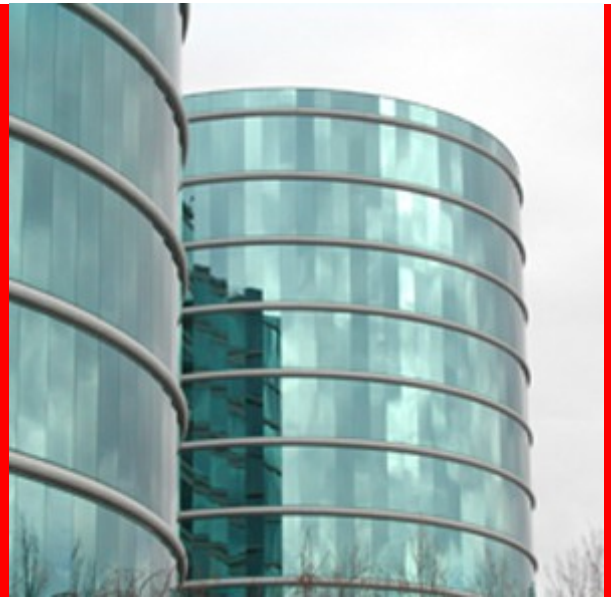


ORACLE®

„Clean Code“ in der Java EE - Welt

Alexander Fox <alexander.fox@oracle.com>
Senior Berater

Agenda

Dauer ca. 45 Minuten

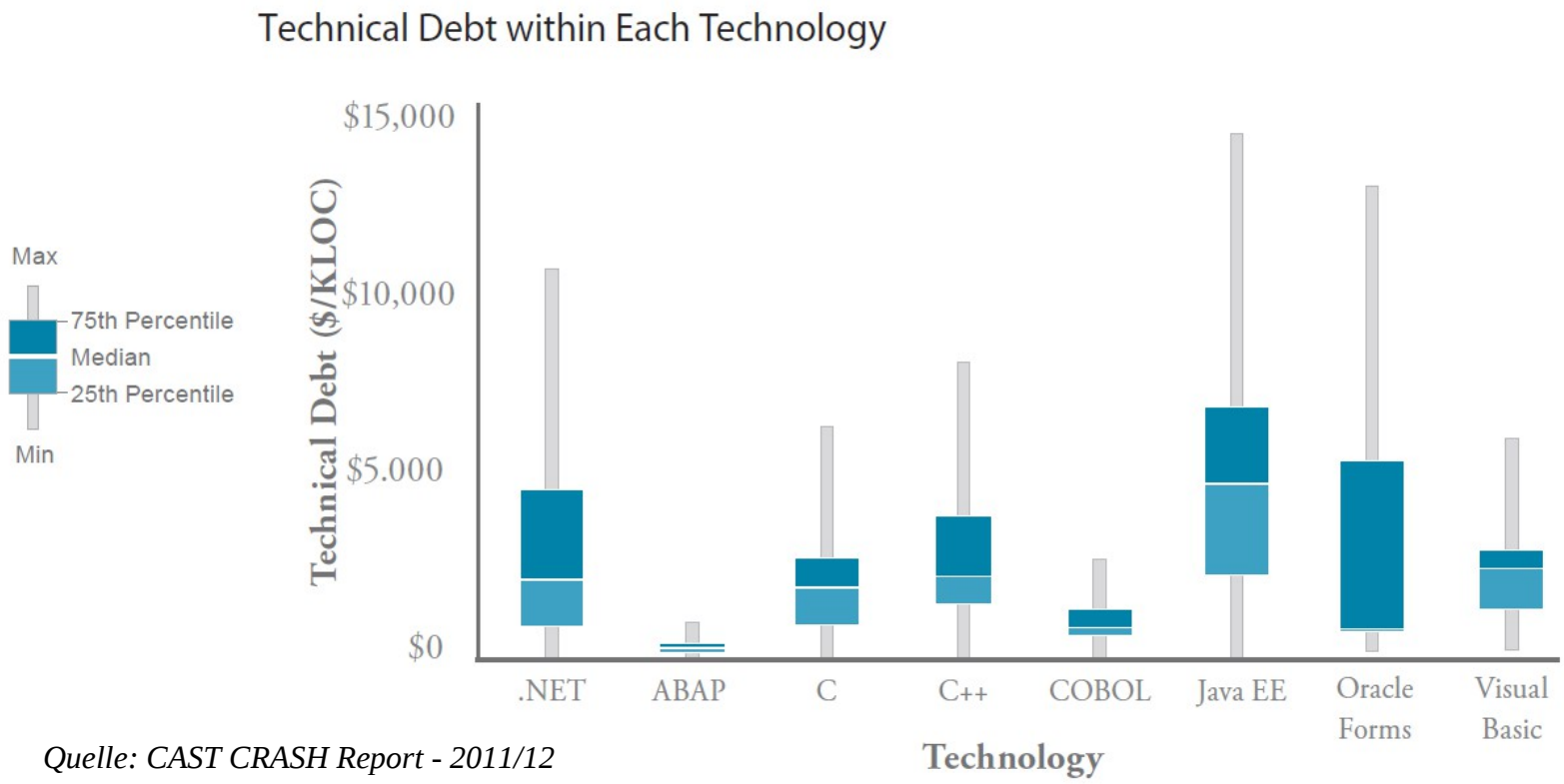
- Einführung und Allgemeines zu Clean Code
- Clean Code in Java
- Clean-Code Beispiele aus Java EE
  - Java Persistence API (JPA)
  - Context and Dependency Injection (CDI), EJB
  - Java Server Faces (JSF)
- Modularisierung (mit Maven)

