und Daten. (z.B.: Datenbankzugriff)
- Im Projekt ...

4.1.2 View

- Im Projekt im Ordner WebApp zu finden.
- Enthält im Projekt xhtml Dateien zur Darstellung und User Interaktion

4.1.3 Controller

- Im Projekt sind Controllerklassen im Ordner web zu finden.
- Sie enthalten die Logik und behandeln Benutzereingaben

4.2 Nenne die Konsequenzen der Anwendung

- Visualisierung abhängig vom Model - nicht umgekehrt
- verschiedene Darstellungen möglich
- einfaches Testen der Domain-Logik
- gute Strukturierung
- View wird leichter austausch bzw. änderbar

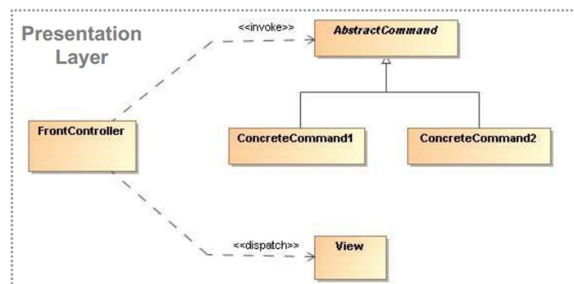
5 Front Controller

5.1 Erkläre die Funktion + Skizze

- Client schickt Request an Front Controller
- FC erfasst nur Infos die er für die weiter Delegation braucht
- FC gibt Request an entsprechenden ConcreteCommand oder View weiter
- es gibt zwei Implementierungsvarianten des Controller

– Servlet

– ConcreteCommand



5.2 Beschreibe ein konkretes Anwendungsbeispiel

- Java Server Faces (bei Java Server Faces enthält das File zwar keinen Java Code, interagiert aber direkt mit Java Code einer Backing Bean)
- Im Projekt wurde der Front Controller in Form eines Servlet realisiert, dessen Einbindung in der Konfigurationsdatei "web.xml" erfolgt:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   ↪  xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   ↪  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
   ↪  http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" version="3.1">
3  ...
4  <servlet>
5  <servlet-name>Faces Servlet</servlet-name>
6  <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
7  <load-on-startup>1</load-on-startup>
8  </servlet>
9  <servlet-mapping>
10 <servlet-name>Faces Servlet</servlet-name>
11 <url-pattern>*.xhtml</url-pattern>
12 </servlet-mapping>
```

5.3 Anwendung im Projekt

JavaServer Faces: mächtiges Framework, wo das MVC Pattern verwendet wird. Auf der einen Seite stehen die reinen Views (XHTML Seiten) und auf der anderen Seite Java Beans (Java Klassen), die als View Helper fungieren können. Beispiel: layered/MVC-JSF Primefaces ist eine konkrete Implementierung der JavaServer Faces (siehe POM.XML).

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE composition PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   ↪  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3  <ui:composition xmlns="http://www.w3.org/1999/xhtml"
4  xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
5  xmlns:h="http://xmlns.jcp.org/jsf/html"
6  xmlns:p="http://primefaces.org/ui"
7  xmlns:f="http://xmlns.jcp.org/jsf/core"
8  template="communityTemplate.xhtml">
9  <ui:define name="communityContent">
10 <h1>#{msg.document_manage_title}</h1>
11 <f:metadata>
12 <f:viewAction action="#{documentListController.loadDocumentsFromCommunity()}" />
13 </f:metadata>
14
15 <h:form id="doclistform">
16 <p:commandButton value="Refresh list"
   ↪  actionListener="#{documentListController.loadDocumentsFromCommunity()}"
   ↪  update="@form doclistform"></p:commandButton>
17 <p:dataTable id="doclisttable" value="#{documentListController.communityDocuments}"
   ↪  var="docs">
```

```

18 <p:column class="documenttimecolumn"
   ↳ headerText="#{msg.document_uploaded}">#{docs.createdTimestamp}</p:column>
19 <p:column class="documenttimecolumn"
   ↳ headerText="#{msg.label_userid}">#{docs.user.userId}</p:column>
20 <p:column headerText="#{msg.label_filename}">#{docs.filename}</p:column>
21 <p:column headerText="" class="documentbuttoncolumn">
22 <p:commandButton value="#{msg.button_download}" ajax="false"
23 onclick="PrimeFaces.monitorDownload(start, stop);">
24 <p:fileDownload value="#{documentController.downloadDocument(docs.id)"/>
25 </p:commandButton>
26 </p:column>
27 <p:column headerText="" class="documentbuttoncolumn">
28 <p:commandButton id="btnDel" value="#{msg.button_delete}"
29 actionListener="#{documentController.removeDocument(docs.id)}"
30 update="@form doclistform">
31 </p:commandButton>
32 </p:column>
33 </p:dataTable>
34 </h:form>
35 <h:form id="formdocupload" enctype="multipart/form-data">
36 <p:fileUpload id="fileupload"
37 dragDropSupport="false"
38 update="@form doclistform"
39 fileUploadListener="#{documentController.uploadDocument}"
40 allowTypes="/(\\.|\\/)(pdf|jpe?g|docx)$/" sizeLimit="5000000"
41 mode="advanced" label="Add document (.pdf .jpg .docx)">
42 </p:fileUpload>
43 </h:form>
44 <p:messages id="feedbackBox" severity="info,error" showDetail="true"
   ↳ showSummary="false">
45 <p:autoUpdate/>
46 </p:messages>
47 </ui:define>
48 </ui:composition>

