Architektur, Analyse und Design

Wie man Software entwirft

Christian Rehn

4. Mai 2012

Organisatorisches [Folie 2]

- Folien enthalten nur wenig Text
  - Besser für Präsentation
  - Als Handout/Skript steht eine ausführlichere Version online
- Inhalte wichtig für Projekt und prüfungsrelevant!
- Die Saalübung heute ist eher vorlesungsartig
  - Geplant: ca. 60 min. „Vorlesung“ + ca. 30 min. Aufgaben
  - Trotzdem auch im ersten Teil Fragen und aktive Mitarbeit wichtig
  - Feedback erwünscht

Was ich hier erzähle [Folie 3]

- SE2: Wie man Analyse- und Entwurfsmodelle *aufschreibt*
  - Notation: UML
- Hier: Wie man zu diesem Modellen kommt
  - Denkweise, Prinzipien, Daumenregeln, hilfreiches Wissen, ...

Es ließe sich noch viel mehr sagen, aber nicht in 90 Minuten. Man kann darüber problemlos ganze Vorlesungen halten. Ich kann hier also nur einen groben Überblick geben.

Das, was ich hier erzähle, wird nur selten gelehrt (warum auch immer). Auch im Netz finden sich erstaunlich wenige umfassende Einführungen (wohl aber viele Artikel zu speziellen Themen). Wer an **anderen Sichtweisen** auf das Thema interessiert ist, kann sich z. B. [14], [25] und [26] ansehen.

Überblick [Folie 4]

Inhaltsverzeichnis

1. Motivation und Begriffe **3**
- 1.1. Warum überhaupt nachdenken? . . . 3
- 1.2. Begriffsverwirrung . . . 4

2. Szenarien und Daumenregeln **8**
- 2.1. Szenarien . . . 8
- 2.2. Daumenregeln . . . 9

3. Abhängigkeiten und Schichten **19**
- 3.1. Abhängigkeiten . . . 19
- 3.2. Schichten . . . 26

4. Patterns **31**

A. Anhang **34**
- A.1. Aufgaben . . . 34
- A.2. Aktivitäts- und Zustandsdiagramme . . . 36

---

1. Motivation und Begriffe

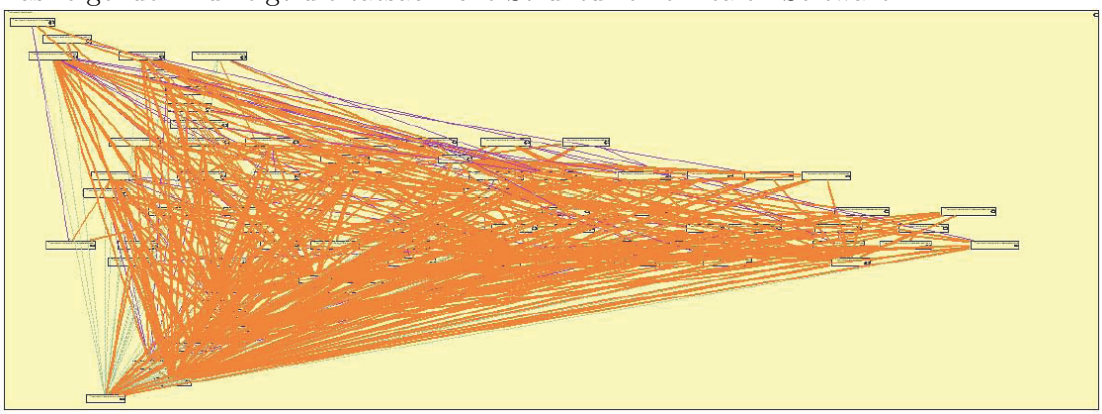
1.1. Warum überhaupt nachdenken?

Warum überhaupt nachdenken? (1/2) [Folie 6]

Warum sollten wir überhaupt nachdenken, bevor wir programmieren?

Warum überhaupt nachdenken? (2/2) [Folie 7]

Das folgende Bild zeigt die tatsächliche Struktur einer realen Software.



[1]

Und das ist nur ein Subsystem von zwanzig. Wenn Software so aussieht, führt das fast unweigerlich zu unnötigen Bugs.

