6 Entwurf

> Design is the art of separation, grouping, abstraction, and hiding. The fulcrum of design decisions is change. Separate those things that change for different reasons. Group together those things that change for the same reason.
>
> *(Bob Martin)*

Artefakt der Phase Entwurf: PDF Dokument mit UML-Diagrammen, Klassenbeschreibungen, Erläuterungen, Designentscheidungen; evtl. großformatiges Klassendiagramm

6.1 Inhalt

- Einleitung mit grobem Überblick. Dieser Abschnitt soll an das Pflichtenheft anschließen und die Aufteilung in die Pakete erklären.
- Detaillierte Beschreibung *aller* Klassen. Das beinhaltet (JavaDoc) Beschreibungen zu allen Methoden, Konstruktoren, Packages und Klassen. Was hier *nicht* reingehört sind private Felder und Methoden. Das sind Implementierungsdetails.
- Beschreibung von charakteristischen Abläufen anhand von Sequenzdiagrammen. Beispielsweise bieten sich Testszenarien aus dem Pflichtenheft hier an. Wir empfehlen Sequenzdiagramme möglichst früh zu erstellen, denn dabei werden die Schnittstellen zwischen Packages und Klassen klar.
- Mit Blick auf den Implementierungsplan: Aufteilung in Klassen/Pakete, die unabhängig voneinander implementiert und getestet werden können.
- Änderungen zum Pflichtenheft, z.B. gekürzte Wunschkriterien.
- Vollständiges großformatiges Klassendiagramm im Anhang. Ausschnitte/Teile können bereits vorher verwendet werden, um Teilkomponenten zu beschreiben. Assoziationen zwischen Klassen dabei bitte mit entsprechenden Pfeilen darstellen, statt nur durch Feldtypen. Ein kleines Beispiel ist in Abbildung 1 gezeigt.
- Identifikation von Entwurfsmustern um Struktur gröber zu beschreiben.
- Erfahrungsgemäßer Umfang:
  - 100 Seiten, primär Klassenbeschreibungen
  - 40–80 Klassen ohne Interfaces
- Möglicherweise weitere UML-Diagrammarten?
- Formale Spezifikation von Kernkomponenten?

