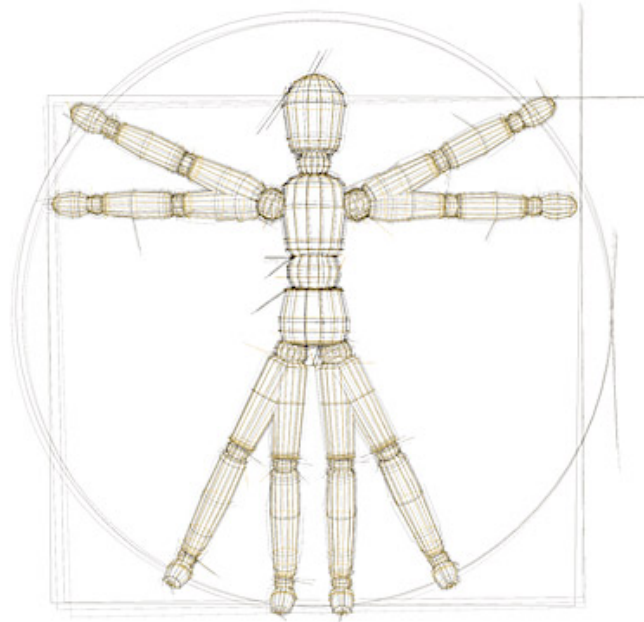


Hinweis:

Diese Druckversion der Lerneinheit stellt aufgrund der Beschaffenheit des Mediums eine im Funktionsumfang stark eingeschränkte Variante des Lernmaterials dar. Um alle Funktionen, insbesondere Animationen und Interaktionen, nutzen zu können, benötigen Sie die On- oder Offlineversion. Die Inhalte sind urheberrechtlich geschützt.
©2018 Beuth Hochschule für Technik Berlin

OOD - Objektorientiertes Design



Lernziele und Überblick



Lernziele

Nachdem Sie die Lerneinheit durchgearbeitet haben, sollten Sie:

- Die Bedeutung der Architektur im Designprozess erklären können
- Eine eigene Vorstellung von den Schritten im Designprozess erlangen und diese auf Projekte anwenden und begründen.
- Die Bedeutung von UML im Designprozess einschätzen und anderen Personen nachvollziehbar erklären können
- Erfahrungen aus dem Design für eigene Projekte sammeln und dokumentieren.



Zeitbedarf und Umfang


Zum Durcharbeiten der Lerneinheit benötigen Sie ca. 90 Minuten und für die Bearbeitung der Übungen ca. 120 Minuten.

Wichtig ist selbst einen Designprozess zu durchlaufen und sich Feedback einzuholen.


Literatur


1. LAHRES, BERNHARD; RAYMAN GREGOR (2009): Objektorientierte Programmierung. Das umfassende Handbuch. Galileo Press, ISBN-13: 978-3-8362-1401-8
Auf der Webseite von Galileo Press ist (Stand 05/2009) eine Leseprobe verfügbar.
2. OESTEREICH, BERND; WESTPHAL, STEFAN (2006): Objektorientierte Softwareentwicklung. Analyse und Design mit UML 2.1. Oldenbourg, ISBN-13: 978-3486579260
3. MCLAUGHLIN, POLLICE, WEST, (2006): Objektorientierte Analyse und Design von Kopf bis Fuß. O'Reilly, ISBN-13: 978-3897214958

Weblinks

 [Aus dem OEP der Teil über Phasen / Entwurf-/Architekturphase](#)

 [Object Oriented Design, Kenneth M. Anderson \(ppt als pdf\)](#)

 [Object Oriented Design, Ian Sommerville 2004 \(ppt als pdf\)](#)

 [Gedanken von Jack Reeves](#)

 [Design Buch von MIT Press mit Beispielen](#)

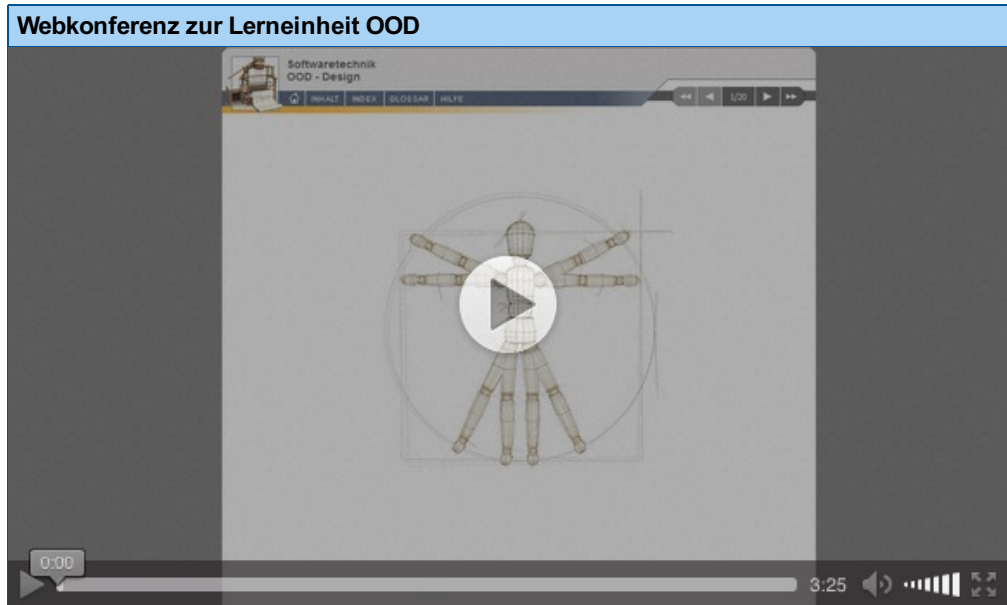
 [Wikipedia](#)

 [SDD \(Description\) von IEEE](#)

 [Exzellentes GUI Design Tool](#)



Film



© Beuth Hochschule Berlin - Dauer: 03:42 Min. - Streaming Media 7.0 MB

Die Hinweise auf klausurrelevante Teile beziehen sich möglicherweise nicht auf Ihren Kurs. Stimmen Sie sich darüber bitte mit ihrer Kursbetreuung ab.


1 Der Designbegriff in der Softwaretechnik



Definition

Software-Design

Unter **(Software-)Design** versteht man den Prozess zwischen Analyse und Implementierung. Es gilt die grundlegende Architektur zu definieren, Komponenten zu spezifizieren und Interaktionen transparent zu machen. Das Design ist oft Grundlage für den Beginn einer konkreten Implementierung.

Im klassischen  Wasserfallprozess (Siehe Anhang) würde auf die Analyse eine lange Designphase folgen, in der mindestens einige Komponenten spezifiziert werden. In einer umfangreichen Analyse werden auch Architekturüberlegungen angeführt und die Zusammenarbeit mit dynamischen UML-Diagrammen erläutert.

Bei agileren Methoden ist dies etwas anders. Hier wird häufig mit einem Prototyp begonnen, der die grundlegende Architektur beinhaltet. Es werden zu Beginn nur wenige Komponenten spezifiziert. Interaktionsmodelle werden zu Beginn nur selten erstellt. D. h. der Spike (Prototyp) repräsentiert einen kleinen Ausschnitt aus dem Design der Software. Durch diesen Prototyp bekommt man schnell Feedback, ob die Architektur so sinnvoll ist oder nicht. Es wird schnell klar, ob die gewählten Komponenten optimal sind und welche Komponenten fehlen. Die Objektinteraktionen sind (hoffentlich) in der konkreten Implementierung sichtbar. Schwächen werden früh aufgedeckt. Der nächste Milestone erweitert diesen Designbereich iterativ.

Die Designphase wird auch **Architekturphase** genannt, weil die Definition bzw. die Gestaltung (Design) der Komponenten eine Architektur ergeben sollte!

Zu Beginn der Designphase - also nach der Analysephase - sollten sämtliche Anwendungsfälle so gut wie möglich bekannt sein. Sie sollten also mit Namen, den auslösenden Ereignissen und resultierenden Ergebnissen dokumentiert sein. Auch bei vielen weiteren Dingen gilt: Man kann nicht alles vorher festlegen, sollte aber einen möglichst guten Überblick haben. Dies gilt für alle Punkte aus der Analysephase, wie z. B. alle nichtfunktionalen Anforderungen.

Dokumente für den Designprozess

Nach  IEEE Std 1016 sind im Designprozess die folgenden Dokumente zu erstellen:

1. Einleitung

- Designübersicht
- Anforderungs-Nachvollziehbarkeits-Matrix

2. System Architektur

- Gewählte Systemarchitektur
- Diskussion alternativer Architekturen
- System Schnittstellen Beschreibung

3. Detailbeschreibung der Komponenten

- Komponente n
- Komponente n+1

4. Benutzerschnittstelle (UI)

5. Zusätzliches Material (Appendix)

Was in dieser Beschreibung nicht explizit erwähnt ist, sind dynamische Zusammenarbeitsmodelle. Diese werden im Folgenden mit einbezogen.

2 Design und Architektur

Gerade für unerfahrene Designer ist es schwierig von einer Komponentensammlung zu einer Architektur zu kommen. Dabei sind zwei klassische Probleme zu lösen:

1. Im klassischen Komponentendiagramm fallen den Designern initial nicht genügend Komponenten ein.
2. Die Komponenten bilden keine Architektur.

Das erste Problem resultiert meistens daraus, dass der Designer noch nicht tief genug in der Gestaltung seines Systems drin ist. Hier hilft oft die parallele Entwicklung eines Prototypen. Bei der konkreten Entwicklung fallen einem dann statt der initialen 10 Komponenten noch sehr viel mehr Komponenten ein.

Erfahrungsaustausch

Auch der Informationsaustausch und die Diskussion mit erfahrenen Designern / Architekten kann bei den beiden Problemen helfen. Erfahrene Designer erkennen schnell welche typischen Komponenten fehlen (z. B. Datenbankzugriff oder Security). Viel wichtiger ist aber, dass Architekten eine Vielzahl an Komponenten in eine leistungsfähige Architektur setzen können. Ein Beispiel dafür ist, die Komponenten in ein 3, 4 oder 5 Schichten-Modell einzusortieren. Beispielsweise ein MVC-Modell zu nehmen und die Komponenten dort zuzuordnen. Alleine dieser scheinbar triviale Schritt hilft bereits Komponenten viel besser zu strukturieren.

Architekturen und Frameworks

Das Designergebnis kann dann auch besser mit Produkten, Frameworks oder den daraus sich ergebenden Architekturen abgeglichen werden. Beispielsweise muss oft eine SOA Architektur realisiert werden. Oder Frameworks wie Spring oder EJB / JEE eingesetzt werden. Diese Architekturen oder Frameworks müssen sich in der Architektur widerspiegeln. In der Regel haben erfahrene Architekten bereits viele (z. B. SOA, JEE, EJB oder Spring)-Projekte mit derartigen Architekturen durchgeführt und berücksichtigen diese sich daraus ergebende nichtfachliche Architektur Anforderung in der Komponentenarchitektur. Viele dieser Frameworks dokumentieren auch die Architektur die sie forcieren (siehe Ruby-on-Rails).