Das Bild wurde von einer Software des Fraunhofer IESE [16] [3] aus dem Code generiert, zeigt also die **tatsächlichen Abhängigkeiten**. Es ist anzunehmen, dass die Entwickler *keine* Idioten waren. Wahrscheinlich haben sie sich sogar Gedanken über die Architektur gemacht. Das zeigt, dass es schwer ist, die Architektur nicht aus den Augen zu verlieren – insbesondere, wenn man im Team und unter Projektbedingungen (Zeitdruck, etc.) arbeitet.

Wenn man also schon mit Nachdenken im Chaos landen kann, ist es ohne Nachdenken wohl noch viel schlimmer. Oft merkt man das nicht direkt, wenn man ein Problem auf Architekturebene hat. Alles funktioniert erstmal irgendwie trotzdem. – Und ein halbes

[1] Danke an das Fraunhofer IESE für das Bild

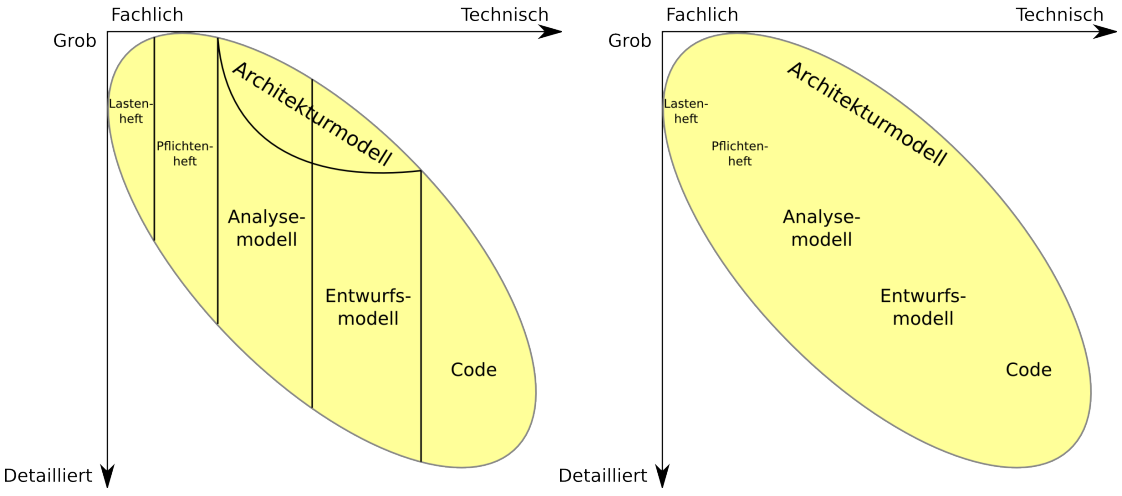
Jahr später kriegt man ernsthafte Schwierigkeiten, die mit der Zeit immer schlimmer werden.

Deshalb ist es sinnvoll, sich immer vorzustellen, dass in einem halben Jahr irgend ein **„Halbidiot“** die Aufgabe erhält, die Software anzupassen. Wenn selbst dieser Halbidiot nicht mehr Fehler einbaut, als er behebt, ist die Software wartbar. Und wenn nicht, hat man ein potenzielles Problem. Oft genug sind wir nämlich selbst dieser Halbidiot. Wir erinnern uns nicht mehr so genau an alles, vergessen gewissen Einschränkungen, machen inkonsistente Änderungen, etc.

1.2. Begriffsverwirrung

Begriffsverwirrung [Folie 8]

Das „Softwareentwicklungs-Ei“ [Folie 9]



- SE2 [15]: Trennung von Analyse- und Entwurfsphase bzw. fachlicher und technischer Lösung
- Andere Möglichkeit der Trennung:
  - Architektur als grobe, alles überspannende, technische Struktur
  - Entwurf als Detaillierung der Architektur auf konkrete Problemlösungsebene
- Beide Trennungen (grob ⇒ fein und fachlich ⇒ technisch) sind immer vorhanden
- Nicht immer ganz klar voneinander zu trennen; Übergänge fließend