```

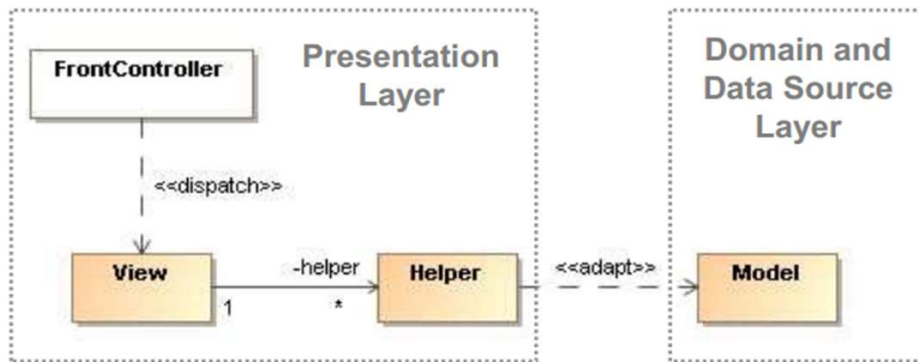
5.4 Nenne die Konsequenzen der Anwendung

- es muss nur EIN (Front) Controller konfiguriert werden
- da bei jedem Request ein neues Command Objekt erzeugt wird ist Thread-Safety nicht notwendig
- da nur EIN Controller sind auch Erweiterungen durch z.B.: Decorator einfach (auch zur Laufzeit)

6 View Helper

6.1 Erkläre die Funktion + Skizze

- View (xhtml-Dateien im Ordner webapp) delegiert Aufgaben an Helper (*Controller-Klassen - z.B. DocumentController im Ordner web)



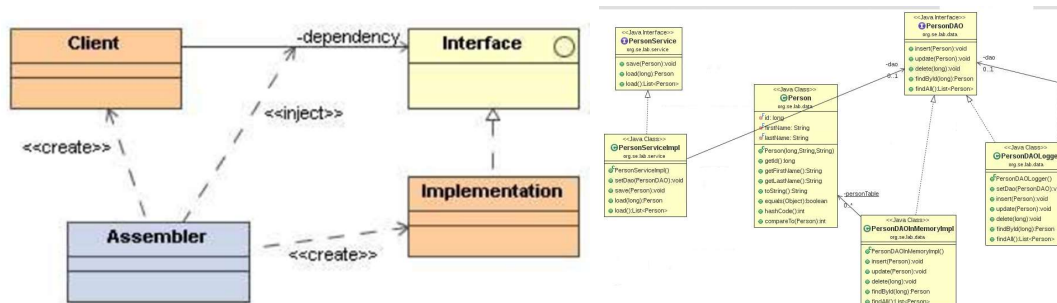
- Helper adaptieren View zu Model (Klassen in den Ordnern service und data)
- in View befindet sich HTML Code im ViewHelper Java Code zur Aufbereitung der Daten (+ wenig HTML)