Kriterien

Wie auch für normale Software gibt es Kriterien, die eine gute Softwarearchitektur / Design erfüllen muss:

- **Änderbar:** Dieses Kriterium stellt gewöhnlich das Wichtigste Merkmal in großen Projekten dar. Wie wird sichergestellt, dass Änderungen im System die geringstmöglichen Auswirkungen haben? (Stichwort DI oder Architekturmetriken)
- **Testbar:** Kann ich meine Komponenten gut testen? Dies hängt meist auch mit dem ersten Kriterium zusammen.
- **Verständlich / Lesbar:** Ein schwieriger Punkt, da eine bessere Änderbarkeit des Codes eventuell auch eine schlechtere Lesbarkeit des Codes zur Folge haben kann. Hier muss ein Gleichgewicht gefunden werden.
- **Wiederverwendbarkeit:** Sind Klassen beispielsweise gut gekapselt? Erbringt eine Klasse auch nur einen Dienst? Kümmert sich eine Klasse nur um seine Daten? Hier gibt es eine ganze Menge von Kriterien, die eine gute Wiederverwendbarkeit ermöglichen. Zu beachten ist hier, dass nicht unbedingt Vererbung gemeint ist, sondern Komponenten auch gut per Delegation nützliche Dienste überall erbringen können (anstatt beispielsweise ein rewrite einer Komponente „so ähnlich“).

Wie aber legt man diese Kriterien fest? Können diese vielleicht auch automatisch geprüft und gut visualisiert werden?

Man kann! Es gibt mittlerweile einige gute Werkzeuge am Markt, mit denen - wenn man klein startet - kostenlos Architekturen und Architekturqualitäten festgelegt und überwacht werden können.

Die nachfolgende Abbildung zeigt eine Architektur. Es können hier Regeln definiert werden, beispielsweise das technische Schichten (GUI, Controller, Domain Logic, Data Layer) nur von oben nach unten zugreifen dürfen. Jede dieser technischen Schichten darf aber auf weitere Hilfspakete zugreifen (Common, Util, User, Connection).

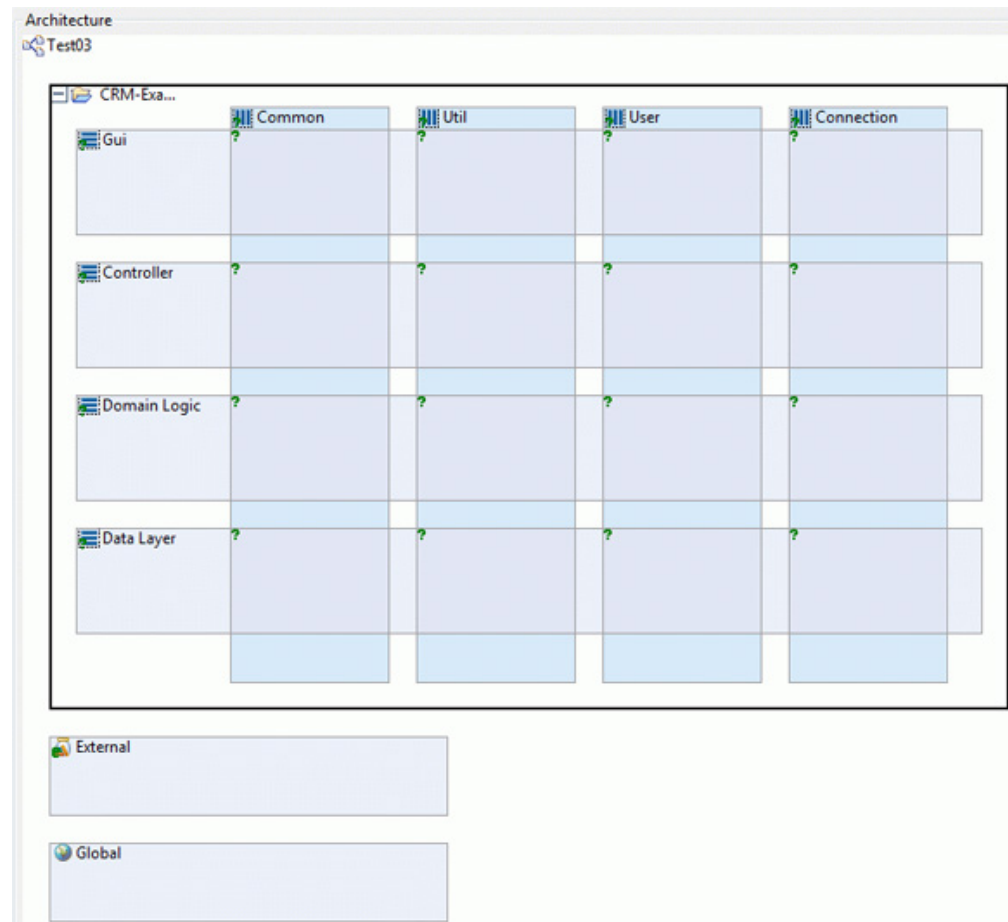


Abb.: Definition einer Architektur

Die folgende Abbildung zeigt die Benutzungshierarchie an. Wenn Paketzugriffe erfolgen, die nicht erlaubt sind (z. B. Bottom-Up Zugriffe oder Zyklen), dann würden diese rot dargestellt werden.

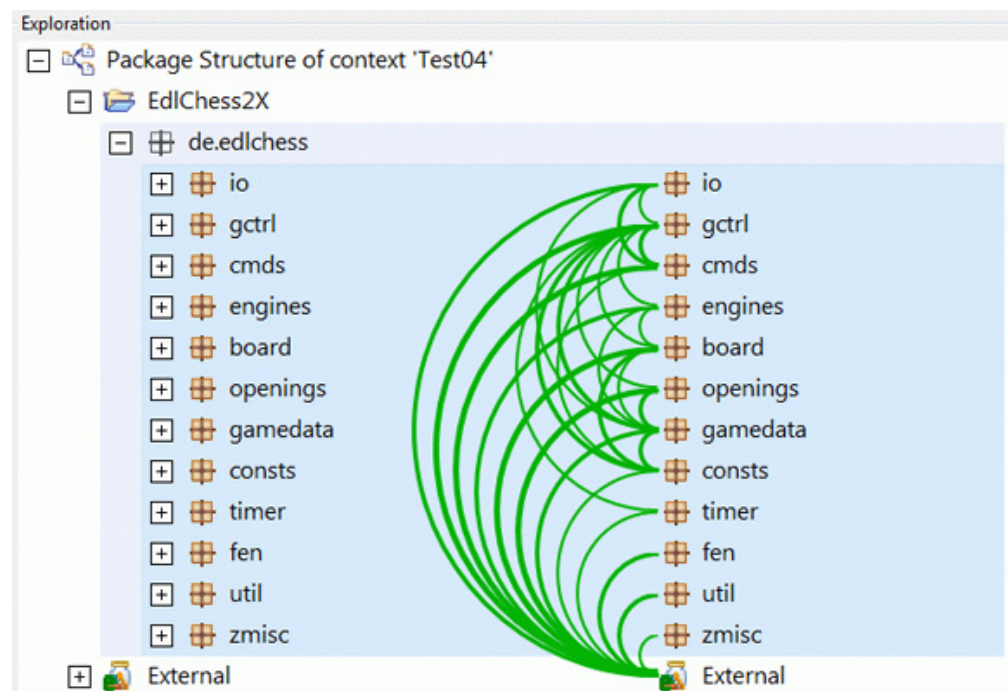


Abb.: Hierarchie

Weitere Werkzeuge

- SonarJ von www.hello2morrow.com
- SotoArc und SotoGraph von www.hello2morrow.com
- Structure 101 Headway Software

Werkzeuge zur Qualitätsanalyse


- Sonar von Codehaus.org
- XRadat, Panopticode, etc.

3 Anwendungsarchitektur

Bei der Definition der Anwendungsarchitektur gilt es zunächst ein grobes Bild der Komponenten zu definieren, welches dann immer weiter verfeinert wird. Daher beginnt man in der Regel auf höchster Ebene mit dem Deployment Diagramm.

Deployment

Ein guter Start ist die Erstellung eines Verteilungsdiagramms (Deployment Diagram).

 Beispiel Toll Collect aus Lerneinheit UML (Siehe Anhang)

Auch beim Deployment-Diagramm gilt: Selbst wenn man denkt, es handelt sich um ein kleines System, fällt einem später ein, dass es doch mehr Interaktionswege oder verschiedene Systeme gibt. In der Regel gibt es also durchaus mehr zu zeichnen bzw. zu entwerfen, als nur einen Server und einen Webclient. Die Schwierigkeit besteht darin, von Anfang an alle möglichen Interaktionswege oder Systeme zu denken und diese entsprechend im Voraus zu berücksichtigen...

Bereits im Deployment-Diagramm können die ersten 5-10 Grob-Komponenten sichtbar werden. Das sollten die Komponenten sein, die danach im Komponenten- (oder Paket-) Diagramm genauer spezifiziert werden.

Schichten und Komponenten

Im Komponentenmodell - dass mit einem Komponentendiagramm beschrieben werden kann - sind in der Regel 15-50 Komponenten sichtbar. Der Designeffekt ist dabei:

- Identifikation möglichst vieler Komponenten. Diese können dann viel leichter gegeneinander abgegrenzt werden und auf Entwickler verteilt werden.
- Über eine Gruppenbildung können Schichten sichtbar werden.

Alle im Diagramm visualisierten Komponenten lassen sich in der Regel gruppieren. Häufig treten beispielsweise die folgenden Gruppen auf:

- GUI-Komponenten / Views
- Controller-Komponenten / Anwendungssteuerung
- Business- / Domain- / Entity-Objekte Komponenten
- Anwendungslogik / - Dienste / - Services
- Datenhaltungs-Komponenten / Datenanbindungen

Man hat also meistens zwei Schnitte durch die Komponenten:

- Die oben genannten vertikalen Schichten, in denen vielleicht sogar über eine kleine Schnittstelle von oben nach unten zugegriffen wird.
- Die fachlichen horizontalen Schnitte innerhalb der Schichten. Es gibt also z. B. in der Schicht der Domain-Objekte Komponenten für Personen, Adressen, Produkte, Lieferanten, etc.

Es ist daher immer ein gutes Vorgehen, alle Komponenten in ein solches Gitter einzutragen (horizontal => Fachlichkeit und vertikal => Schichten). Werden dann noch die Zugriffe mit ihrer Richtung eingezeichnet, so wird Architektur sichtbar und die Sinnhaftigkeit transparent.

Mit dieser Anordnung werden auch leichter fehlende Komponenten erkannt. Beispielsweise:

- Security / Access-Control
- Transfer Objekte
- Logging / Transaktionalität
- User-Management

Die Abbildung zeigt ein Beispiel für eine einfache Schichtenbildung.