Probleme bei „unsauberem“ Code

Unsauberer Java EE Code ...

- lässt sich schwer verstehen
- lässt sich schwer reparieren
- lässt sich kaum erweitern
- läuft im Produktionssystem
- ist leider kompilierbar
- macht „fast“ genau das, was er soll

Warum Clean Code in Java EE?



Quelle: CAST CRASH Report - 2011/12

Anzeichen für „unsauberen“ Code

- Geringe Testabdeckung
- Redundanter Code (Sauberer Code ist minimal)
- Zu große Klassen/Methoden
- Zu hohe Komplexität von Klassen/Methoden

- Code Ownership hat hohe Bedeutung
- Fehlende Code Reviews

Beispiel für Clean Code

Listing 3-3: HtmlUtil.java (re-refactored), Quelle: Clean Code, Robert C. Martin, 2009, S. 35

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception
{
  if (isTestPage(pageData)) {
    includeSetupAndTeardownPages(pageData, isSuite);
  }
  return pageData.getHtml();
}
```

Allgemeines zu Clean Code in JavaEE

- Java EE ist die Sammlung verschiedener Frameworks
- Frameworks wurden z. T. separat entwickelt
- Viele alternative Lösungsmöglichkeiten existieren, z.B. CDI und EJB
- Unterschiedliche Herangehensweisen und Lösungsmöglichkeiten

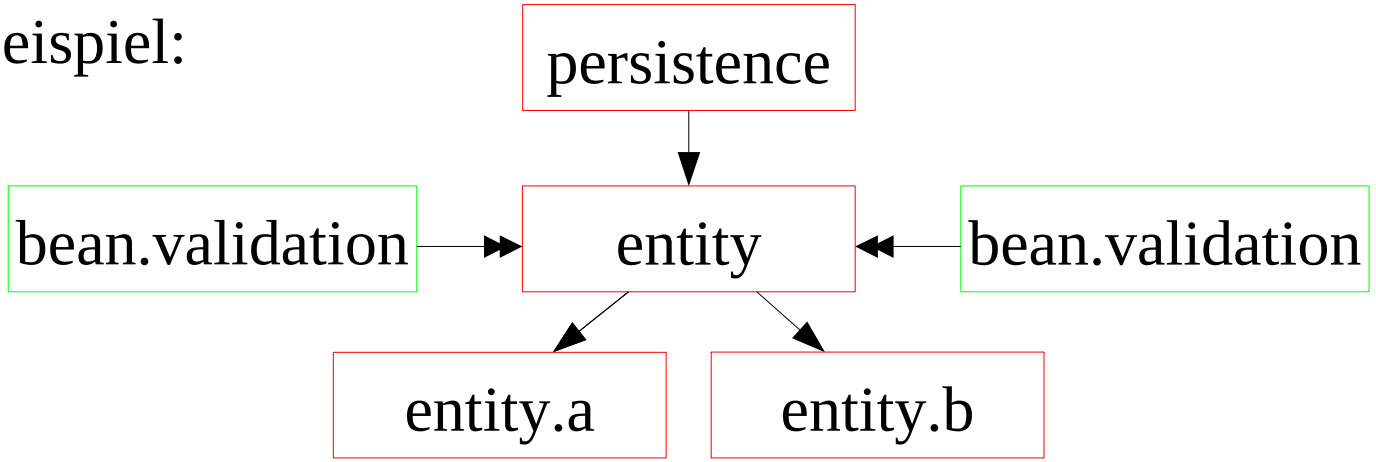
Umgang mit Workarounds

