"Clean Code" in der Java EE-Welt

Alexander Fox
Oracle Deutschland B.V. & Co. KG
Frankfurt

Schlüsselworte:

Java JavaEE Software-Entwicklung Cacheless-Programming

Einleitung

Das Thema "Clean Code" wurde in der Java-Community in den letzten Jahren ausführlich betrachtet und es wurden für Java SE einige Prinzipien entwickelt, die sich auch in der Praxis bewährt haben.

Wie sich durch jüngste Untersuchungen zeigte, werden bei der Entwicklung von Java EE Anwendungen in höherem Maße "technische Schulden" (technical dept) eingegangen, als es bei der Entwicklung mit anderen Frameworks der Fall ist. Eine Möglichkeit diese “Schulden” zu reduzieren ist, sich bei der Software-Entwicklung an "Clean Code" Grundsätzen zu orientieren. Was bedeutet aber Clean code für Java EE?

Dieser Vortrag beleuchtet das Thema Clean Code in der Java EE-Welt und arbeitet Themen heraus, die sich von "Clean Code" in Java SE unterscheiden, bzw. spezifisch für Java EE sind. Nach einer kurzen Einführung in "Clean Code" in Java SE, wird speziell auf Clean Code Paradigmen im Umfeld JSF, EJB und CDI eingegangen.

Zielgruppe des Vortrags sind Java EE-Entwickler und Manager, die sich für den Umbau von Legacy Java EE und den Neuaufbau von Java EE Anwendungen interessieren.

Clean Code, ein Versuch einer Definition.

Der Begriff Clean Code wurde von Robert C. Martin in und durch sein Buch „Clean Code“ geprägt.

Clean Code ist Code ist Code der

- die Funktionen abdeckt, die er ausführen soll
- die Funktionen so performant wie nötig ausführt
- leicht zu verstehen ist
- intuitive verständliche Namen verwendet (für Klassen, Methoden und Variablen)
- leicht modifizierbar und erweiterbar ist
- durch automatische Tests abgedeckt ist
- Kurz: Code, der nach dem Stand der Technik entwickelt wurde

Warum “Clean Code” in Java EE?

Wie die der CAST CRASH Report – 2011/12 es darstellt, gehen Projekte, die auf Basis von JavaEE implementiert werden, höhere “Technische Schulden” ein, als dies bei anderen Technologien der Fall ist. Allerdings zeigt Abbildung 1, dass es für JavaEE auch eine hohe Streuung in der technischen

Schuld gibt. Das legt nahe, dass die Situation vom jeweiligen Projekt abhängt und nicht in der Wahl der Technologie zu suchen sein kann.

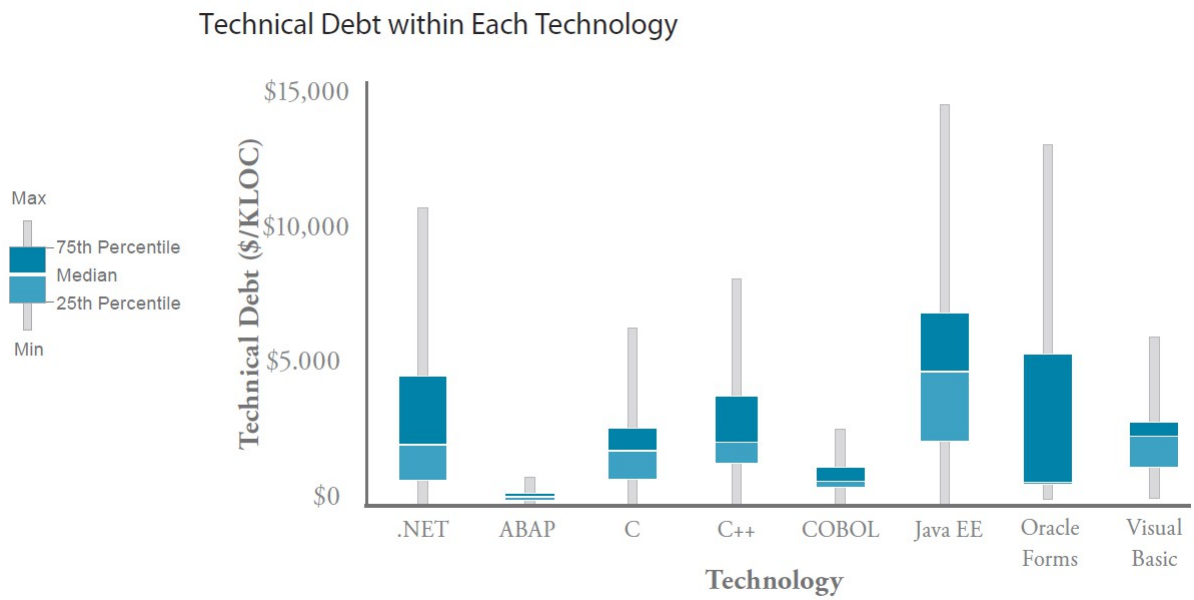

Abbildung 1: Quelle: CAST CRASH Report - 2011/12

Clean Code kann ein Mittel sein, die technische Schuld so gering wie möglich zu halten.