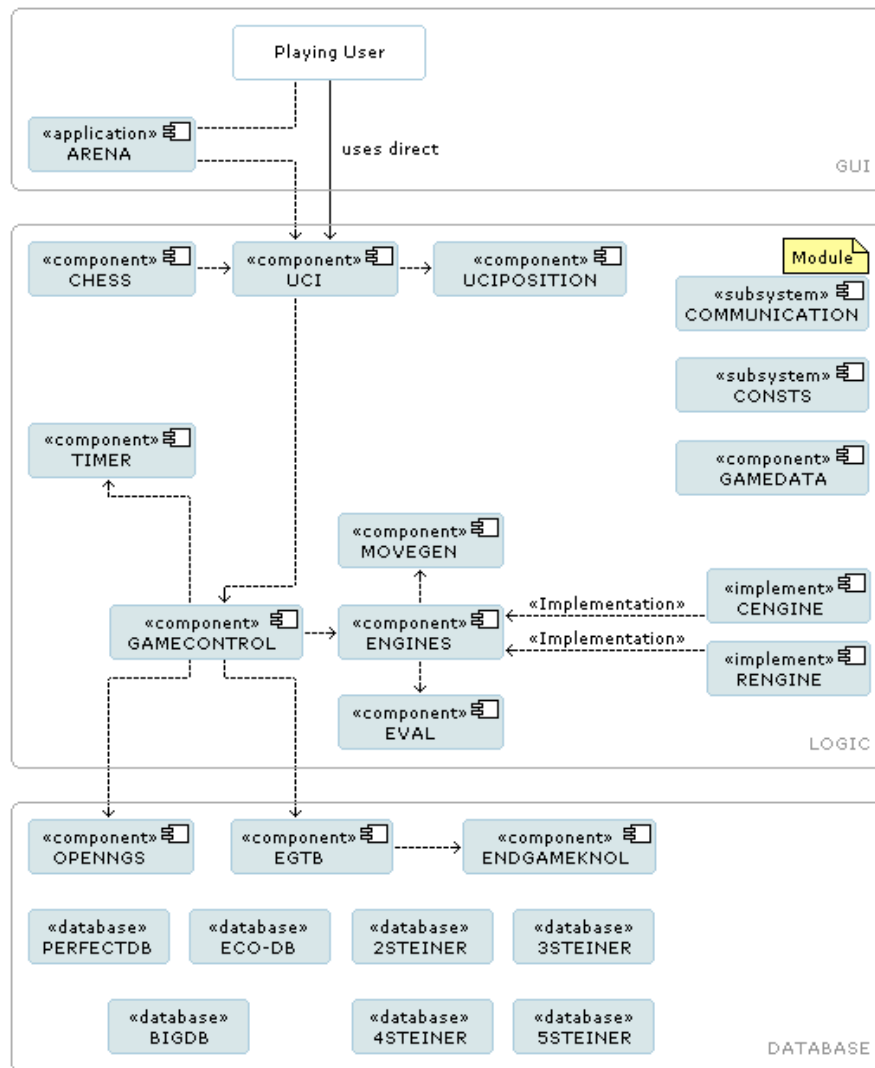


Abb.: Unterteilung in Schichten

In dieser Grafik sind drei Schichten sichtbar. Die GUI-Ebene, eine Datenbank-Ebene und eine mittlere Ebene, in der die Logik enthalten ist. Auch hier ist die Logikschicht nicht optimal spezifiziert. Wichtige Domain-Objekte wie Person, Adresse, Firma, etc. also hier GAMEDATA (die Stammdaten einer Partie) könnten gesondert dargestellt werden. Weiterhin fehlen natürlich Aspekte einer Multiuser-Architektur wie Userverwaltung oder Security, die in anderen realen Anwendungen vorkommen könnten.

Die Rahmenbedingungen müssen ebenfalls an dieser Stelle geprüft werden, da diese das später zu verwendende Framework implizieren. In der Regel sorgen die Komponenten nicht selbst für die Ablaufsteuerung. Das Framework ruft Komponenten nach dem Hollywood Prinzip auf, wenn diese registriert sind.

Im Webframework „Ruby on Rails“ gibt es beispielsweise per Definition vier Orte für Klassen:

- **controllers** - Alle Controller Klassen. Sie implementieren die Ablaufsteuerung der Anwendungsfälle.
- **models** - Die Domain Objekte, also alle Basisdatenstrukturen wie Person. In Rails werden diese automatisch auf Datenbanken abgebildet und der gesamte Datenbankcode automatisch generiert. Mit Scaffolds können sogar sofort Beispiel Views aus diesen Modellen erstellt werden.
- **views** - Alle View-Klassen in Form von HTML-Code die z. B. `<% Prog-Code %>` dynamische Tags enthalten.
- **helpers** - Sonstige Services die implementiert werden müssen und nicht in das obige Schema passen. Hier wird sichtbar, wie gute Frameworks eine Schichtenarchitektur erzwingen und damit auch teilweise sicherstellen, wer mit wem kommuniziert.

Checkliste

- Habe ich ein Deployment Diagramm gezeichnet?
- Gibt es später doch noch mehr Systeme im Deployment Diagramm?
- Sind schon einmal möglichst viele Grobkomponenten im Deployment Diagramm eingetragen?
- Habe ich im Komponenten-Diagramm alle Schichten identifiziert?
- Habe ich im Komponenten-Diagramm ganz viele Komponenten identifiziert und eingetragen?
- Wer soll bzw. darf wen benutzen?

4 Fachliche Komponenten

Definition der Fachklassen

Aus dem Deployment-Diagramm gehen normalerweise die ersten 5-10 sehr groben Komponenten hervor. Diese werden dann im Komponentendiagramm jeweils wieder in Komponenten zerlegt. An dieser Stelle muss überlegt werden, ob dies schon Komponenten auf Klassenebene sind. Das heißt, jede im bisherigen Verlauf angefallene Komponente sollte jetzt bis auf Klassenebene herunter gebrochen werden.



Hinweis

Wir verwenden hier weiter den Sprachgebrauch Klasse als kleinstes Element einer Komponente. Klassen bestehen üblicherweise aus Daten und Methoden. Jedoch sind hier auch im weiteren Sinne Datengruppen (wie Structs) oder reine Module gemeint. Letztere können nur aus Methoden bestehen und keine Daten innehalten. In diesem Sinne sind sowohl Structs, Module und Methoden Komponenten. Wir beschränken uns hier nur auf die Nennung von Klassen. Dennoch sind Komponenten meistens auch größere Einheiten, die z. B. mit Artikeln, Kunden oder Bestellungen zu tun haben können.



Abb.: Fachklassen

Externe Komponenten

Jedes System hat externe Schnittstellen. Dies können GUI, Webservices, externe Datenbanken, etc. sein. Diese sollten früh genug erfasst und mit geeigneten Komponenten dargestellt werden.

Workflows

In einem Softwaresystem gibt es nicht nur Fachklassen, sondern auch Klassen, die für den Ablauf zuständig sind. In kleinen Systemen ist dies üblicherweise eine Controller-Klasse. In größeren Systemen geben üblicherweise die Controller-Klassen Funktionalität an Workflow-Komponenten ab.

Ein Anwender kann so im System beispielsweise seine Adressdaten eingeben. Da dies ein mitunter sehr komplexer Vorgang sein kann - man denke hier nur an Fehlerbehandlung und Fehlerbehebung - gibt es oft Workflow-Komponenten für die entsprechende Ablaufsteuerung. Denken Sie sich beispielsweise für die oben abgebildeten Fachklassen jeweils eine Komponente, die den speziellen Ablauf steuert.

Oft ergibt sich dabei ein Zusammenhang mit den Anwendungsfällen, die sie in der Analyse definiert haben. So könnte die Registrierung einer Person im Webshop genau ein Anwendungsfall sein. Diese Registrierung endet im Erfolgsfall mit einer oder mehreren Adressklassen und die Bearbeitung und Steuerung der Eingaben im GUI könnten genau von ein oder mehreren Workflowkomponenten zu diesem Thema vorgenommen werden.

Gruppen bilden

Die Abbildung zeigt, dass es hilfreich ist, die Komponenten in Gruppen einzuteilen und sich Beziehungen zu überlegen.



Abb.: Einteilung in Gruppen

In vielen Fällen zeigt es sich, dass zusammengehörende Fachklassen auch zusammen behandelt werden können. Zum Beispiel mit einer Workflow-Komponente.

Checkliste

- Wurden alle Komponenten aus dem Komponentendiagramm auf kleinstmögliche Einheiten runtergebrochen?
- Wurden alle externen Komponenten erfasst?
- Wurden alle Workflowkomponenten erfasst?
- Wurden alle sonstigen Logik- und Hilfskomponenten erfasst?
- Sind schon einige Beziehungen erfasst?

5 Komponentenspezifisches Klassenmodell

Im komponentenspezifischen Klassenmodell werden die einzelnen Entitäten genau ausgearbeitet, d. h. spezifiziert. Dazu geht man in einzelnen Schritten vor. In den vorigen Kapiteln wurden die Fachklassen nur grob definiert. Unter Umständen wurden auch keine Assoziationen definiert. Dies muss jetzt alles nachgeholt werden:

1. Initial sollte man ein Lösungskonzept nicht nur für die Fachklassen sondern für alle Klassen definieren. Auch umliegende oder weitergehende Klassen müssen betrachtet werden.
2. Als nächstes sollten alle Assoziationen definiert werden, sofern nicht schon im vorigen Schritt geschehen.
3. Anschließend sollte man sich für jede Klasse überlegen, welche Verantwortlichkeiten sie einnimmt und welche Zustände Sie einnehmen kann.
4. Abschließend sollte die Klasse selbst ausgearbeitet werden.

Gehen wir die Punkte einzeln an Beispielen durch.

1. Identifikation aller endgültigen Fachklassen

In einem System kann vorher eine Komponente Requirements existiert haben. Also ein Modul für die Ermittlung und Verwaltung von Requirements. Diese Komponente ist für ein feineres Design noch viel zu grob und muss weiter aufgeteilt werden. Requirements werden von Personen erstellt. Auch dies muss erfasst werden.

Es gibt also mindestens zwei Komponenten: Person - Requirement. Personen aber müssen eine Adresse haben. Bei Fragen zu einem Requirement muss man den Autor sofort kontaktieren können. Man muss vielleicht auch wissen, in welchem Projekt er derzeit ist und in welchem Projekt er bisher mitgearbeitet hat. Dabei ergibt sich schnell eine Grafik mit vier Komponenten. Adresse - Projekt - Person - Requirement. Generell sieht man immer mehr Klassen, je tiefer man in die genaue Definition hineinschaut

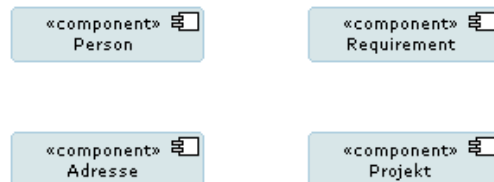


Abb.: Requirements - Komponenten

2. Ermittlung der Beziehungen

Die Analyse der Beziehungen ist nur oberflächlich gesehen einfach. In unserem Beispiel würde eine Person viele Requirements erstellen. Eine Person könnte mehrere Adressen haben und auch im Laufe der Zeit an vielen Projekten teilgenommen haben. Aus letzterem könnte man dann auch das aktuelle Projekt ermitteln.

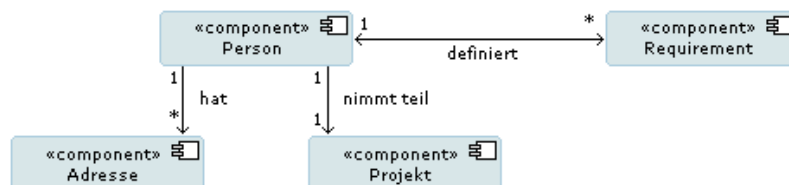


Abb.: Beziehungen

Zwei Dinge sind hier wichtig, die dann dokumentiert werden müssen:

1. Was repräsentieren die Klassen?
2. Welche Zustände können die Klassen annehmen? In unserem Beispiel also:

Person

- Eine Person repräsentiert jemanden, der ein Requirement erstellt hat.
- An Zuständen könnte z. B. vermerkt werden, ob diese Person eine interne Person oder ein externer Berater ist.

Requirement

- Ein Requirement repräsentiert eine konkrete Definition einer Anforderung.
- Requirements können verschiedene Zustände annehmen offen / geschlossen oder funktional / nicht funktional.

Generell kann man sich bei Zuständen natürlich gleich alle Eigenschaften / Attribute überlegen und diese im letzten Schritt dokumentieren.

3. Exakte Komponentenspezifikation

Die exakte Komponentenspezifikation könnte dann textuell alle Attribute und Methoden enthalten. Alternativ könnte auch gleich in UML Notation gearbeitet werden:

```
Person
id : int
name : String
position : {EXTERNAL, INTERNAL}
adressen : collection
projekte : collection
```

```
Requirements
id : int
name : String
datum : Date prio : {LOW, MEDIUM, HIGH, CRITICAL}
status : {OPEN, CLOSED, PENDING, DELETED}
beschreibung : String
docs : collection
```

Mit dieser Information kann der Entwickler - sofern er dies nicht selbst definiert hat - reale Klassen generieren.