Beispiele:

klar Architektur Definition von Subsystemen und Schichten, Bestimmung des Architekturstils (Layers, Pipe and Filter, Blackboard, ... [9]), Nutzung von Komponentenframeworks (EJB, etc.), „alle zu persistierenden Klassen werden von PersistentObject abgeleitet“, „Keine Klasse aus dem Package `X` darf auf die GUI zugreifen“, „Die Kommunikation zwischen `X` und `Y` geschieht über Java RMI“, ...

klar Analyse „Die Klasse `Person` hat das Attribut birthday“, „Reader ist eine Subklasse von Person“, ...

klar Entwurf „Die Klasse User implementiert das Interface LibraryListener“, „MainWindow ist von JFrame abgeleitet“, ...

klar Implementierung „Zum Suchen in der Liste wird Binärsuche verwendet“, „Hierfür eine for-Schleife...“, ...

zwischen Analyse und Entwurf Klasse MainWindow

zwischen Architektur und Entwurf Definition von konkreten Klassen (statt nur Subsystemen und „Komponenten“)

zwischen Entwurf und Implementierung „Diese List instanziiere ich als ArrayList“, „Library hat eine private Methode computeDueDate“, ...

[6] zeigt sehr schön den Unterschied zwischen Analyse und Design.

OOA und OOD [Folie 10]

natürliche vs. künstliche Klassen[2]

- OOA: Object Oriented Analysis
  - Fachliche Modellierung der Problemstellung
  - Führt zu „natürlichen“ Klassen
  - Beispiele: `Book`, `Reader`, `Library`, ...
- OOD: Object Oriented Design (= Entwurf)
  - Technische Ausgestaltung
  - Technische Aspekte auf natürliche Klassen verteilen
  - *Wenn nötig*, „künstliche“ Klassen ergänzen
  - Beispiele: `LendingEventListener`, `ErrorHandler`, `SessionManager`, ...
- Natürliche Klassen sind intuitiver und sollten deshalb erstmal bevorzugt werden
- Oft ist die Unterscheidung natürliche ⇔ künstliche Klasse ganz einfach und intuitiv zu treffen. Es gibt aber Fälle, in denen die Unterscheidung verschwimmt.
- Gefahr: Pseudo-OOP
  - Die Verwendung von Klassen macht einen Entwurf noch nicht objektorientiert
  - Häufiges Problem:
    - Übermäßige Verwendung von künstlichen Klassen
    - Alarmsignal: Viele Klassennamen enden auf -er, -or, -Handler, -Manager, ...
  - Grund: prozedurales Denken
    - „Ich brauch jetzt etwas, das XY macht.“

[2] Die Begriffe stammen von mir; allgemein anerkannte sind mir dafür nicht bekannt.

Architektur [Folie 11]

Architektur: Definition des SEI
The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. [1]

Architektur: Meine Definition
Die Softwarearchitektur beschreibt die groben Strukturen der Software und definiert wie man als Entwickler von der Software *denkt*.

- Durch eine gute Architektur kann man sich leichter im Code zurecht finden und weiß, wie man die Software „anfassen muss“ um etwas zu ändern oder hinzu zu fügen.
- Architektur (aber auch Detaildesign) wird in Sichten (Views) dokumentiert, die einzelne Teilaspekte beschreiben[3]
- Teilaspekte: statische Struktur (d. h. Struktur der Klassen, etc.), dynamische Struktur (d. h. Struktur zur Laufzeit), Verhalten, Zuordnung zu Dateien/Verzeichnissen, ...
- Analogie zur Gebäude-Architektur: Verschiedene Pläne (Grundriss, Aufriss, Plan der Wasser- und Abwasserleitungen, Plan der Stromleitungen, ...)

[3] Mehr dazu in den Vorlesungen *Grundlagen des Software-Enigeering* (GSE) und *Softwarearchitektur verteilter Systeme* (SAVS)

2. Szenarien und Daumenregeln

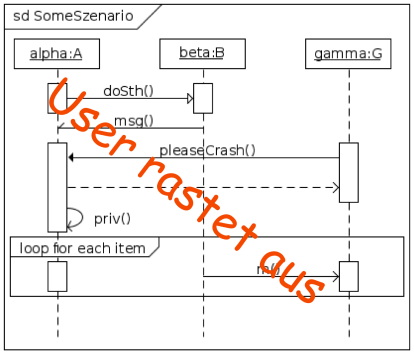
2.1. Szenarien

Beim Erstellen der Architektur hilft es, einzelne Szenarien durch zu spielen.