Was sind die Hauptprobleme von “unsauberen” Code?

Unsauberer Java EE Code ist:

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

Woran erkennt man “unsauberen” Code?

Festzustellen, ob man “unsauberen” Code in seinem Projekt einsetzt, ist gar nicht so einfach. Zu oft wird das Problem zu einfach angegangen:

- Eigener Code ist sauber
- Fremder Code unsauber

Werkzeuge zur Code-Analyse (z.B. Sonar) können einen groben Hinweis geben, ob die “unsauberer Code” vorliegt. Mit diesen Werkzeugen läßt sich ein erster Überblick herstellen, der wichtige Hinweise geben kann:

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

Endgültige Gewissheit liefern diese Metriken allerdings nicht, sondern können als Anhaltspunkte gesehen werden.

Ein Beispiel für Clean Code in Java SE

Um einen Eindruck dafür zu bekommen, um was es bei Clean Code geht, wird ein Beispiel besprochen.

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();
    }
```

Was ist gut gelöst

- Methoden haben aussagekräftige Namen
- Variablen haben aussagekräftige Namen
- Die Methode renderPageWithSetupsAndTeardowns ist kurz und prägnant

Was kann besser gemacht werden

- Die Exception von renderPageWithSetupsAndTeardowns sollte typisiert oder eine RuntimeException sein (z.B. PageRenderingException)
- isSuite Parameter ist ein boolescher Parameter und sollte ersetzt werden durch zwei Methoden renderPageWithSetupsAndTeardownsForSuite und renderPageWithSetupsAndTeardownsForTest
- pageData ist ein In/Out Parameter, der Methodenaufruf enthält somit einen Seiteneffekt
- Beim IF Statement werden keine Klammern verwendet, was zukünftig zu Fehlern führen kann
- Die Methode renderPageWithSetupsAndTeardowns ist static, was das Überschreiben der Methode erschwert (kein Dynamic Dispatch, somit reduzierte Objektorientierung)

Umgang mit Workarounds

Workarounds sind aus verschiedenen Gründen notwendig:

- Implementierungsfehler der eingesetzten Software
- Die eingesetzte Software verhält sich nicht so wie geplant
- Die für das Projekt entwickelte Architektur „passt“ an einer Stelle nicht

Dem Umgang mit Workarounds ist ein hoher Stellenwert zuzurechnen, da Workarounds erfahrungsgemäß oft eingesetzt werden, den Sinn eines Programmfragments aber verschleiern können.

Beispiel für einen „versteckten“ Workaround

```
public class VeryComplexBusinessClass {
    // This is a workaround
    @EJB
    private SampleEjb service;
    // ... Other EJBs
    // ... Other logic

}
```

Beispiel für einen gut erkennbaren Workaround

```
public class WorkaroundEjbInitializer {
    @EJB
    private SampleEjb service;
    // ... all other EJBs
}
```

Clean Code ist:

- den Workaround so „prominent“ wie möglich darzustellen
- Einen Service Request oder ein anderes Ticket zu öffnen, damit der Workaround irgendwann entfernt werden kann

Besseres Beispiel für einen gut erkennbaren Workaround

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

CDI (Contexts and Dependency Injection)

CDI ist ein leichtgewichtiges Container-Framework, das Entkopplung von Schnittstellen und Implementierung ermöglicht. Es stellt ein Event/Notification-System und die Möglichkeit des Lifecycle-Managements bereit. Weiterhin integriert CDI in EL (Expression Language), was seinerseits in JSF Verwendung findet.

Modularisierung mit CDI

Mit CDI kann leicht Interface und Implementierung getrennt werden.

Beispiel für „harte“ Abhängigkeiten

```
public class CallingClassOldStyle {
    MyInterface dependent = new MyClass();

    public String getName() {
        return dependent.getSomeThing();
    }
}
```

Beispiel für Trennung von Interface und Implementierung