6.2 Nenne die Konsequenzen der Anwendung

- kapselt Design-Code in View und View-Processing-Code Logik in Helper
- steigert Wiederverwendbarkeit, Wartbarkeit und Strukturierungsqualität der Anwendung
- vereinfacht Tests (Helperfunktionen ohne View)
- bessere Trennung zwischen
 - Presentation und Data Source Layer
 - Entwickler und Designer

7 Dependency Injection (don't call us, we'll call you)

7.1 Erkläre die Funktion + Skizze



- Grundidee sind loose gekoppelte Objekte
- Objekte werden mittels externem Assembler verknüpft

- Abhängigkeiten bestehen nur auf Interfaces
- Assembler Objekt (Framework) erzeugt die Interface-Implementierungen (z.B.: durch Factory)
- Es wird zwischen Constructor Injection und Setter Injection unterschiedlichen

```

1      // Constructor Injection
2      public class Client
3      {
4          private Interface iface;
5          public Client(Interface iface)
6          {
7              this.iface = iface;
8          }
9
10     // Setter Injection
11     public class Client
12     {
13         private Interface iface;
14         public setIface(Interface iface)
15         {
16             this.iface = iface;
17     }

```

- Im Spring Context: Dependency Injection mit XML-Datei
- alle Beans sind dort gelistet und werden verknüpft
- Context wird geladen damit alles verknüpft ist
- erspart Factories

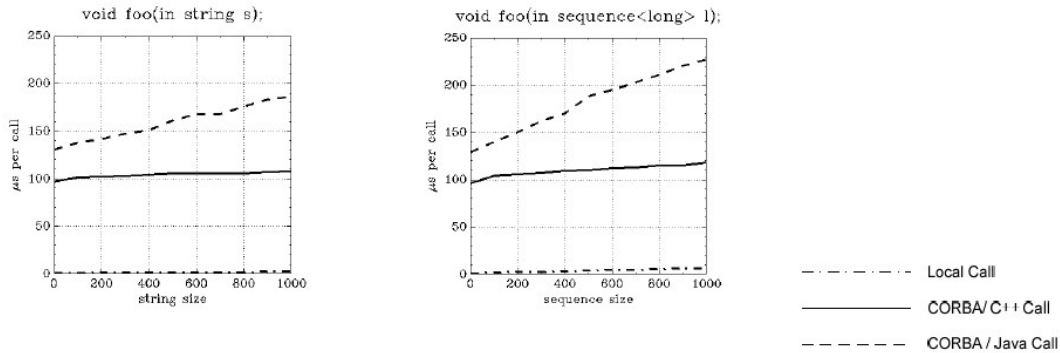
7.2 Nenne die Konsequenzen der Anwendung

- loose gekoppelte Objekte
- Referenzen nurmehr auf Interfaces
- hohe Flexibilität (Strategy, Proxy,..)
- bessere Erweiterbarkeit und Testbarkeit
- bei Spring kann Dependency Injection mittels XML oder Annotation erfolgen
 - Vorteil Annotation: Typ-Sicherheit (Tippfehler passieren schnell im XML)
 - Nachteil Annotation: nicht so flexibel wie XML

8 Data Transfer Object (DTO) Pattern

8.1 Erkläre die Funktion (Skizze - ein Grund für DTO)

- Transportiert Daten zwischen Prozessen um Remote Methodenaufrufe zu minimieren



- besteht aus Fields, Getter und Setter
- fasst Daten verschiedener Objekte zusammen die vom Remote Objekt benötigt werden
- ev. Map, Record Set, ...

8.2 Beschreibe ein konkretes Anwendungsbeispiel

8.3 Nenne die Konsequenzen der Anwendung

- kapselt und versteckt
- nimmt Komplexität
- steigert Effizienz da weniger Aufrufe über Remotegrenze

9 Page-Object-Pattern

PageObjectPattern

HTML – wrappen mit JavaCode, um es zu manipulieren

GUI-Test-Klasse und Page-Object

Über die Page-Object-Klasse manipulierte ich das HTML-Dokument

Bei HTML-Änderung muss ich nur Page-Objekt ändern und ansonsten nichts angreifen (Verkapselung)