Hilfreich wären noch weitere Randbedingungen. So kann `name` zwingend nicht null sein oder `datum` muss vor 01.01.1900 liegen. Hier fallen ihnen als Designer sicherlich noch weitere Möglichkeiten ein.

Checkliste

- Sind alle Klassen definiert worden?
- Ist noch immer ein schlüssiges Designkonzept sichtbar?
- Sind alle Assoziationen definiert worden?
- Sind alle Assoziationsparameter definiert? (Navigierbarkeit, Stereotypen, Rollen, etc.). Gibt es für die Navigierbarkeit ausreichend Schlüssel?
- Passen alle Zustände und Attribute zu der Rolle der Klasse? Stimmen die Verantwortlichkeiten?

6 Designprinzipien

Die spannende Frage im Design ist: Wie kommt man von lauter Komponentenzeichen zu einem guten Komponentendesign?

Wir gehen einmal davon aus, dass der Designer ein guter Architekt ist und entsprechende Lerneinheiten und Bücher gelesen hat. Welche Mittel hat er in der Hand um ein gutes Design zu erschaffen? Dabei wollen wir uns im Folgenden auf die Komponenten und deren Zusammenspiel selbst beschränken. Design ist mehr als die Anordnung der Komponenten (auch GUI-Design oder das „transparent machen“ von Interaktionen gehören dazu). Jedoch sind die Anordnung und die Interaktion der Komponenten das wichtigste Kernmerkmal eines guten Designs.

A. Berücksichtigung des Umfeldes:

Als Designer und Architekt wird man sowohl das Umfeld als auch die Aufgabe, das Team und die Rahmenbedingungen (Standards, Frameworks) berücksichtigen.

B. Einteilung in Pakete:

Als Werkzeug steht dem Designer die Einteilung in Pakete zur Verfügung. Analog der Lerneinheit „ARC - Architektur“ gibt es zwar noch höhere Ebenen, aber diese sollen hier erst einmal nur implizit berücksichtigt werden. Ein Designer wird sich also die Paketstruktur bestmöglich überlegen und in N-Dimensionen aufteilen. Technisch, fachlich oder in anderen Konstellationen zusammengehörig.

Diese Dimensionen lassen sich gut graphisch visualisieren und mindestens zweidimensional in der Paketstruktur darstellen: Eine Dimension in der Verkettung des Namensraumes

`level1.level2...leveln`

(z. B. `de.sun.com.procect.app.persistence.dao`)

und einmal innerhalb eines Namensraumes auf einem Level `level1.level2.`

`[teilA|teilB|teil C...]`

(z. B. `de.swt.app.zinsrechner`; `de.swt.app.rentenrechner`;

`de.swt.app.steuerrechner`; etc.).

Nach Ansicht vieler Architekten trägt die Beachtung von Teil B bereits ein Großteil zu einem guten Design bei:

- Anordnung der Komponenten
- Visualisierung der Komponenten als Definition
- Visualisierung der Komponenten im konkreten Zugriff
- Prüfung von Zugriffsrechten von Paket A auf Paket B oder von Komponente / Klasse A nach B
- Auswertung von Metriken zum Zugriffsverhalten (z. B. mit JDepend oder Sonar)

C. Einteilung in Klassen:

Welche Klassen befinden sich innerhalb der Pakete. Wie werden diese sinnvoll benannt? Sind diese in sich selbst wohl strukturiert?

D. Interaktion der Pakete:

Der Designer klärt die Frage: Wie kommunizieren diese Pakete miteinander. Kann jedes Paket mit jedem kommunizieren? Wird die Kommunikation kanalisiert? Wie in den vorigen Kapitel schon erläutert sollte der Designer / Architekt eine klare Vorstellung haben, welches Paket mit welchem kommunizieren darf.

Ggf. kann auch über definierte Schnittstellen kommuniziert werden, wie z. B. Proxies. Hierfür gibt es - wie schon erwähnt - leistungsfähige Werkzeuge, die dies visualisieren und prüfen können (SonarJ, Soto*, Structure101 oder auch einfache Java Frameworks wie Macker) und genügend Metriken zu diesem Thema auswerfen.

E. Interaktion der Klassen:

Wie kommunizieren die Klassen miteinander? Gemeint ist hier im Wesentlichen die Kommunikation innerhalb der Pakete. Werden Design Patterns verwendet? Wird ein Dependency-Injection Framework verwendet? Hier gibt es eine Vielzahl von Fragestellungen, die später noch vertieft werden und auch Teil der Lerneinheit „Architektur“ sind.

F. Design der Klassen selbst:

Obwohl das innere Design der Klassen selbst schon in ein Coding hineinreicht, kann ein sauberes Coding auch selbst viel zum guten Design beisteuern. Dazu im Folgenden ein paar Prinzipien.

6.1 DRY und SRP Prinzip

DRY - Prinzip

DRY steht für *Don't Repeat Yourself*. Dies ist eines der am meisten verletzten Prinzipien. Gleicher oder ähnlicher Code soll zusammengefasst werden, da er damit wartbarer wird und weniger Fehler entstehen. Dies gilt sowohl für Code (mit Methoden) als auch für Daten (dafür gibt es Normalisierung).

Einer der Top-Fehler in der Softwareentwicklung sind Copy&Paste Fehler, weil Copy&Paste einfach und schnell ist. Leider aber auch extrem fehleranfällig. Moderne Continuous Integration Frameworks oder Code-Analyse-Werkzeuge prüfen Copy&Paste in Anwendungen und warnen. Die Beachtung von DRY in der Designphase schafft ein besseres Design.

SRP - Single Responsibility Design Prinzip

Klassen sollen nur eine Aufgabe übernehmen. Warum? Weil der Code irgendwann auch mal geändert werden muss und Änderungen an einer Klasse sollen nur eine Aufgabe betreffen und nicht noch eine weitere. Trägt eine Klasse z. B. zwei Aufgaben:

1. Verschlüsselung eines Passwortes und
2. Berechnen des Hashwertes eines Passwortes,

so könnten Probleme auftreten.

Wird die Klasse im Projekt an nur einer Stelle im Code verwendet, um ein Passwort zu verschlüsseln und dann umbenannt, so kann es an 100 anderen Stellen zu Problemen kommen: nämlich genau dort, wo die Funktionalität zum Hashen verwendet wird, die u.U. nicht mit dem Passwortverschlüsseln zu tun hat. Die Beachtung von SRP vermindert die Änderungsunempfindlichkeit. Kein SRP führt zu einem höheren Kopplungsgrad und damit auch zu unverständlicherem Code. In einer Teilform wird dieses Prinzip auch das **“Separation of Concerns”** genannt.

Eine Implikation dieses Patterns ist, Komponenten so zu implementieren, dass weniger Unvorhergesehenes eintritt: **The principle of least astonishment**. Die Komponente sollte nur genau das tun, was aus Ihrem Namen (und ggf. der Interfacedokumentation) ableitbar ist.

6.2 Information Hiding (Tell, don't ask)

Wissen sollte an der richtigen Stelle vorhanden sein und nicht hinausgetragen werden. Dies gilt sowohl für Klassen als vielleicht auch für Packages. Schon **getter** in Klassen verletzen auf gewisse Art und Weise das Prinzip des Information Hiding. Die anfragende Klasse B nimmt Werte von A entgegen und vollführt damit Logik, die vielleicht viel besser in A aufgehoben wäre. Nämlich da, wo in A die Daten auch abgelegt worden sind. A sollte in der Regel keine reine Klasse zur Datenhaltung sein, sondern gleich Logik beinhalten. Dann sind die Objekte auch loser gekoppelt und nicht über die Daten miteinander verbunden.

Ausnahmen gibt es natürlich immer. So sind reine Datenobjekte wie DTOs (Data Transfer Objects) nur temporär lebende Objekte die keine Logik brauchen. Dennoch haben auch solche Beispiele Logik, die es erlaubt, Daten aus der Datenbank in das Zielformat zu übertragen.

Außerdem ist es extrem wichtig, dass Design lebendig zu machen und zu halten. Es gibt viele Projekte in denen wird umfassend designed und nach einer kleinen Weile ist das Design veraltet. Dann hat es seinen Sinn verloren und man hätte es sich vielleicht auch sparen können. Ein Design muss sich der Entwicklung anpassen können. Hierbei obliegt es dem Designer, die richtigen Tools zu finden.

Dies können entweder UML-Tools für das Reverse Engineering sein. Genauso interessant sind aber die oben und vorher genannten Werkzeuge für die Architekturanalyse, die Paketstrukturen analysieren, (reverse) visualisieren und durchsetzen (enforcement) können.



Formulieren

Übung OOD-01

Paketstruktur und Kommunikationswege

Beantworten Sie die folgenden Fragen zu Ihrem Projekt und bereiten Sie sich darauf vor, den Aufbau ihres Projektes in einer Diskussion vorzutragen und zu erläutern.

- Wie ist die Paketstruktur in ihrem Projekt?
- Wie kommunizieren die Pakete und Klassen miteinander?
- Haben Sie eine Vorstellung der Kommunikationswege in Ihrem (Komponenten-) Design?
- Können Sie falsche Kommunikationswege (Zyklen oder aufwärts gerichtete Kommunikation) visualisieren oder dann mittels (Unit-) Tests automatisiert Warnungen erzwingen?

Bearbeitungszeit: 20 Minuten

7 Dynamische Modelle

Die dynamischen Modelle eignen sich zum Entwerfen und Modellieren im Wesentlichen aus zwei Gründen:

1. Veranschaulichung von Abläufen für die Teammitglieder
2. Einfachere Überprüfung der Vollständigkeit

Veranschaulichung von Abläufen

Entwicklern fällt es oft schwer, sich in bestehende oder noch zu bauende Systeme hineinzudenken. Meistens gibt es jedoch einen Architekten, der entweder das System kennt oder die Interaktionen in dem zu bauenden System bereits vor Augen hat. Hier helfen dynamische Diagramme durch die Veranschaulichung von Sachverhalten und dem Zusammenspiel der Komponenten - sogenannte Zusammenarbeitsmodelle in Sequenz- oder Kollaborationsdiagrammen.

Einfachere Überprüfung der Vollständigkeit

Aktionen in einem System lassen sich oft Top-Down herunter brechen. Ein Flugbuchungssystem besteht zu Beginn ausschließlich aus dem Ablauf, dass sich der User einloggt und sich einen Flug bucht. Je genauer die Betrachtung, desto mehr sieht man, dass alle Vorgänge und Interaktionen der Komponenten heruntergebrochen werden müssen. Eine Flugbuchung besteht in der Regel aus Einloggen, Suchen, Auswählen, Parameter einstellen, Zahlungsdetails eingeben und Vorgang abschließen. Jeder Einzelne der hier genannten Vorgänge lässt sich in einem realen System wiederum in einzelne Bestandteile eines dynamischen Diagrammes darstellen.



Die Modellierungstiefe hängt hierbei generell von der Projektart und dem Projektbedarf ab. In vielen Projekten werden jedoch die wichtigsten Komponenten und Abläufe eher zu wenig visualisiert. Die Entwickler verlieren das „Big-Picture“ aus dem Auge. Andersherum werden - häufig in der Startphase - zu viele UML-Diagramme erstellt, die dann nicht mehr aktuell gehalten werden können und somit wertlos sind. Daher gilt es das richtige Maß zu finden - zweckmäßige und effiziente Diagramme zu erstellen, die stets aktuell gehalten werden können.