```
@Named
public class CallingClass {
    @Inject
    private MyInterface dependent;

    public String getName() {
        return dependent.getSomeThing();
    }
}
```

Warum überhaupt noch EJB (Session Beans)?

Im Gegensatz zu CDI bieten EJBs einen Transaktionskontext, der in CDI erst bereitgestellt werden müsste. Weiterhin können EJBs leicht „veröffentlicht“ werden (Remote EJBs, JAX-WS). In einer Anwendung sollte man sich auf ein einziges Transaktions-Handling reduzieren. Momentan wäre die Verwendung von Session Beans die minimale Implementierung und somit Clean Code.

CDI- oder EJB-Interceptor

Interceptors können die Anzahl der programmierten Zeilen Code stark verringern. Code in Interceptors implementiert selten funktionale Anforderungen, sondern implementiert Querschnittfunktionen. Selbst wenn ein Interceptor nur einmal verwendet wird kann er Klassen und Methoden verkleinern und die Querschnittfunktion zentral pflegbar machen, was Wiederverwendbarkeit fördert.

Beispiel wie ein CDI-Interceptor den Code klarer strukturiert

```
@InterceptorBinding
@Retention(RetentionPolicy.RUNTIME)
@Target({ TYPE, METHOD, FIELD })
@Qualifier
public @interface MyErrorHandler {}

@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
    }
}
```

Verwendung von Scopes und JSF

Es ist eine altbekannte Weisheit, dass man bei Web-Entwicklungen so wenig wie möglich in der Session speichern soll. Durch die Verwendung des Request Scopes sollte also bessere Performance und Skalierbarkeit erreicht werden können.

Eine JSF Anwendung zu entwickeln, die weitestgehend im Request Scope liegt ist mit JSF fast nicht möglich. Das Fehlen des Server Status führt zu kompliziertem Code. Der Server-Status muss konsistent gehalten werden.

Wann kann Request Scope verwendet werden?

Request Scope kann in einfachen Eingabe-Masken (Texteingaben) verwendet werden oder in Masken mit statischen Listen oder Tabellen. Sollte eine der Bedingungen nicht erfüllt sein, sollte ein anderer Scope benutzt werden.

Probleme mit Request Scope:

- Getter werden in jeder JSF Lifecycle Phase einmal aufgerufen, was in einem zustandslosen Modell normalerweise zu schlechter Performance führt.
- Tabellen und (Multi-) SelectListen können nicht dynamisch aufgebaut werden
- Beim Einsatz von JSF Frameworks ist der partielle Neuaufbau (Rendern) der Anwendung oft nicht kontrollierbar, was zu weiteren aufrufen der Request Scoped Bean und der weiteren Verschlechterung der Performance führt.
- Höherer Scope „infiziert“ andere Managed Beans

Da Request Scope in normalen Anwendungen nicht leicht und performant verwendet werden können, bleiben noch der Conversation Scope und der Session Scope übrig.

Probleme mit Session Scope

- Mehr Informationen müssen im Speicher gehalten werden
- Verringerte horizontale Skalierbarkeit, Sticky Sessions
- Zustands-Modell des Servers muss konsistent gehalten werden
- Aktualität der Daten ist nicht gewährleistet (Gefahr der Refresh-Inflation)
- Arbeiten mit mehreren Browser Tabs nicht einfach möglich

Conversation Scope

Conversation Scope ist prinzipiell das Gleiche wie der Session Scope. Conversation Scope hat aber den Vorteil, dass der Conversation Timeout viel kürzer ist und mit dem gleichen Browser auf der gleichen Anwendung gearbeitet werden kann (solange die Conversation „richtig“ geöffnet wird).

Scopes und Observer

Session Scope und Request Sope enthalten die Daten und es kann leicht zu Dateninkonsistenzen kommen. Die Speicherung von Daten auf dem Server widerspricht der Regel des **Cacheless-Programming,** was keinen Clean Code darstellt.

Java EE enthält im CDI Standard ein Mittel um die überbordende Komplexität, die durch den Server-Status entstehen, besser unter Kontrolle zu bekommen, CDI-Observer.

Im Conversation Scope, erzeugt man sich Observer auf Setter Methode setzen, um Konsistenz zu bewahren. Endlosschleifen verhindern. Mischung von Scopes und wann feuern Observer.

Exception Handling im Allgemeinen

Oft wird in Interviews oder Reviews diese Frage gestellt: „Wie haben Sie das Exception Handling implementiert“. Eine bessere Antwort als: „Exception Handling implementiere ich am liebsten gar nicht“, ist mir dazu bisher nicht eingefallen. Damit will ich nicht sagen, dass Exceptions im Allgemeinen überflüssig sind, ganz im Gegenteil, sondern dass in den meisten Anwendungen Exceptions im besten Fall mit einem eigenen ExceptionWapper nach oben weiter gegeben werden, im schlechtesten Fall einfach weg geschluckt werden, wofür Checked Exceptions sehr anfällig sind. In normalen Web-Anwendungen dürfte der Anteil von RuntimeExceptions gegenüber Checked Exceptions um die 100% liegen. Bei der Entwicklung einer eigenen Programmbibliothek sieht das natürlich anders aus.

Beispiel für Wrapper-Exceptions

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

Zieht jemand an einem Java EE-Server das Netzwerkkabel ab, ist die Festplatte defekt oder bestehen keine Zugriffsrechte auf Verzeichnisse, so handelt es sich aus Java EE-Anwendungssicht um Konfigurations- oder Umgebungsprobleme. Unter diesen Umständen ist eine benamte RuntimeException das richtige Mittel auf den Fehler hinzuweisen, wodurch die Notwendigkeit von Exception Handling, damit Komplexität reduziert und und Fehler vermieden werden.

Exception Handling über JSF ExceptionHandlerWrapper

RuntimeExceptions können in JSF eleganter als über den sehr statischen web.xml Error Page Mechanismus mit einem eigenem ExceptionHandler verarbeitet werden. Dieser kann dann JSF Messages erzeugt, die im aktuellen Context einer Web-Anwendung über <h:messages> angezeigt werden können.

ExceptionHandlerWrapper implementieren

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

Verwendung von Convertern

Converter wandeln eine Menge von Eingabeparameter (z.B. ID) in ein Objekt um. Die wiederkehrende Übersetzungslogik kann so zentralisiert werden. Converter sind in verschiedenen Kontexten wiederverwendbar. Ein Problem ist, dass nicht alle JSF Frameworks Converter immer unterstützen.

Nutzung eines Converters

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

Beispiel für einen Converter