Szenarien [Folie 13]

Szenarien für ...

- Funktionalität
- Nicht-funktionale Anforderungen
- Softwareentwicklung und -wartung



Beispiele:

- Use Cases: Leser leiht ein Buch aus ⇒ Funktionalität
- Missuse Cases: Leser verschafft sich unberechtigt Zugang zum Verwaltungssystem ⇒ Sicherheit
- Fehlerfälle: Das Programm stürzt während eines Schreibvorgangs ab ⇒ Robustheit
- Lastfälle: 1000 Nutzer wollen gleichzeitig den Bücherbestand durchsuchen ⇒ Performance
- Änderungsfälle: Entwickler portiert die Software von MySQL auf ein anderes DBMS ⇒ Modifizierbarkeit
- Erweiterungsfälle: Entwickler ergänzt die Möglichkeit die Ausleihfrist nachträglich zu verlängern ⇒ Erweiterbarkeit
- Wartungsfälle: Entwickler fixt einen Bug in der Datenhaltung ⇒ Wartbarkeit
- ...

Szenarien nutzt man ganz intuitiv. Das allein wäre also nicht erwähnenswert. Das Problem ist eher: Man vergisst leicht wichtige Szenarien – insbesondere solche, die sich nicht auf die Funktionalität beziehen.

Nutzen von Szenarien [Folie 14]

- Identifizieren von Klassen, Methoden, etc.
- Auffinden von Problemen
- Vergleich unterschiedlicher Ansätze

Wie kann man Szenarien durchspielen?

- Im Kopf
- Durch Pseudocode
- Mit Diagrammen (z. B. Aktivitätsdiagramme, Sequenzdiagramme oder Kommunikationsdiagramme)
- In verteilten Rollen (jeder Entwickler „spielt“ ein Modul)

2.2. Daumenregeln

- Szenarien sind aufwändig
- Prinzipien/Heuristiken/Daumenregeln sind oft praktischer
  - (können Szenarien aber nicht ersetzen)

The Tradeoff Game [Folie 15]

Wie ich Softwareentwicklung sehe
Softwareentwicklung ist das ständige Ausbalancieren diverser Prinzipien („Daumenregeln“).

- Prinzipien widersprechen sich scheinbar/wirklich
  - „wirken in unterschiedliche Richtungen“

- Ein Ausgleich/Gleichgewicht ist zu finden
- Dieses „Ausbalancieren“ nennt man im Englischen „Tradeoff“[4]
- Mehr dazu in [22].

Meine obersten drei Daumenregeln [Folie 16]

1. Brich die Regeln
2. Die Niemals-heißt-nie-nie-Regel
3. Die Antwort auf alle Fragen: it depends

---

1. Brich die Regeln
Wenn du meinst, eine Regel brechen zu müssen, dann tu es. Wundere dich aber nicht über die Konsequenzen.

2. Die Niemals-heißt-nie-nie-Regel
Wenn eine Regel besagt, dass etwas niemals getan werden darf, so bedeutet das “niemals” niemals wirklich “nie”. Ebenso bedeutet “immer” nie wirklich “immer” und in jeder Situation. Es ist immer eine Abwägungssache.

3. Die Antwort auf alle Fragen
Die Antwort auf alle Fragen heißt ~~42~~ „it depends“ bzw. „kommt drauf an“.

Weiteres in [21] und [19].

Technical Debt [Folie 17]

- Es kann auch nötig/sinnvoll/unausweichlich sein, für eine gewisse Zeit eine schlechte Lösung zu akzeptieren.

[4] Ein wirklich äquivalentes deutsches Wort dafür existiert nicht.