Aktivitätsdiagramme

Aktivitätsdiagramme werden in der Designphase für alle Anwendungsfälle modelliert. In der Regel erst einmal grob und dann werden Teile des Aktivitätsdiagrammes verfeinert. Diese Teile eines Aktivitätsdiagrammes können dann auch Ausgangsbasis für die nachfolgenden Sequenzdiagramme sein, die einen Teilablauf der Anwendung aus Sicht der beteiligten Komponenten modellieren.

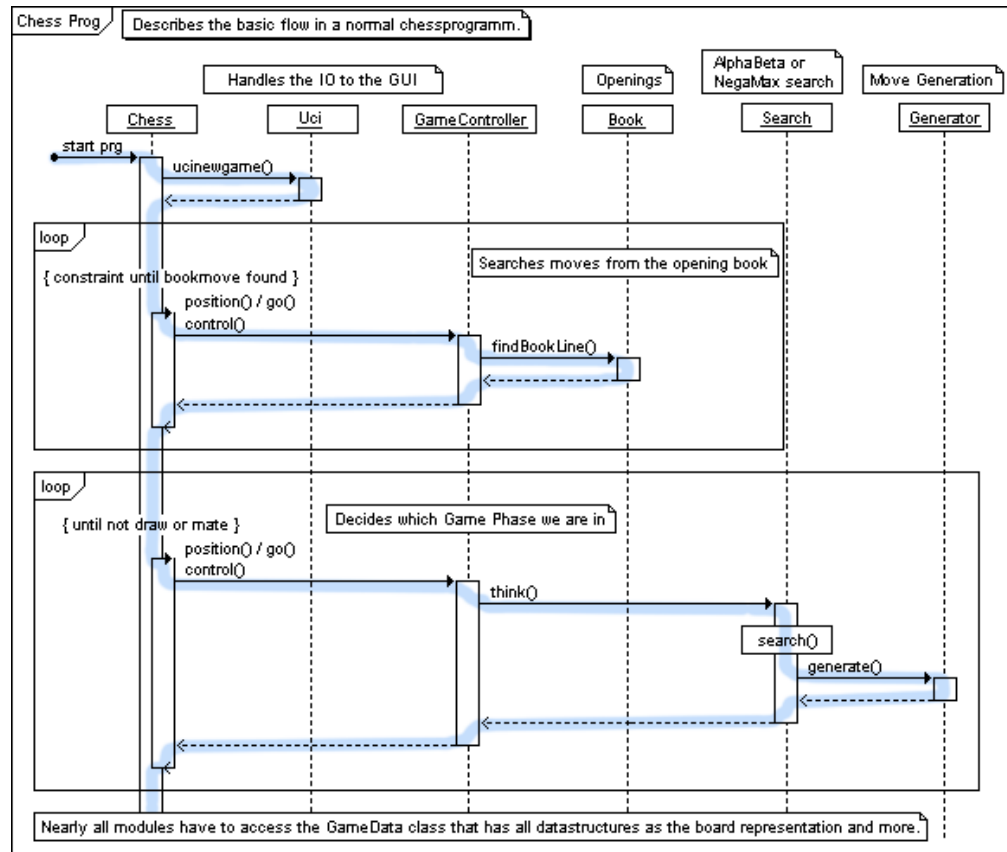


Abb.: Beispiel
Sequenzdiagramm

Interaktionsmodelle

Um den Ablauf zu veranschaulichen, sind **Sequenz-** oder **Interaktionsdiagramme** anzufertigen. Wie bereits in der Lerneinheit zu UML dargestellt wurde, können damit Komponenteninteraktionen veranschaulicht werden - beispielsweise der Kommunikationsablauf im DAO Pattern zwischen den Logik- und den Datenbankkomponenten.

Zustandsmodelle

Im weiteren Verlauf der Modellierung können **Zustandsmodelle** interessant sein. Ein **Zustandsdiagramm** kann beispielsweise die Zustände aus einer Flugbuchung enthalten.

- Anwender unerkannt vs. Anwender bekannt
- Flug unbekannt vs. Flug bekannt / selektiert
- Flug unreserviert vs. Flug reserviert
- Bezahlungsdaten ungeprüft vs. Bezahlungsdaten geprüft
- Bestellung nicht abgeschlossen, Bestellung akzeptiert und übermittelt
- Geld nicht abgebucht, Geld abgebucht

Oft entsprechen diese Punkte Feldern in Komponenten oder sogar Tabellenspalten in den Datenbanken.

8 Schnittstellen und Abhängigkeiten

Schnittstellen sind ein bedeutender Teil eines jeden Softwaresystems. Die Definition einer Schnittstelle legt die Zusammenarbeit zwischen den Teilen des Softwaresystems fest. Ist eine Schnittstelle schlecht oder fehlerhaft definiert, so findet auch eine schlechte und fehlerhafte Interaktion zwischen den Komponenten statt. Es ist daher nicht verwunderlich, dass in größeren Projekten, die Mehrzahl der Arbeit und Fehler in Bereiche investiert wird, die Schnittstellen betreffen.

In der Designphase werden drei Arten von Schnittstellen unterschieden:

1. Komponentenschnittstellen also die Schnittstellen jeder Komponente
2. Interne Schnittstellen zwischen Komponentengruppen
3. Externe Schnittstellen

Auf den folgenden Seiten untersuchen wir alle drei Arten dieser Schnittstellen und Ihre Bedeutung für die Designphase.



8.1 Komponentenschnittstellen

In vielen Programmiersprachen können Schnittstellen explizit formuliert werden. In Java geschieht dies beispielsweise mit **Interfaces**. Diese stellen quasi ein Vertrag zwischen Providerkomponente und Consumerkomponente dar, woraus einige Vorteile resultieren:

Geheimnisprinzip

Der Consumer des Komponentendienstes braucht nur das **Interface** zu kennen und kann sich auf die Erbringung des Dienstes verlassen. Eventuelle zusätzliche **public Komponenten** sind nicht von Interesse: sie dürfen nicht genutzt werden, da sie sich wahrscheinlich deutlich mehr ändern können als Schnittstellenmethoden.

Zuständigkeit

Es liegt mit einem **Interface** eine klare Zuständigkeitsteilung vor. Jeder verlässt sich wechselseitig auf den Kontrakt. Der Consumer vertraut auf Korrektheit, Vollständigkeit und auch darauf, dass sich die Schnittstelle nicht ändert. Der Producer vertraut darauf, dass er sich nur um die Erfüllung der Schnittstelle kümmern muss und um mehr nicht.

Testbarkeit

Komponentenschnittstellen können getestet werden, und da sie in einem guten Projekt immer getestet werden sollten, kann sich der Consumer darauf verlassen, dass der Kontrakt korrekt erfüllt wird. Dem Prinzip der kleinsten Überraschung wird dann Genüge getan. Im Test-Driven Development geht dies soweit, dass sinnvollerweise der Client die Schnittstelle festlegt und auch der erste Consumer ist, noch bevor die eigentliche Komponente geschrieben worden ist. Schnittstellen werden damit von außen nach innen definiert und unnötige Lösungen (YAGNI) haben dann keine Chance, da sie kein Client fordert.

Aus diesem Grunde ist es komplett ratsam, immer gegen **Interfaces** zu programmieren. Selbst im Kleinen und nicht nur zwischen Paketen. Hierbei bedeutet „gegen“, möglichst nie die Klasse zu instanzieren, sondern immer die Klasse aus dem Interface her zu erstellen.

Schauen wir uns zwei Beispiele an:



Beispiel

```
public interface ICmds {
    public void execute();
    public String identify();
}
```

Der Client / Entwickler erkennt, dass es sich um ein Interface handelt (ggf. Name und Schlüsselwort). Es handelt sich also um ein Interface, dass alle Kommandos implementieren muss. Ein Kommando könnte z. B. „Stop“, „Pause“ oder „Neustart“ sein. Der Client kann ein Kommando daher einfach mit `execute` ausführen. Weiterhin wird jedes Kommando garantiert (und mit Tests überprüft) seine Stringrepräsentation mit `identify()` zurückliefern. Auch darauf kann sich der Consumer verlassen.

Was auffällt ist, dass die Schnittstelle angenehm schmal ist. Dies vermindert zusätzliche Abhängigkeiten und spätere Änderungs- oder Anpassungsschwierigkeiten. Schnittstellendesign sollte daher immer dem KISS-Prinzip (*Keep it simple, stupid*) genügen.

In den Paketen von .NET und Java sind viele sinnvolle Schnittstellen zu finden. Ein Beispiel ist das Prinzip des **Iterator Patterns** auf dem Java Package `jav.util`:



Beispiel

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```

Hier hat sich ein Standard für eine immer wiederkehrende Aufgabe etabliert: die Iteration durch Listen mit der Möglichkeit, die Elemente zu erhalten, zu löschen oder auf Listenende zu prüfen.

Ein nicht zu unterschätzender vorteilhafter Faktor ist, dass **Interfaces** die Verwendung von Dependency-Injection Containern, wie z. B. [Spring](#) oder [Google Guice](#), fördern. Auch hier implementiert man gegen ein **Interface**, jedoch ist die konkrete Implementierung und besonders ihr Name entkoppelt. Sowohl der Name, als auch die Implementierung selbst können einfach ausgetauscht werden. Aber auch für viele andere Dinge wie einfache Unit-Tests oder die Verwendung von Mock-Objekten sind Interfaces hilfreich.

8.2 Schnittstellen der Komponentengruppen

Ziel einer guten Architektur ist es auch, die Abhängigkeiten insgesamt zu vermindern. Ein wichtiges Werkzeug dabei ist, die Komponenten zu gruppieren - üblicherweise in Packages - und zu versuchen, die Zugriffe zwischen den Gruppen untereinander zu minimieren. Wie die nachfolgende Grafik zeigt, wäre es nicht wünschenswert, wenn A, B und C aus Gruppe M auf X, Y, Z aus Gruppe N zugreifen:

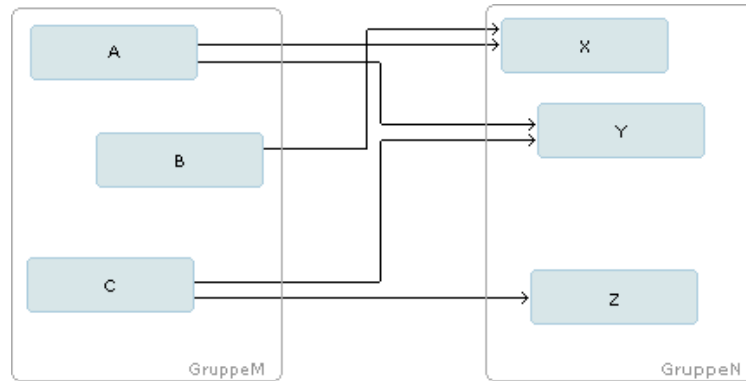


Abb.: Komponentengruppen

Eine Methode die Zugriffe zu vermindern ist daher, für Komponentengruppen/Pakete Proxies einzuführen, die Zugriffe kanalisieren bzw. vermitteln. In diesem Fall ist der Proxy für das Package die Analogie einer public Methode zu einer Klasse. Weiterhin ermöglicht es dem Entwickler von Paket N, die Klassennamen intern zu ändern, ohne das dies Auswirkungen auf die Gruppe M hat, da der Proxy stabil bleibt.