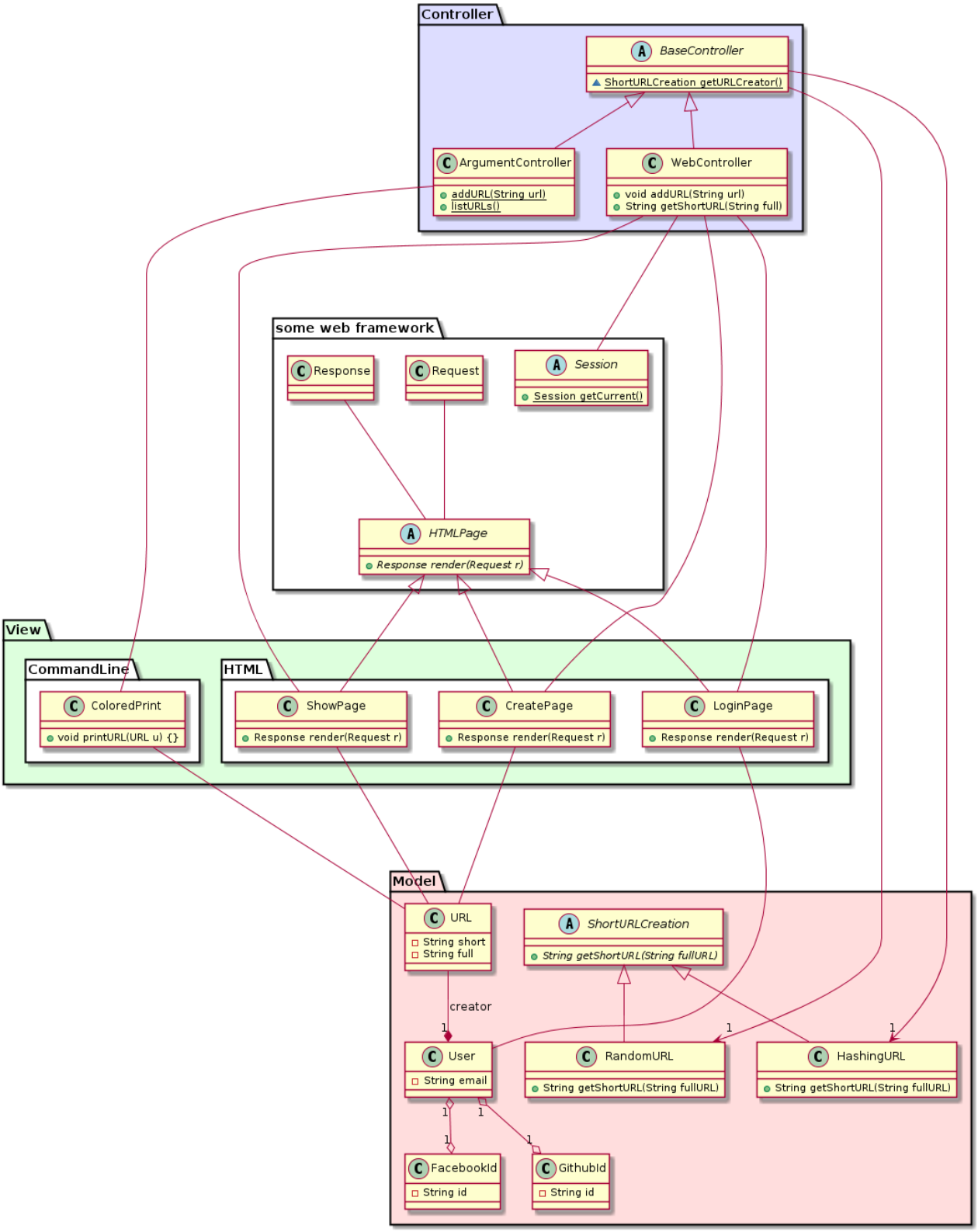

Abbildung 1: Klassendiagrammbeispiel: Grobe Paketstruktur farblich hervorgehoben. Ausgewählte Verwendungen als Assoziation eingetragen, um auch Beziehungen durch Parameter oder lokale Variablen zu verdeutlichen.

Unserer Erfahrung nach ist die Entwurfsphase die schwierigste, da Studenten hier so gut wie keine Erfahrung haben (im Gegensatz zur Implementierungsphase). Deswegen hier ein paar konkrete Schritte, die helfen sollen einen Anfang zu finden.

1. Sucht alle relevanten Substantive im Pflichtenheft heraus. Viele davon lassen sich direkt als Klassen abbilden. Zum Beispiel: Button, Level, Spielfigur, Bild, etc.

2. Was kann man mit diesen Objekten tun? So erhält man die Methoden. Beispielsweise kann man ein Bild drehen, anzeigen und umfärben.

3. In welcher Beziehung stehen die Objekte miteinander? So erhält man die Pfeile im UML Diagramm. Zum Beispiel weiß eine Spielfigur vielleicht in welchem Level sie ist.

4. Spielt die Testfallszenarien durch. So finden sich noch fehlende Methoden und Assoziationen. Außerdem könnt ihr dabei schon die ersten Sequenzdiagramme sammeln.

5. Organisiert die Klassen sinnvoll in Gruppen. So erhält man die Paketstruktur. Beispielsweise Model-View-Controller könnte hier sichtbar werden.

6. Diese ersten Schritte kann man erstmal zügig in einer Sitzung mit dem ganzen Team machen. Anschließend kann man die Pakete oder Klassen einzelnen Personen zuordnen, die dann eigenverantwortlich die Details ausarbeiten.

7. Nun ist die erste Woche rum. Man hat einiges an Material produziert und kann es dem Betreuer zeigen.

8. Jetzt ist es auch an der Zeit mit dem eigentlichen Dokument zu beginnen. Inbesondere sollten technische Fragen geklärt sein. Womit erstellen wir unsere UML Diagramme? Automatisches Erzeugen von Java-Code mit JavaDoc-Kommentaren und daraus LaTeX und PDF erzeugen? Das alles sollte man in der zweiten Woche zum Laufen bringen.

9. Da man nun bereits ein Dokument hat, kann man grundlegende Entwurfsentscheidung sofort niederschreiben. Sammelt erstmal alles in dem Dokument. Ordnen und polieren kann man das später in der Phase.

10. Ein weiterer großer Brocken ist die Anbindung an die Platform, die man benutzt. Beispielsweise Qt, Android oder ein anderes Framework. Hier ist es oft ratsam ein paar minimale Beispielanwendungen zu bauen, um ein Gefühl für die API zu bekommen.