```
public class VeryComplexBusinessClass {
    // This is a workaround
    @EJB
    private SampleEjb service;
    // ... Other EJBs
    // ... Other logic
}
public class WorkaroundEjbInitializer {
    /** This is a workaround for bug xxx opened
     *  from SR yyy http://zzz
     */
    @EJB
    private SampleEjb service;
    // ... all other EJBs
}
```

Modularisierung von Projekten mit Maven

- Festlegung einer Architektur und Architekturrichtlinien
- Definition von Schichten
- Aufbau von Maven Hierarchien (Multi-Modulen) und Aufteilung nach Technologien
- Mögliche Validierung von Architekturrichtlinien durch Maven-Plugins

Beispiel:



Java Persistence API

```
@Entity
public class MyEntity {
    @Id
    private Long id;
    @Enumerated
    private Greeting greeting;
...}
```

Speicherung: Ordinal!

```
@Entity
public class MyEntity {
    @Id
    private Long id;
    @Enumerated(EnumType.STRING)
    private Greeting greeting;
...}
```

Speicherung: Name

- Fehlende Check-Constraints bei SQL-Generierung
- JPA-Relationship ist vorzuziehen (FK-Constraints)

Modularisierung von Projekten mit CDI

```
public class CallingClassOldStyle {
    MyInterface dependent = new MyClass();
    public String getName() {
        return dependent.getSomeThing();
    }
}
```

```
@Named public class CallingClass {
    @Inject private MyInterface dependent;
    public String getName() {
        return dependent.getSomeThing();
    }
}
```

CDI-, bzw. EJB-Interceptor (Beispiel)

```
@MyErrorHandler
@Interceptor
public class ErrorHandlerInterceptor {
  @AroundInvoke
  protected Object invoke(InvocationContext ctx) throws Exception {
    try {
      return ctx.proceed();
    } catch (Exception e) {
      handleExeption(e, ctx.getParameters()[0]);
      throw e;
    }
  }
  private void handleExeption(Exception e, Object firstParam) {
    // The handling
  }
}
```

```
@Named
@MyErrorHandler
public class ErrorHandledClass {
...
}
```

JSF und Scopes (CDI @RequestScope)

- @RequestScope garantiert maximale Skalierbarkeit der Anwendung, solange die Zugriffe auf die Datenschicht performant umgesetzt werden können
- Problem 1 JSF LifeCycle: Jeder „Getter“ wird mehrfach aufgerufen
- Problem 2 JSF LifeCycle: Beim Wiederherstellen der View (RestoreView) müssen bei dynamischen Views im @RequestScope schon alle „Setter“ gefeuert worden sein
- => @RequestScope ist in „statischen“ Anwendungsteilen einzusetzen
- Z.B. Keine Dynamischen Selectlisten, dynamisch erscheindende commandButtons

JSF und Scopes (CDI)

- Scopes wirken "ansteckend“.
- Wird eine @RequestScoped Managed Bean A in eine Managed Bean B mit längerer Lebensdauer injeziert, so enthält B beim zweiten Request „veraltete“ Daten

JSF und Scopes (Fazit)

- Durchgehende Software Entwicklung im @RequestScope ist meist sehr schwierig und erzeugt schwerer lesbare JSF Seiten
- Ohne Veränderung des JSF-LifeCycles ist der Einsatz von @RequestScope manchmal unmöglich
- Ein Scope „über“ @RequestScope ist notwendig
- @ConversationScope oder @SessionScope müssen eingesetzt werden

JSF und Scopes (Nachteile)

- Mehr Information müssen im Speicher gehalten werden
- Verringerte horizontale Skalierbarkeit (Sticky Sessions)
- Daten müssen „aktuell“ gehalten werden (Gefahr der Refresh-Inflation)
- Refreshes sind aus Clean Code Sicht Seiteneffekte
- Vermeidung einiger negativer Effekte der „Refreshes“ mit CDI Observern (z.B. Dependencies)

Exception Handling

- Exceptions sollten lokal für das Modul definiert werden