10 Remote

11 Beschreibe die Unterschiede zwischen lokalem und Remote Interface Design

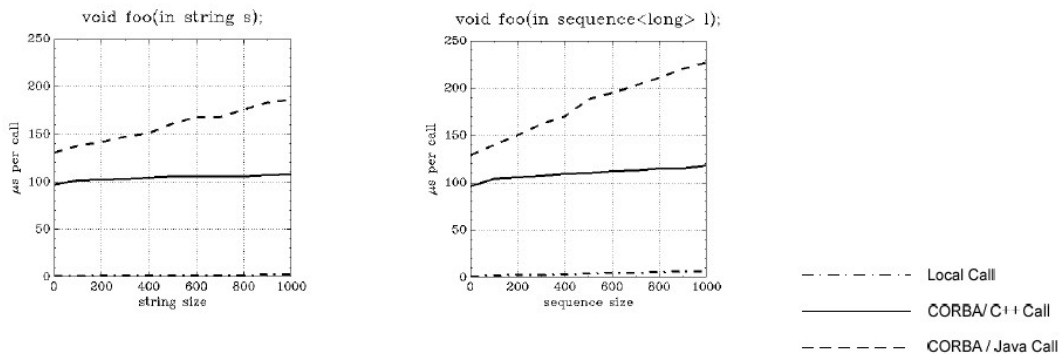
- Aufrufe innerhalb des Prozesses sind schneller als über Prozessgrenzen
- lokale Interfaces sind möglichst fein-granular während Remote Interfaces grob-granular sein müssen (weniger Aufrufe - Effizienz)
- viele kleine Aufrufe mit wenigen Daten sind "teuer" (Latenz durch Verbindungsherstellung)

12 Beschreibe drei Situationen wo Multiple Prozesse in Applikationen verwendet werden müssen

- Trennung zwischen Clients und Servern in Business Software
- Trennung zwischen server-basierter Applikationssoftware und Datenbank (SQL ist als Remote Interface designed, daher sind hier schnelle Abfragen möglich)
- Trennung wegen unterschiedlichen Anbietern oder Programmiersprachen

13 Beschreibe das folgende Diagramm. Was können wir daraus für das Design von Remote Interfaces folgern?

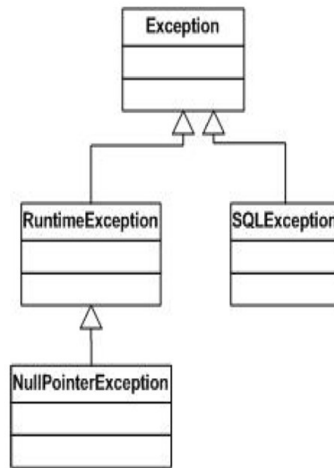
- speziell bei "teuren" Remote Calls ist es empfehlenswert weniger Calls mit großen Datenmengen anstatt vielen Calls mit wenigen Daten zu machen
- dieser Gedanke befürwortet auch den Einsatz von DTO um Calls und Daten zu bündeln



14 Exception Handling

15 Beschreibe den Unterschied zwischen Checked und Runtime Exceptions in Java (inkl. Klassendiagramm)

- Checked Exceptions (z.B. SQL-Exception) leiten von Exception Klasse ab und müssen behandelt werden (throws - catch)
 - Verwendung für Probleme die durch User behoben werden können (alternative Aktion)
- Unchecked Exceptions (z.B. NullPointerException) leiten von RuntimeException ab
 - Verwendung für technische Probleme (User kann nichts machen außer neu starten)



16 Beschreibe einen Use Case für eine Checked Exceptions in Java

- eine Netzwerkübertragung schlägt fehl - es ist vorgesehen, dass der Applikations-User dies neu anstoßen kann

17 Beschreibe einen Use Case für eine Runtime Exceptions in Java

- Die Datenbank ist beschädigt - die Exception geht durch alle Layer erst mit Implementierungsspezifischer Exception später mit Runtime ohne Stacktrace (Sicherheit) bis zum User.