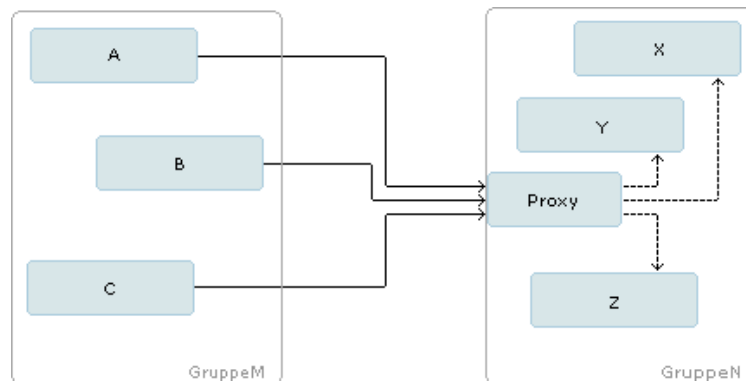


Abb.: Komponentengruppen mit Proxy

Das Ganze nochmal anschaulich an kompletten Packages dargestellt:

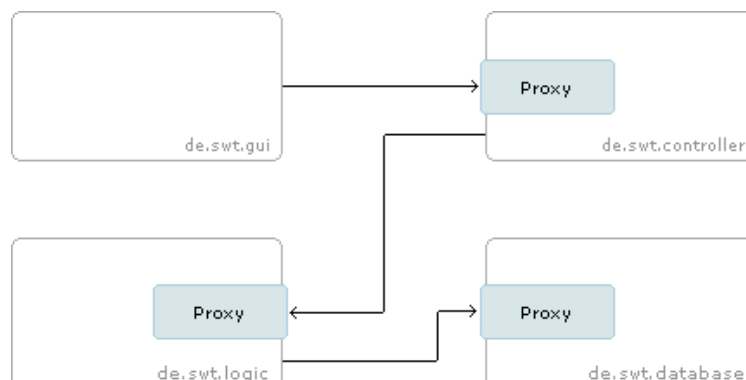


Abb.: Packages mit Proxy

Es ist also immer ratsam wenn sich beispielsweise der Datenbankexperte Tom nicht nur die **Interfaces** seiner Service-Klassen, sondern auch über die **Proxies** seiner **Packages** abstimmt. Damit kann dann Logik-Entwickler Reiner viel schmäler auf die Datenbankebene zugreifen.

8.3 Externe Schnittstellen

In der Designphase ist es wichtig, die externen Schnittstellen genau anzuschauen und diese zu beschreiben. Die meisten großen Systeme interagieren irgendwie mit Ihrer Umwelt. In der Regel denkt man dabei zunächst nur an ein GUI aber es gibt meistens noch mehr Schnittstellen wie zum Beispiel:

- Dokumente / (gescannte) Briefe
- XML aus anderen Quellen
- Legacy Systeme

Überprüfen Sie ihr System also dahingehend, ob es diesbezügliche Quellen geben könnte. Diese in der Designphase von Anfang an im Blick zu haben hilft später sehr. Dabei gilt es, die Datenquellen und deren Verhalten und Erwartungen genau zu beschreiben. In der Regel machen die nichtfunktionalen Randbedingungen der Schnittstellen (Format, Zeitverhalten, Schnittstelle, etc.) die größten Probleme.

Fazit

Welche Diagramme können nun für die Schnittstellenbeschreibung verwendet werden? Vorausgesetzt man will Zusammenhänge / Abhängigkeiten transparent machen und nicht die Schnittstelle selbst in einem UML-Klassendiagramm entwerfen, gibt es eigentlich nur zwei die dafür in Frage kommen:

1. Das Komponentendiagramm (siehe auch Lerneinheit UML)
2. Das Kompositionsstrukturdiagramm



Beispiel

Schauen Sie sich als Beispiel dazu das Kompositionsstrukturdiagramm einer Stereoanlage bzw. Radios an:

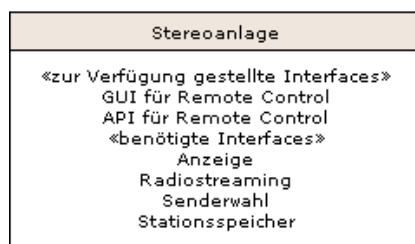
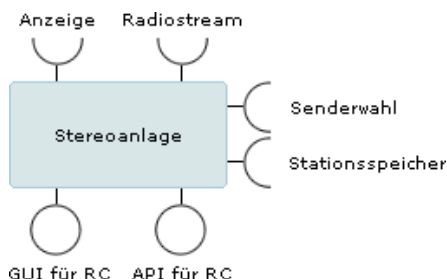


Abb.: Kompositionsstrukturdiagramm einer Stereoanlage

Derartige UML-Grafiken lassen sich sowohl im Allgemeinen, als auch für die **Package Proxies** gut verwenden und machen die Abhängigkeiten und Services transparent.

9 Testdefinitionen

An dieser Stelle erfolgt ein kurzer Vorgriff auf die Lerneinheit: „Objektorientiertes Testen und Test-Driven Development“.

In dieser Phase des Designs sind die Anforderungen an die Komponenten bereits relativ klar. Es liegen zudem die Anwendungsfälle vor, die den Ablauf der Anwendung in Bezug auf definierte Anforderungen beschreiben. Entwickler könnten nun beginnen und versuchen diese Komponenten entsprechend der Spezifikation zu entwickeln. Die Frage ist: Wer prüft danach das Ergebnis? Der Projektleiter, ein Kollege, der Auftraggeber oder eine Maschine? Anhand dieser Fragestellung zeigt sich, dass das Qualitätskonzept vor der Implementierung vorliegen muss. Es ist ja bereits in der Analyse definiert worden.

Das bedeutet allerdings auch, dass die Überprüfung der Ergebnisse sogar vor der Entwicklung der Komponenten ein Thema sein muss. Genau dies ist auch die Thematik des Test-Driven Development, das sich vielfach bewährt hat.

Bei den Testdefinitionen gibt es, wie bereits angesprochen, viele Ebenen auf denen man Tests auch in der Designphase vorbereiten muss.

Tests für Anwendungsfälle

Zu jedem Anwendungsfall gilt es Tests zu entwickeln, um die Abläufe zu testen. Dies gilt sowohl für die Daten, als auch für das zu erwartende Ergebnis.

- A. Für jeden Anwendungsfall muss feststehen, welche Eingangsgrößen vorliegen und welches Ergebnis nach den Operationen des Anwenders erwartet wird. Beispielsweise muss bei einer Bestellung mit Eingabe eines Rabattcoupons der richtige Preis mit Mehrwertsteuer vorliegen. Es gibt viele Anwendungsfälle, wo dies manuell geprüft werden kann. D. h. die Abläufe müssen idealerweise vom Auftraggeber geprüft werden.
- B. Im obigen Beispiel können unter Umständen jedoch auch die Ergebnisse von ganzen Komponentenabläufen getestet werden. Wenn Komponente A für die Berechnung eines Ergebnisses (für Anwendungsfall X) die Komponente B und C verwendet und C noch D verwendet, dann wird hier das Endergebnis getestet, das die Komponente A liefert. Dies steht im Gegensatz zu Unit Test, wo die Komponente A möglichst isoliert getestet wird, d. h. B und C werden durch Mock-Objekte ersetzt!

Diese Fälle sollten schon zu Beginn dokumentiert werden und entsprechende Tests vorbereitet oder auch schon implementiert werden - dies wäre dann ein früher Teil der Implementierungsphase.

Klassentests

Im Gegensatz zu den vorigen Tests beziehen sich Klassentests nur auf den Test der Methoden. Daher spricht man auch von Unit Tests. Es sind hier also Tests für jede Methode zu entwerfen - die dann später in der Implementierungsphase gebaut werden können. Dabei werden die Methoden - wie bereits erwähnt - isoliert getestet. D. h. eventuelle Abhängigkeiten zu anderen Komponenten sollten durch Mock-Objekte ersetzt werden.

Wurde gut analysiert und entworfen, dann wurden für die Methoden der Klassen auch Randbedingungen angegeben. Nehmen wir ein Beispiel.



Beispiel

```
/* ... Randbedingungen... */

class Person {
    ...
    /* ... Randbedingungen... */
    void setAge(int aAge) {...}
    ...
}
```

Hier könnten beim Design schon Randbedingungen angegeben worden sein. Wie z. B. das ein Alter nicht kleiner Null sein darf. Dieses könnte im Testplan für alle Methoden einer Klasse berücksichtigt werden. Im Übrigen werden hier dann auch alle weiteren Tests wie Grenzwerttests etc. geplant. Das Thema wird in der Lerneinheit „Objektorientiertes Testen und Test-Driven Development“ behandelt.

Tests für externe Schnittstellen

Externe Schnittstellen sind in allen Anwendungen immer besonders kritisch zu sehen. Dies liegt daran, weil man diese Schnittstellen immer unterschätzt und sie nicht unter Kontrolle hat. Ein Beispiel:



Beispiel

Call-Center Anwendung

Es soll eine Call-Center Anwendung entwickelt werden, die es ermöglicht, die Telefonmitarbeiter auf beliebigen Rechnern zu unterstützen. Gleichzeitig müssen aber bestehende Telefonanlagen unterstützt werden. So muss das System in der Lage sein, Namen, Telefonnummern und Ereignisse aus dem Telefonsystem zu holen / zu bekommen. Dabei wird davon ausgegangen, dass die Daten immer verfügbar sind, richtig übermittelt werden und keine Seiteneffekte auftreten können.

Leider ist dies in den seltensten Fällen der Fall. Tests für externe Schnittstellen sind meistens kleine Prototypen, die früh Schwachstellen aufdecken können. Diese Tests früh zu definieren und durchzuführen verhindert daher Fehlinvestitionen im Kernbereich, weil eine falsche Schnittstelle beispielsweise nicht verändert werden kann.

Klasseninterne Tests vorbereiten

Bei der Definition der Klassen ist es wichtig neben der Signaturdefinition aller Methoden und Felder - wie oben erwähnt - alle Randbedingungen anzugeben. Wenn diese vorhanden sind, kann man davon ausgehen, dass sie bereits in den Klassentests berücksichtigt wurden. Allein das Wissen, dass auf derartige Randbedingungen getestet wird, sollte den Designer und den Entwickler dazu bringen, klasseninterne Tests gleich zu berücksichtigen.

Der Designer hat daher bereits in der Klasse vermerkt, dass das Alter nicht kleiner als null sein darf. Weiterhin wurde ein Test dafür vorbereitet. Der Test überprüft in diesem Fall, ob eine Exception korrekt geworfen wurde. Der Designer oder Entwickler muss also intern darauf vorbereitet sein eine Exception zu werfen, indem dies entweder in der Designphase notiert oder später implementiert wird.



Beispiel

```
void setAge(int aAge) {
    /* Assertion für age < 0 */
    assert aAge >= 0; /* Diese Zeile gleich oder später */
    this.age = aAge;
}
```

Sun weist darauf hin, dass für die Prüfung der Parameter besser RuntimeExceptions geworfen werden sollten! Der Code ist daher nur zur Demonstration der Assertion.

Es ist lohnend sich frühzeitig Gedanken über klasseninterne Prüfungen zu machen beispielsweise:

- assertions für Vorbedingungen
- assertions für Invarianten, d. h. für Zustände in dem Code

- assertions für Nachbedingungen

Ein Beispiel für den letzten Fall ist die Prüfung des Rückgabewertes. Viele Fehler entstehen dadurch, dass „null“ zurückgeliefert wird oder das Objekt nicht den richtigen Wertebereich hat. Eine Prüfung mittels **assert** ist trivial zu integrieren und hilft, dass die Komponente zumindest im Rückgabewert ihren Vertrag erfüllt.

Testautomatisierung vorbereiten

Sofern es sich nicht um Ablauftests handelt, die keine maschinellen Tests erlauben, sollten die Tests so weit wie möglich automatisierbar sein. D. h. in ein System eingebunden sein, welches automatisiertes Testen ermöglicht. Dies kann im einfachsten Fall eine Buildsprache sein (Ant, Maven, Rake, Gradle, etc.), im besten Falle ein Continuous Integration System wie Hudson, Cruise Control, Bamboo oder das Hosting System selbst (sf.net), welches die Tests eigenständig anstößt.