```
try {
    out = new FileOutputStream(path.toFile());
} catch (FileNotFoundException e) {
    throw new ZipExtractorException("Cannot extract file: " +
targetPath, e);
}
```

- Viele Fehler-Situationen führen zu „internen Fehlern“
- Für „interne Fehler“ sollten RuntimeExceptions verwendet werden
- Wird das Modul in andernen Kontexten verwendet?
- Welche Aktion, die der End-Benutzer ausführen kann, soll ausgelöst werden?

Exception Handling in JSF

- RuntimeExceptions können in JSF über ExceptionHandler verarbeitet werden
- „Unnötiges“ Exceptionhandling verschwindet aus dem Code
- Der ExceptionHandler kann über den currentFacesContext Nachrichten erzeugen, die dann über <h:messages> angezeigt werden können

```
public class GlobalExceptionHandlerWrapper
    extends ExceptionHandlerWrapper {
    @Override
    public void handle() throws FacesException {
        // Implementation
    }
}
```

Achtung bei der Anzeige von Messages

- Sofort erfassbar für den Benutzer ist die Anzeige von Fehlern und Warnungen oben auf der Seite
- <h:messages> (auch <rich:messages>) zeigt nicht alle Nachrichten unter allen Umständen an, was eine Platzierung unten auf der Seite notwendig macht (Ajax)
- Neue JSF-Lifecycle Phase „HandleExceptions“ notwendig?
- Evtl. absolute Positionierung auf der Seite

Verwendung von JSF-Convertern

- JSF-Converter übernehmen die Konvertierung von Objekten
- Unnötige Getter und Setter aus Managed Beans entfallen, was zu weniger Code und kleineren Klassen führt
- Der „Build“ von komplexen Objekten wird aus den Managed Beans entfernt

```
<h:inputText size="30" value="#{helloMb.entity}"
                      converter="#{testEntityConverter}"
/>
```

Verwendung von JSF-Convertern (Code)

```
@Named
public class TestEntityConverter implements Converter {
    @Override
    public Object getAsObject(FacesContext context, UIComponent
component, String value) {
        if (value == null) {
            return null;
        }
        return new TestEntity(Long.parseLong(value), name);
    }

    @Override
    public String getAsString(FacesContext context, UIComponent
component, Object value) {
        if (value == null) { return null; }
        TestEntity entity = (TestEntity) value;
        return entity.getId() + "";
    }
}
```

JSF-View Modul

- Ein „View Modul“ ist eine Sammlung von XHTML-Seiten., Property Bundle und Managed Beans die einen Teil des UIs implementieren
- „View Module stehen zwischen „JSF Composite Components“ (seit JSF 2.0)
- Achtung: „<f:loadBundle var=“text“> verwendet einen globalen Namensraum (liegt im Request)

Modul Layout:

```
src/main/java
src/main/resources/my/module/module.properties
src/main/resources/META-INF/beans.xml
src/main/resources/META-INF/resources
src/main/resources/META-INF/resources/content
src/main/resources/META-INF/resources/content/view.xhtml
```

JSF-View Modul Refactoring