18 Beschreibe 5 Best Practice Beispiele beim Einsatz von Exceptions

- Exceptions nicht für Programmflusskontrolle verwenden (schlechte Performance)
- offene Ressourcen schließen (try-with-resources bzw. close im finally)
- selbst erstellte Exceptions auch mit nützlichen Infos ausstatten
- Implementierungsspezifische Exceptions nicht bis zum User durchwerfen (stattdessen catch + throw RuntimeException)
- dokumentieren mit @throws im DOC, testen mit JUnit

19 Beschreibe 5 Exception Handling Anti Pattern

- Log and Throw (nie beides: entweder, oder)
- Throwing Exception bzw. catch Exception (spezifischere anstatt Basisklasse verwenden)

- Destructive Wrapping (wenn bei catch + throw = wrapping nicht die Original Exception weitergegeben wird)
- Log and return Null (provoziert an einer anderen Stelle eine NullPointerException)
- Catch and Ignore
- UnsupportedOperationException return Null (besser UnsupportedOperationException)

20 Logging

21 Nenne die Nachteile von debugging mit printf() sowie die Vorteile die Logging Frameworks wie log4j bieten

21.1 Nachteile printf()

- Produktiv-Code wird überfüllt -> erschwert Lesbarkeit
- Consolenausgabe wird bei vielen prints auch schnell unübersichtlich
- im Falle eines vorzeitigen Absturzes können Ausgabedaten verloren gehen
- Performance bei vielen Logprints

21.2 Vorteile Logging mittels Framework (z.B.: log4j)

- Nutzt ein einheitliches Format / Konventionen
- logging kann optional an und ausgeschalten werden
- durch verschiedene Log-level können Logs gefiltert erstellt werden
- Layout für Ausgabe kann zentral definiert/geändert werden

22 Project Structure

23 Toni

24 Annotationen

24.1 @MappedSuperclass

- ist im Hibernate Framework eine Klasse durch die gemeinsame Felder definiert werden.
- definiert eine abstrakte Superklasse

@Produces – kommt während deployment, markiert Factory Method damit man nicht direkt auf die Klasse zugreifen muss @Typed – zeigt die Vererbung Wieso bei uns allein stehend? @Named – Zeigt bei Mehrdeutigkeit das richtige Objekt mit dem Namen @Resource – fast wie Dependency Injection @Stateless – speichert den Client Status nicht @Entity – Data

Access Layer @Table – Tabellenname im SQL @Column – SQL-Spalten nullable=false @One-ToMany Beziehung @JoinColumns – welche Spalten zusammen gehören FK @OneToMany FK auf anderen Seite @ApplicationScoped – lebt die ganze Applikation lang, wird einmal gemacht. @PersistenceContext – persistence.xml auslesen für Treiber und andere JPA Geschichten + Data Source. Entity Manager Injection. @Id – das ist die id @GeneratedValue – Wert kommt aus der DB @Local – Klasse für lokale Aufrufe. @Remote – interprozessaufrufe. RMI @ApplicationException – Rollback wenn so eine Exception kommt, Nachricht zum Client.

25 Patterns in Practice

Data Access Layer Entity – Java Repräsentation vom DB Entity DAO damit man auf die Entities zugreifen kann. DB abstrahieren. Methoden mit denen man auf die DB zugreifen kann. DAOImpl – Implementierung DAOImpl – DAOException fehlt. Schlecht weil Input wird nicht kontrolliert. EntityManager in try catch, sonst kann es kleschen. Zusätzlich in DaoException wrappen. AbstractEntity – hier wird die id gemanaged Service Layer DTO – Aufrufgeschw. Verringern, nicht jedes Objekt einzeln aufrufen, sondern mit einmal alle notwendigen Objekte. Mapper – von DTO in Entity und Entity ins DTO.

FrontController web.xml ViewHelper *ServiceImpl

26 Konfigurationsdateien (pom.xml), (persistence.xml) und noch a bissl mehr Scheiß

Resource plugin – klar für Ressourcen Wildfly – server Primeafce = jsf Framework Jacoco = test Coverage Slf4j = logger Jaxb – xml Cdi = context dependancy injection

27 Reihenfolge - Wildfly - Abfolge - einzelne Schritte

Reihenfolge:

1. Compile
2. Surefire (unitTests)
3. Packaging - war file erstellen
4. Wildfly - fressen und deployen
5. Failsafe IT-test
6. MVN site
7. Gui test

28 Frageart Prüfung

Welche Fehler können bei Exception-Handling vorkommen in unserem Projekt?? – wie funktioniert es grundsätzlich in unserem Code

DocumentDAO – DocumentService – DocumentController – so sollte Exception-Handling implementiert werden

DAO wirft Exception – im ServiceLayer wird dies gefangen und der Stack-Trace wird im weggeelogt und eine benutzerfreundliche Fehlermeldung wird ausgegeben (Destructive Wrapping).