Um diese Denkweise zu fördern, hat KENT BECK im Jahre 2008 eine JUnit-Erweiterung namens JUnit Max geschrieben, die bei jedem Speichern alle Tests ausführt. So ist sichergestellt, dass nur korrekter Code eingereicht werden kann.

Fazit

Es zeigt sich, dass auch in der Designphase das Qualitätskonzept umgesetzt werden muss und dabei alle Qualitätsebenen angesprochen werden sollten. Die Lerneinheit „Testen“ wird dabei zeigen, dass es noch mehr Testebenen gibt als die bisher Angesprochenen, wie beispielsweise Lasttests oder Integrationstests.

Checkliste

- Sind alle Anwendungsfälle mit Tests abgedeckt?
- Welche High-Level Tests für Anwendungsfälle müssen manuell durchgeführt werden und welche können automatisiert vorgenommen werden?
- Sind alle Klassentests vorbereitet?
- Sind alle Schnittstellen erfasst und dafür Tests geplant?
- Ist sichergestellt, dass auch die Klassen / Methoden selbst intern genügend Testen, um ihren Vertrag sicher zu erfüllen?
- Können alle Tests mit einem Klick gestartet oder zeitgesteuert angestoßen werden?

10 Attribute

Am Ende der Designphase können die Attribute der Komponenten / Klassen definiert werden. Dabei gilt es die folgenden Dinge zu berücksichtigen:

- A. Sind alle Attribute erfasst?
- B. Ist das vorliegende Attribut ein Grundtyp oder besser doch ein Objekt einer anderen Klasse? Die Frage geht dahin, ob das Klassendesign wirklich ausgewogen ist oder nicht. Ist beispielsweise ein Feld Adresse in einer Person ein String mit diesem Inhalt:

```
"Wilhelmstr.25, 16356 Augsburg, Deutschland"
```

oder sollte die Adresse doch besser ein Objekt sein, so dass das Objekt Person 1:1 oder 1:n auf die Adresse verweist? Bei einer Assoziation müssen natürlich nochmals alle Kardinalitäten überprüft werden.

- C. Ist das Attribut vielleicht ein Schlüssel oder eine Collection, d. h. eine Liste, ein Set oder ein Array von Werten? Wenn letzteres zutrifft, welche Collection? In vielen Anwendungsbereichen sind bestimmte Collections zwingend vorgeschrieben oder besser / schneller als andere.
- D. Sind die Zuständigkeiten der Attribute klar? Oftmals werden Attribute definiert an denen plötzlich eine andere Klasse viel mehr interessiert ist. In der Lerneinheit „Refactoring“ wird hier auch von „Neid“ gesprochen. Ziel ist es daher zu überdenken, ob das entsprechende Attribut nicht gleich besser in der Klasse aufgehoben ist, die sich ständig dafür interessiert und nachfragt.
- E. Und natürlich die Frage, ob es Zusicherungen für jedes Attribut gibt. Dies ist auch im nächsten Teil des GUI-Designs enthalten und betrifft oftmals auch das GUI ganz direkt. Also jede Art von Format oder Wertebereichsbedingungen sind hierbei wichtig.

Natürlich sind dabei immer Standards des Unternehmens und des Frameworks zu berücksichtigen. So kann beispielsweise die Arbeit mit Hibernate Annotations (Java Annotations für das Persistenzframework Hibernate) für die Persistenz dazu führen, dass von vorneherein Schlüsselfelder in der Klasse eingeführt werden müssen, die für die Navigierbarkeit der Objekte sorgen.

Schauen wir uns nochmals die Attributdefinitionen aus dem komponentenspezifischen Klassenmodell an.

```
Person
id : int
name : String
position : {EXTERNAL, INTERNAL}
adressen : collection
projekte : collection
```

```
Requirements
id : int
name : String
datum : Date
prio : {LOW, MEDIUM, HIGH, CRITICAL}
status : {OPEN, CLOSED, PENDING, DELETED}
beschreibung : String
docs : collection
```

Könnte in den folgenden Code übersetzt werden:

```
/** ... comments */<br>public class Person {
    int id;                /* Kein Defaultwert, immer > 0,
                           reicht Raum aus oder besser long? */
    String name;           /* Darf nicht null oder leer sein */
    String position;       /* Kann nur EXTERNAL oder INTERNAL sein.
                           Evtl. Enum */
    List <Adressen> adressen; /* Muss mindestens eine Adresse enthalten */
    List <Projekte> projekte; /* Kann auch leer sein */
}
```

```
public class Requirements {
    int id;                /* Kein Defaultwert, immer > 0,
                           reicht Raum aus oder besser long? */
    String name;           /* Darf nicht null oder leer sein */
    Date date;             /* Datum des ersten Anlegens des Requirements.
                           Nach 01.01.2000. */
    Enum priority; /* Kann nur LOW, MEDIUM, HIGH, CRITICAL sein. */
    Enum status; /* Kann nur OPEN, CLOSED, PENDING, DELETED sein. */
    Beschreibung String /* Ausführlicher Text > 20 chars.
                           Muss ungleich dem Namen sein! */
    String docs ; /* Pfad zur anhängenden Datei. Muss korrekte
                  Pfadsyntax haben und validierbar sein. */
}
```

11 Design der Anwenderdialoge (GUI)

Abschließend müssen nun noch die Dialoge für die Anwender gestaltet werden.

Die Abbildung zeigt ein Beispiel, wie das für eine Anwendung aussehen könnte, die ein Projektmanagement und ein Requirements Management beinhaltet.

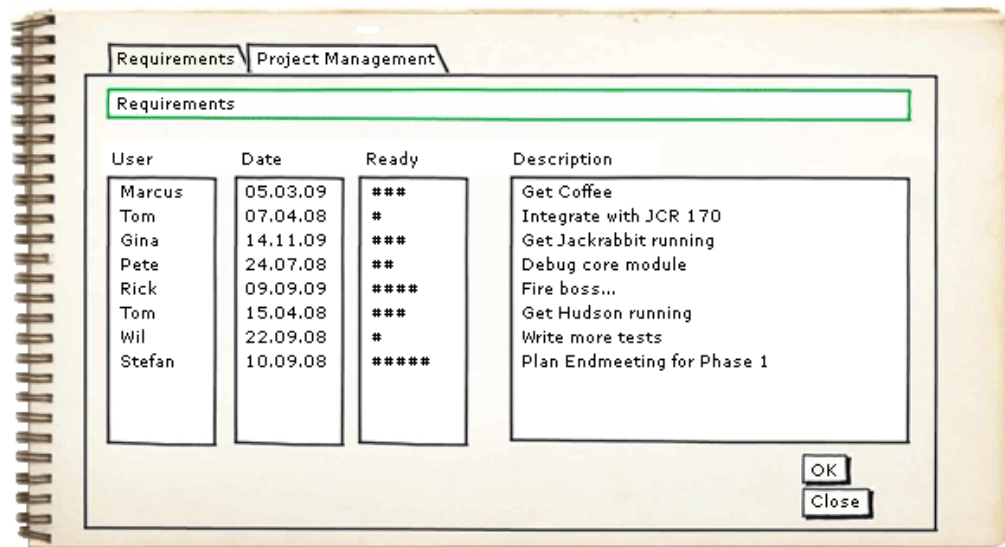



Abb.: GUI für
Beispielanwendung

Was fällt ihnen dabei auf? Was könnte man verbessern? Was fehlt? Machen Sie sich selber kurz Gedanken bevor Sie sich die Hinweis anschauen!

 Was man verbessern könnte! (Siehe Anhang)

Bei der Gestaltung sollte immer beachtet werden, welche Geräte der Anwender überhaupt bedienen soll. In der Regel haben umfassende Anwendungen mindestens drei Eingabemöglichkeiten:

1. Ein Rich-Client auf dem Desktop. Dies kann z. B. eine Java-Swing Anwendung sein. Die komplette Logik wird dazu auf dem Desktop-Rechner installiert.
2. Ein Thin-Client. D. h. eine Webanwendung für alle Anwender, die per Web-Browser auf Ihre Anwendung zugreifen möchten.
3. Ein mobiler Client für den Zugriff per Handy. Die Verbreitung wird hier immer weiter steigen:
 - iPhones werden immer leichter zu programmieren.
 - Unter Google Android ist normale Java Programmierung möglich
 - Die Industrie arbeitet an einem ganz normalen offenen UNIX für mobile Endgeräte, etc.

Was ist noch zu beachten?

- Natürlich sollte im Idealfall Zeit sein, um einen Prototypen (Spike) zu erstellen und diesen zu evaluieren. Also am besten den Anwendern eine Aufgabe stellen und die Arbeit mit dem Werkzeug filmen und Zeiten messen. Die Auswertung aus einer derartigen Analyse ist immer das beste Feedback.
- GUIs sollten immer mehrere leicht sichtbare Abbruchmöglichkeiten haben und es ermöglichen eine Ebene höher zu gehen.
- Es ist immer eine gute Idee, die Navigation sichtbar zu machen. An welcher Stelle befinde ich mich (im Baum)?
- Es gilt zu prüfen: Sind alle Anwendungsfälle im GUI sichtbar? Und natürlich auch umgekehrt: Sind alle GUI-Elemente Anwendungsfällen zugeordnet?
- Auf kurze Erreichbarkeit der Wege sollte geachtet werden. Gruppieren und dies auch optisch anzeigen ist immer eine gute Idee.
- Auf konsistente d. h. einheitliche Darstellung aller Masken muss geachtet werden. Am Ende alle Masken übereinander legen und vergleichen.
- Dialoge werden in unterschiedlichen Kontexten verwendet. Dies muss mit berücksichtigt werden. Also z. B. von Personen mit komplett verschiedenen Rollen oder z. B. eine Reisebuchung mit oder ohne vorherigem Login.
- Es hat sich bewährt, schon die Evolution des GUIs von vornherein mit einzubeziehen. Beispielsweise durch einen Advanced Mode oder später dazuschaltbare GUIs.
- Sehr wichtig sind auch Randbedingungen für jedes Feld. Beispielsweise:
 1. Defaultwerte: Welche Werte haben die GUI-Elementen initial?
 2. Grenzwerte: Welche Grenzwerte sind überhaupt zugelassen? -80000 € für eine Spende? 300 Jahre für ein Alter?
 3. Validierungsregeln: Zu fast jedem Feld gehören Validierungsregeln! Idealerweise werden die schon im UML-Tool spezifiziert, damit diese für MDA zur Verfügung stehen. Was ist z. B. mit Regeln für eine E-Mail-Validierung wie: `%r{^(?:[_a-z0-9-]+)(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-zA-Z0-9-]+\.)*(\.[a-z]{2,4})$}i?`



Hinweis

Suchen Sie sich für die Gestaltung der Anwenderdialoge ein GUI Design Tools aus.
Tipp: [www.balsamiq-mockups](http://www.balsamiq-mockups.com) (ist auch online für die Lehre nutzbar).

Zusammenfassung

Auch in der Designphase arbeitet man vom Allgemeinen zum Speziellen und spezifiziert immer genauer. Als Designer sollten Sie sich selbst eine Vorgehensweise zurechtlegen die z. B. aus den bisher aufgeführten Elementen bestehen kann:

- Entwurf der Architektur
- Definition und Anordnung der Komponenten
- Entwurf der Klassen
- Transparent machen der Interaktionen der Klassen. Definition der Zusammenarbeitsmodelle
- Analyse der Schnittstellen und Abhängigkeiten untereinander. Ggf. Verminderung dieser.
- Entwurf und Definition der Qualitätsvorgaben und der Tests in allen Ebenen
- Design der GUI Dialoge

Dabei stellt sich immer wieder die Frage, wie viel Design überhaupt nötig ist. Manche agile Modelle (wie XP) behaupten komplett ohne Design auszukommen und das Design lieber iterativ zu entwickeln - was unter Umständen einen höheren Refactoringaufwand zur Folge haben kann. Als Designer muss man also ein Gleichgewicht finden zwischen den beiden Polen:

A. Kein Design, lieber iterative Designentwicklung direkt am Code:

Vorteile: Kein initialer Papieraufwand nötig. Anwendung kann sich besser auf Situationen / Designs / Architekturen ausrichten, die erst bei der Entwicklung aufkommen (können).