11. Nach zwei Wochen sollten die meisten Klassen entworfen sein. Allerdings hat üblicherweise jedes Teammitglied seinen Bereich für sich bearbeitet. Nun wird es Zeit sich mit der Integration zu beschäftigen. Nehmt euch noch einmal die Testfallszenarien aus dem Pflichtenheft vor und geht diese anhand des Klassendiagramms detailliert durch. Wer ruft wen mit welchen Argumenten auf? Üblicherweise fallen dabei viele Details auf, wo man mehr Informationen weitergeben muss.

12. Auch nicht zu vergessen sind Dinge die nicht im UML auftauchen. Bilder, SQL-Schema, JSON-Schema, Tools wie ein Leveleditor, etc.

13. Nun sind drei von vier Wochen rum und eigentlich sollte der Entwurf so ziemlich vollständig sein. Jetzt ist genügen Zeit für Feinschliff, Präsentation und Konsistenzprüfung.

6.2 Bewertung eines Entwurfs

Hier ein paar Tipps, wie man einen Entwurf einschätzen kann. Paradoxe Anforderungen zeigen, dass gutes Design eine Kunst ist.

- Geheimnisprinzip (information hiding) beachtet? Jede Entwurfsentscheidung sollte in genau einer Klasse gekapselt sein, so dass eine Änderung dieser Entscheidung auch nur diese eine Klasse betrifft. Allgemein sollte ein Klasse/Paket möglichst wenig interne Details nach außen preisgeben.

- Dazu gehört: Behalten Klassen ihre internen Datenstrukturen für sich? Eine Klasse, die eine Liste von Objekten verwaltet, sollte selbst Methoden zum Hinzufügen, Löschen, u. s. w. bereitstellen. Sie sollte *keine* veränderbare Referenz auf die Liste nach außen geben und ihre Aufrufer die Liste verändern lassen. Für Java siehe z. B. Unmodifiable View Collections.

- Lose Koppelung zwischen Klassen/Paketen? Abhängigkeiten zu fremden Schnittstellen machen spätere Änderungen aufwendiger. Im UML-Diagramm sollten möglichst wenig Verbindungen zwischen Klassen zu sehen sein.

- Keine indirekten Abhängigkeiten? Wenn eine Abhängigkeit zwischen Objekten sein muss, soll sie direkt und explizit sein. Sich stattdessen an Referenzen entlangzuhangeln sieht auf dem Papier aus wie lose Kopplung, koppelt aber tatsächlich mehr Klassen enger aneinander. Siehe auch Law of Demeter.

- Starke Kohäsion innerhalb von Klasse/Paket? Wenn Methoden einer Klasse eigentlich unabhängig voneinander sind, ist es ein Zeichen, dass eine Auftrennung in zwei Klassen sinnvoll sein könnte. Kohäsion führt zu besserer Wiederverwendbarkeit der einzelnen Klassen.

- Klassen/Pakete sind gleichzeitig erweiterbar und stabil? Erweiterbarkeit bei stabilem Interface ist der große Vorteil von objekt-orientiertem gegenüber prozeduralem Entwurf, der durch Vererbung und Polymorphie erreicht wird. Siehe auch Open-Closed Principle.

- Liskovsches Substitutionsprinzip bei Vererbung erfüllt? Unterklassen sollten alle Nachbedingungen und alle Invarianten der Oberklasse erfüllen. Andernfalls könnte es zu Fehlern kommen, wenn eine Unterklasse als Oberklasse verwendet wird.

- Verhalten von Implementierung getrennt? Das Verhalten (Was soll getan werden) ändert sich sehr viel häufiger als die Implementierung (konkrete Algorithmen). Beispielsweise sind Sortieralgorithmen recht statisch, während die Frage wonach sortiert werden soll, sehr flexibel sein sollte.

- Keine zyklischen Abhängigkeiten? Beispielsweise ist eine zyklische Abhängigkeit von Konstruktoren schlicht nicht möglich. Eine zyklische Abhängigkeit von größeren Modulen bedeutet, dass man alles auf einmal implementieren muss, bevor irgendwas funktioniert. Entwurfsmuster um Abhängigkeiten zu entfernen: Observer, Visitor, Strategy

- Lokalitätsprinzip beachtet? Eine Änderung der Spezifikation sollte nur lokale Änderungen benötigen. Das impliziert: Eine Klasse/Paket/Methode sollte für sich verständlich sein, ohne dass Kontext notwendig ist.