```
@Named
@RequestScoped
public class TestEntityConverter implements Converter {

    @Override
    public Object getAsObject(FacesContext context, UIComponent component
                              , String value) {
        if ((value == null) || value.equals("-") || value.equals("")) {
            return null;
        }
        TestEntity entity = new TestEntity();
        entity.setId(Long.parseLong(value));
        entity.setName("name " + value);
        return entity;
    }

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

View Modularisierung

Ohne Modularisieren sind riesige Application Resource Bundles mit vielen unbenutzten Werten und große XHTML Seiten die Regel.

Begriffsklärung „View Modul“

Ein View Modul ist eine Sammlung von XHTML-Seiten, Property Bundles und Managed Beans, die einen Teil des UIs implementieren.

Da es sich bei View Modulen um etwas ähnliches wie JSF Composite Components handelt, können sie ab JSF 2.0 im Standard, genutzt werden.

Verzeichnisstruktur zur View Modularisierung

```
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
```

Integration von JSF View Modules über pom.xml

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

Probleme ergeben sich daraus, dass IDEs unterschiedlich gut mit modularisierten Views umgehen können.

JSF ist eine Skript-Sprache

JSF bindet Java-Objekte, der Typ der Java-Objekte ist für JSF aber irrelevant. Überflüssige Casts können vermieden werden, sie sind nur für die Autocompletion-Funktion der IDE relevant. Typsichere Schnittstellen müssen nicht verwendet werden. Duck Typing, wie aus anderen Skriptsprachen bekannt, kann verwendet werden.

Verwendung von Bean Validation

Bean Validation ist integriert in JSF. Durch die Verwendung der identischen Validierungslogik in EJBs und JSF, wird Code reduziert und Redundanz entfernt.

Clean Code und XHTML/JSPX

Kleine JSF Fragmente erhöhen die Übersichtlichkeit, große XML-Blöcke sind nur schwer zu verstehen.

Fazit

Komplexität entzsteht bei der Entwicklung von Software von alleine. Da Software-Entwicklung das erzeugen von Code-Zeilen bedeutet, wird ständig Komplexität hinzugefügt. Die Komplexität wieder zu reduzieren ist die Aufgabe eines guten Software-Entwicklers. In diesem Artikel habe ich einige Möglichkeiten erläutert, wie Komplexität in Java EE reduziert werden kann.

Literaturverzeichnis

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

Kontaktadresse:

Alexander Fox
Oracle Deutschland B.V. & Co. KG
Robert-Bosch Straße, 5
D-63303 Dreieich

Telefon:	+49 (0) 6103 397 178
Fax:	+49 (0) 6103 397 397
E-Mail	alexander.fox@oracle.com
Internet:	www.oracle.com/de/index.html