Alle Patterns, die vorkommen – praktische Beispiele aus dem Code

Was sind JavaBeans? Wie funktioniert das Konzept? Wie wird es genau implementiert?
NamedBean, TypedBean etc.

DTO

29 Die CONFIG-Files

29.1 web.xml

- konfiguriert den Java Webserver (Wildfly - JBOSS)
- befindet sich im Ordner `src/main/webapp/WEB-INF/web.xml`

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   ↪ xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   ↪ xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
   ↪ http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" version="3.1">
3 ...
4 <servlet>
5 <servlet-name>Faces Servlet</servlet-name>
6 <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
7 <load-on-startup>1</load-on-startup>
8 </servlet>
9 <servlet-mapping>
10 <servlet-name>Faces Servlet</servlet-name>
11 <url-pattern>*.xhtml</url-pattern>
12 </servlet-mapping>
13
14 <!-- Security roles -->
15 <security-role>
16 <description>administrators</description>
17 <role-name>ADMIN</role-name>
18 </security-role>
19 <security-role>
20 <description>portal administrators</description>
21 <role-name>PORTALADMIN</role-name>
22 </security-role>
23 <security-role>
24 <description>standard user</description>
25 <role-name>USER</role-name>
26 </security-role>
27
28 <!-- Security constraints -->
29 <security-constraint>
30 <web-resource-collection>
31 <web-resource-name>admin area</web-resource-name>
32 <url-pattern>/admin/*</url-pattern>
33 </web-resource-collection>
34 <auth-constraint>
35 <role-name>ADMIN</role-name>
36 </auth-constraint>
37 </security-constraint>
```

```

38
39 <security-constraint>
40 <web-resource-collection>
41 <web-resource-name>community area</web-resource-name>
42 <url-pattern>/community/*</url-pattern>
43 </web-resource-collection>
44 <auth-constraint>
45 <role-name>USER</role-name>
46 <role-name>PORTALADMIN</role-name>
47 <role-name>ADMIN</role-name>
48 </auth-constraint>
49 </security-constraint>
50
51 <security-constraint>
52 <web-resource-collection>
53 <web-resource-name>user administration area</web-resource-name>
54 <url-pattern>/userAdministration/*</url-pattern>
55 </web-resource-collection>
56 <auth-constraint>
57 <role-name>USER</role-name>
58 <role-name>PORTALADMIN</role-name>
59 <role-name>ADMIN</role-name>
60 </auth-constraint>
61 </security-constraint>
62
63
64 <security-constraint>
65 <web-resource-collection>
66 <web-resource-name>user functionalities</web-resource-name>
67 <url-pattern>/user.xhtml</url-pattern>
68 <url-pattern>/userlist.xhtml</url-pattern>
69 <url-pattern>/notImplemented.xhtml</url-pattern>
70 </web-resource-collection>
71 <auth-constraint>
72 <role-name>USER</role-name>
73 <role-name>PORTALADMIN</role-name>
74 <role-name>ADMIN</role-name>
75 </auth-constraint>
76 </security-constraint>
77
78 <security-constraint>
79 <web-resource-collection>
80 <web-resource-name>other functionalities</web-resource-name>
81 <url-pattern>/notImplemented.xhtml</url-pattern>
82 </web-resource-collection>
83 <auth-constraint>
84 <role-name>USER</role-name>
85 <role-name>PORTALADMIN</role-name>
86 <role-name>ADMIN</role-name>
87 </auth-constraint>
88 </security-constraint>
89
90 <login-config>

```

```
91 <auth-method>FORM</auth-method>
92 <realm-name>pse</realm-name>
93 <form-login-config>
94 <form-login-page>/login.xhtml</form-login-page>
95 <form-error-page>/login.xhtml</form-error-page>
96 <!--           <form-error-page>/loginerror.xhtml</form-error-page> -->
97 </form-login-config>
98 </login-config>
99 </web-app>
```