```
                        <h:graphicImage value="/images/rollup.gif" />
                        <f:ajax event="click" render="history" listener="#{navigationMb.toggleHistoryPanel}" />
                    </a4j:commandLink>
                </f:facet>
                <a4j:commandLink value="#{globalResource['Allgemein']}" styleClass="allgemein big" action="#{TreeHandler.showAllgemeinHistoryAction}" />
                <br />
                <a4j:commandLink value="#{globalResource['Bonitaet']}" styleClass="bonitaet big" action="#{TreeHandler.showBoniHistoryAction}" />
                <br />
                <a4j:commandLink value="#{globalResource['Stoerung']}" styleClass="stoerung big" action="#{TreeHandler.showStoerungHistoryAction}" />
                <br />
                <a4j:commandLink value="#{globalResource['Vertraege']}" styleClass="vertrag big" action="#{navigationMb.toggleHistoryList}" />
                <h:panelGroup rendered="#{navigationMb.showHistoryList}">
<?xml version='1.0' encoding='UTF-8'?>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:ui="http://java.sun.com/jsf/facelets" xmlns:h="http://java.sun.com/jsf/html" xmlns:f="http://java.sun.com/jsf/core"
    xmlns:rich="http://richfaces.org/rich" xmlns:a4j="http://richfaces.org/a4j">
<body>
    <ui:composition>
        <!-- Parameter to be saved in page -->
        <h:panelGroup id="navigationPanelGroup">
            <rich:panel id="navi_mini" rendered="#{!navigationMb.navigationShow}" styleClass="navi_mini" bodyClass="emptyBodyClass">
                <f:facet name="header">
                    <h:outputText value="Navigation" />
                    <f:ajax event="click" render="content navigationPanelGroup" listener="#{navigationMb.show}" />
                </f:facet>
            </rich:panel>
            <rich:panel id="navi" rendered="#{navigationMb.navigationShow}" styleClass="navi_max" bodyClass="emptyClass">
                <rich:panel bodyClass="emptyBodyClass" styleClass="emptyClass">
                    <f:facet name="header">
                        <h:outputText value="Navigation" />
                        <f:ajax event="click" render="content navigationPanelGroup" listener="#{navigationMb.hide}" />
                    </f:facet>
                </rich:panel>
                <ui:include src="nav_customer_data.xhtml"/>
                <ui:include src="nav_history.xhtml"/>
                <ui:include src="nav_activity.xhtml"/>
                <ui:include src="nav_document.xhtml"/>
            </rich:panel>
        </h:panelGroup>
    </ui:composition>
</body>
</html>
                    <rich:treeModelRecursiveAdaptor id="naviTreeRecursiveAdaptor" leaf="#{node.leaf}" roots="#{navigationMb.nodes}" nodes="#{node.childsFiltered}">
                        <rich:treeNode expanded="#{node.gutExpandingStateOfNode.expanded}" iconLeaf="/images/tree/folder.png" iconCollapsed="/images/tree/folder.png" iconExpanded="/images/tree/folder-open.png">
                            <h:outputText value="#{node.schlagwort} (#{node.numberOfDocumentsWithChilds})" styleClass="#{node.style}" title="#{node.schlagwort} (#{node.numberOfDocumentsWithChilds})"/>
                        </rich:treeNode>
                    </rich:treeModelRecursiveAdaptor>
                </rich:tree>
            </rich:panel>
        </rich:panel>
    </h:panelGroup>
</ui:composition>
</body>
</html>
```

Integration von JSF-View Modulen

```
<dependency>
    <groupId>groupid</groupId>
    <artifactId>web.view.index</artifactId>
</dependency>
```

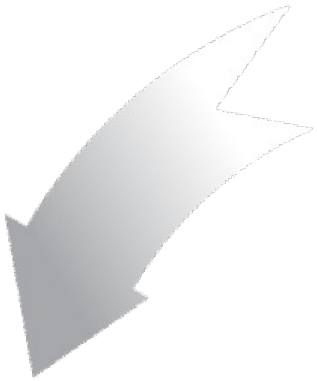
- „View Module“ erhöhen die Lesbarkeit von JSF Komponenten, indem sie JSF Seiten verkleinern und Wiederverwendbarkeit fördern.

Fazit und Ausblick

- Java EE stellt Mittel zur Verfügung, um Clean Code zu programmieren
- Java EE 6 ist mächtig und komplex
- Weitere Clean Code Paradigmen müssen gefunden und diskutiert werden

Literatur

- CAST Technical Debt Estimation:
  http://www.castsoftware.com/research-labs/technical-debt-estimation
- Clean Code: A Handbook of Agile Software Craftsmanship; Robert C. Martin, 2008
- The Java EE 6 Tutorial:
  http://docs.oracle.com/javaee/6/tutorial/doc/
- Core JavaServer Faces (Sun Core Series); Geary, Horstmann, 2010




Q&A