- Sind Methoden frei von unerwarteten Nebeneffekten? Eine Methode, die Informationen aus

einem Objekt zurückgibt, sollte nichts Wesentliches am Objektzustand verändern. Auch eine Methode, die dazu dient, den Objektzustand zu verändern, sollte nur eine überschaubare Änderung durchführen. Siehe auch Command-query separation

Ähnlich zu den üblichen Entwurfsmustern gibt es auch Anti-Entwurfsmuster. Also Muster, die man im Entwurf erkennen kann, die praktisch immer zu Problemen führen. Ein paar Beispiele, die in vergangenen PSE-Projekten auftraten:

- God Object. Wenn zuviel Funktionalität in eine Komponente (Klasse) gesteckt wird. Es verletzt Lokalitäts- und Geheimnisprinzip.
- Anemic domain model. Wenn das Model praktisch nur noch Datenspeicher ist. Objekt-Orientiertes Design zeichnet sich gerade dadurch aus, dass Daten *und* Verarbeitung in Objekten zusammengebaut werden. Objekte, die nur zur Datenhaltung da sind, erzwingen prozedurale Programmierung.
- `switch` und `instanceof`. Sieht man im Entwurf zwar noch nicht direkt, ist aber manchmal absehbar. Dynamische Bindung ist meistens die bessere Wahl.

6.3 UML Diagramme

- UMLet noch ein UML Tool
- Umbrella um einfach nur Diagramme zu zeichnen
- BOUML noch ein UML Tool
- UMLGraph für Versions-Control-freundliche UML Diagramme. (Tipp: Alles in eine Datei und erst in der Implementierungsphase aufspalten)
- PlantUML noch ein Tool für Versions-Control-freundliche UML Diagramme. Allerdings eigene Syntax, so dass kein Java-Code daraus erzeugt werden kann.
- ObjectAid UML Explorer for Eclipse ein Source-to-Diagram plugin.
- Eclipse Papyrus
- Visual Paradigm
- ArgoUML um auch Code zu generieren und beim Entwurf zu helfen (Vorsicht: keine „Undo“-Funktionalität)

6.4 Mehr Links

- TeXdoclet um mit JavaDoc LaTeX zu erzeugen. Das ermöglicht den automatischen Flow: ArgoUML → JavaDoc+TeXdoclet → LaTeX → Section „Klassenbeschreibung“ im Entwurf.
- Prinzipien für den objektorientierten Entwurf, Guter Überblicksartikel.

- Prinzipien Der Softwaretechnik Blog zum Thema Prinzipien im Software Engineering.
- Game Programming Patterns Buch zu Design Patterns in der Spieleprogrammierung
- Example Toolchain zum Wiederverwenden
- How to Write Doc Comments for the Javadoc Tool mit einigen Tipps und Beispielen.

6.5 Inhalt der Präsentation

- Gerade für UML-Diagramme wäre eine unkonventionelle Präsentation mit Prezi einen Versuch wert. Man kann ein großes allumfassendes Klassendiagram zeigen und dann in Packages und Klassen reinzoomen.
- Kurze Einführung und Verbindung zum Pflichtenheft
- Ein Sequenzdiagram als anschauliches Beispiel mit Ausschnitten aus dem Klassendiagramm. Hier kann man ein Testfallszenario aus dem Pflichtenheft auswählen.
- Überblick über das Gesamtklassendiagramm, Pakete, Module geben. Hier kann man über die Verwendung von Entwurfsmustern erzählen. Das **Klassendiagramm** sollte den Kern der Präsentation bilden, also sollte man hierfür auch die meiste Zeit aufwenden.
- Ein großformatig ausgedrucktes Klassendiagramm vermittelt einen Eindruck von der Gesamtarchitektur und ist für die folgende Diskussion nützlich.
- Einhaltung softwaretechnischer Prinzipien zeigen (z.B. Kohäsion, Lokalitätsprinzip, etc.)
- Gestrichene Wunschkriterien können, aber müssen nicht erwähnt werden. Man muss hier aufpassen, dass kein negativer Eindruck zurückbleibt. Für manches gibt es gute Gründe, aber oft ist es geschickter Streichungen nur im Dokument zu erwähnen.
- Verwendete externe Resourcen (Bilder, Frameworks, Bibliotheken, Sounds, Musik, etc.)
- Nur kurz erwähnen weil eher langweilig: eigene Formate, Datenbankschemata, Einstellungen, Menüstruktur und Dialoge.