Nachteile: Falls es offensichtlich gute Designs / Architekturen gibt, kann man diese auch mit einem nicht so großem Designaufwand schneller erreichen. Man verschwendet also Zeit im ausprobieren, wo vielleicht „Best-Practices“ offensichtlich sind.

B. Viel Design, keine Versuchsentwicklung.

Vorteil: Schneller einen definierten Stand erreichen, ohne vielleicht genau zu wissen ob dieser Stand gut ist. Keine Zeit mit live-Erfahrungen verbringen.

Nachteil: Viel Papierarbeit vorher. Vielleicht wird mit dem vorliegenden Design eine Lösung angesteuert die nicht optimal ist, sondern die eine Person favorisiert.

Fazit:

Was Sie als Designer wählen hängt sicherlich wieder davon ab, wie die Aufgabenstellung lautet, wie fit das Team ist und ob es Vorgaben gibt. Oftmals gibt es auch bereits Referenzimplementierungen oder Referenzerfahrungen. Sollte es diese nicht geben, ist in der Regel ein Mix aus beiden Varianten ideal. Dabei kann man auch ruhig in Betracht ziehen, Dinge nicht komplett auszuspezifizieren. Im Englischen spricht man hier von „Perfect Design“ und einem „Good Enough Design“.

Sie sind am Ende dieser Lerneinheit angelangt. Auf der folgenden Seiten finden Sie noch die Übungen und die Einsendeaufgaben.

Übung und Einsendeaufgaben



Formulieren

Übung OOD-02

Design, oder kein Design?

Wieviel Design ist überhaupt nötig? Manche agile Modelle behaupten komplett ohne Design auszukommen und das Design lieber iterativ zu entwickeln. Als Designer findet man entweder ein Gleichgewicht zwischen den beiden Polen oder einen festen Standpunkt.

Bereiten Sie sich auf eine Diskussion vor, die entweder im Audiochat oder in der Präsenzveranstaltung stattfinden wird. Notieren Sie sich Stichworte zu den Fragen.

- Wie würden Sie die Thematik aus Ihrer Sicht schildern?
- Würden Sie eher mehr oder weniger Designen? Begründen Sie Ihre Entscheidung.
- Was und wieviel ist ein Good Enough Design?

Bearbeitungszeit: 20 Minuten

Bevor Sie mit der Bearbeitung der Einsendeaufgabe beginnen, besprechen Sie bitte mit Ihrer Modulbetreuung in welcher Form die Aufgaben bearbeitet werden sollen.



Einsendeaufgabe

Einsendeaufgabe OOD-E1

Designentwurf

Führen Sie für ihr durchgehendes Softwareprojekt ein Design durch. Dokumentieren Sie alle Schritte. Bringen Sie das Design dann ausgedruckt oder als PDF zur Präsenz mit und besprechen Sie das Design mit dem Dozenten.

Bearbeitungszeit: 45 Minuten



Einsendeaufgabe

Einsendeaufgabe OOD-E2

Redesign

Versuchen Sie ein bestehendes Projekt von Ihnen zu Redesignen oder nach den hier genannten Designvorgehensweisen zu analysieren und zu verbessern.

Bearbeitungszeit: 45 Minuten



Einsendeaufgabe

Einsendeaufgabe OOD-E3

GUI - Design

Designen Sie ein GUI für ein beliebiges Projekt.

Bearbeitungszeit: 20 Minuten

Appendix

Wasserfallmodell

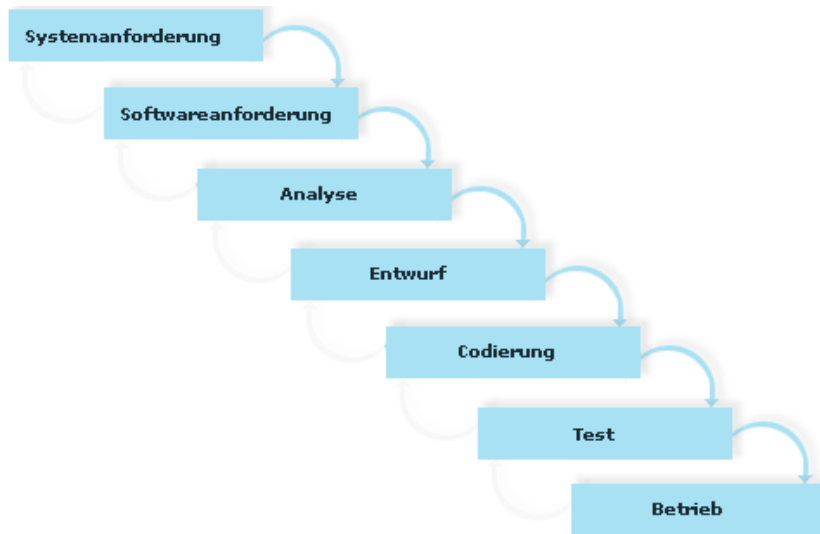


Abb.: Starrer Ablauf im Wasserfallmodell

Beispiel für ein Verteilungsdiagramm (Deployment Diagram)

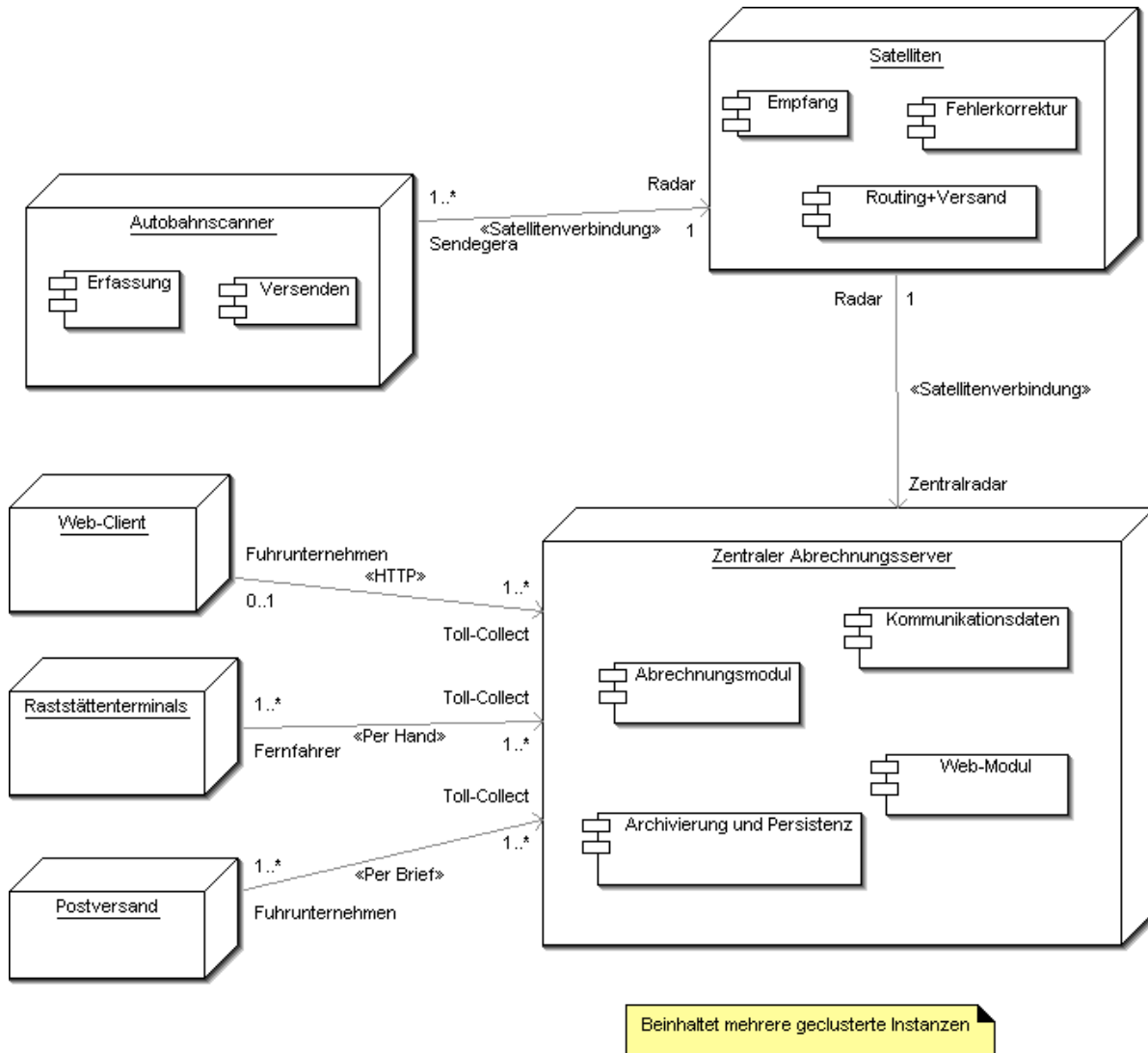
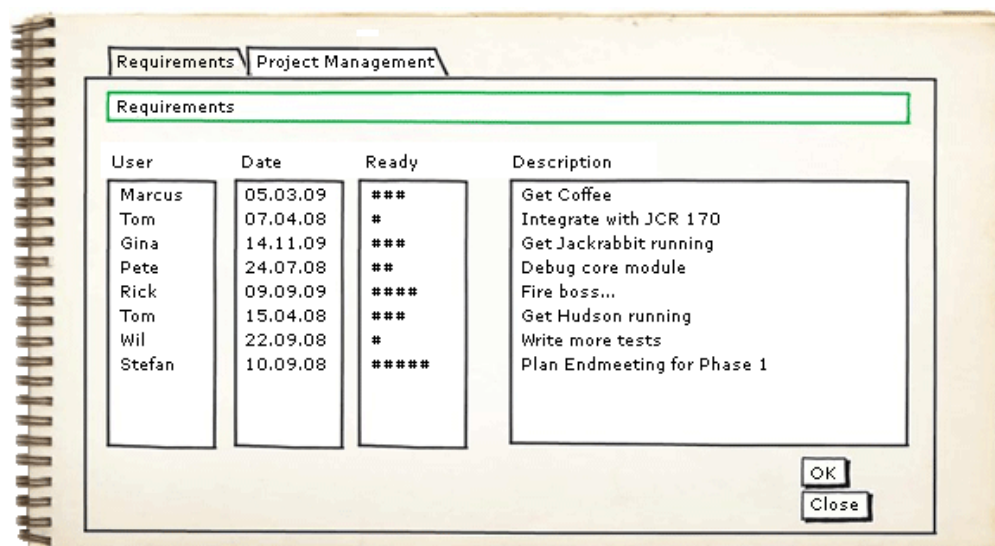


Abb.: Verteilungsdiagramm des Systems Toll-Collect

Hinweise zur Abbildung: GUI für Beispielanwendung



Was fällt ihnen dabei auf? Was könnte man verbessern? Was fehlt?

Was man verbessern könnte:

Was ist hier OK?

Was ist Close?

Warum stehen die Buttons übereinander?

Wie kann ich editieren, speichern, löschen, neu anlegen?

Kann man sortieren?

Was ist mit Versionierung?

und sicher noch einiges mehr...