  - Kompromiss zwischen zeitlich Möglichem und technisch Notwendigem

- Analogie: Schulden: Schulden machen ist nicht per se schlecht (es kann sogar wirtschaftlich sein) aber:
  - Man zahlt Zinsen und Zinseszinsen!
  - Überschuldung ist schlecht
  - Die Relation muss stimmen: Brauche ich den diamantbesetzen Kugelschreiber wirklich? Soll ich wirklich für das Kaugummi eine Hypothek aufnehmen oder lieber doch zum Auto gehen und meinen Geldbeutel holen?
  - Man muss seine Schulden überblicken und Verwalten
- Technische Schulden sind ähnlich wie Geldschulden.[5]
- Mehr zu „Technical Debt“ in [30] und [7]

Grundlegende Daumenregeln [Folie 18]

- Divide and Conquer
- Keep it Simple, Stupid (KISS)
- Abstraktionsprinzip
- ...

KISS
Wenn es eine einfache und eine komplizierte Möglichkeit gibt, ein Problem zu lösen, nimm die einfache, auch, wenn sie noch so dumm erscheint.

Abstraktionsprinzip bzw. Generalisierungsprinzip
Wenn es für ein Problem sowohl eine spezielle Lösung gibt, die nur dieses eine konkrete Problem löst, als auch eine allgemeine, die darüber hinaus noch weitere Probleme löst, so nimm die allgemeine. Verwende abstrakte Konzepte, die sich auf eine Vielzahl von konkreten Problemen anwenden lassen.

[5] Im Übrigen kann die Analogie helfen den vorgesetzten BWLer zu überzeugen, dass gewisse Tätigkeiten notwendig sind.

More is More Complex [Folie 19]

 [6]

More is More Complex[7]

Viel Code ist tendenziell komplizierter als wenig Code. Lange Methoden sind schlecht, viele Klassen sind schlecht, viele Librarys sind schlecht, viele Abhängigkeiten sind schlecht, etc. Der menschliche Geist ist nur begrenzt aufnahmefähig.

An der More-is-More-Complex-Regel sieht man ganz deutlich, dass solche Regeln nicht absolut gelten. Oder anders gesagt: Das implizit vorhandene, zwischen den Zeilen stehende, „immer“ heißt nach der „Niemals-heißt-nie-nie-Regel„ nicht wirklich „immer“. Selbstverständlich gibt es viele gute Beispiele, bei denen eine längere Methode deutlich leichter und verständlicher ist als eine kurze, die den selben Zweck erfüllt. Aber tendenziell ist es z. B. ein Nachteil, wenn eine bestimmte Lösung verlangt, dass eine zusätzliche Klasse definiert wird.

Den Extremfall, bei dem die Software aus einer undurchsichtigen Sammlung vieler kleiner, künstlicher Klassen besteht, nennt man Ravioli-Code.

Beispiel: Was ist besser?

- Ein Stück Code, der die Quadratwurzel von 2 berechnen kann
- Eine Funktion, die beliebige Quadratwurzeln aus reellen Zahlen ziehen kann
- Eine Funktion, die beliebige Wurzeln aus reellen Zahlen ziehen kann
- Eine Funktion, die beliebige Wurzeln aus komplexen Zahlen ziehen kann
- Eine Funktion, die beliebige Potenzen komplexer Zahlen berechnen kann
- Ein Modul, das beliebige Gleichungen über reellen Zahlen lösen kann
- Ein Modul, das beliebige Gleichungen über komplexen Zahlen lösen kann

[6] CC-BY Dezidor http://commons.wikimedia.org/wiki/File:Ravioly.jpg

[7] Die Formulierung stammt von mir; eine verbreitete Formulierung dieses Prinzips ist mir nicht bekannt.

Don't Repeat Yourself [Folie 20]

DRY ~~sag nicht alles doppelt~~ ~~wiederhol' dich doch nicht immer so~~

- Keine doppelte Datenhaltung!
  - Auch nicht in unterschiedlicher Repräsentation
- Kein doppelter Code!
  - Auch kein Copy'n'Paste mit kleinen Änderungen
  - Besser: Methoden mit Parametern
- Mehr zu DRY in [12]

Modellprinzip [Folie 21]

Objekte sollten vorrangig etwas *sein* und nur „nebenbei" etwas *tun*.

Modellprinzip
Baue ein Objekt-Modell der Prozesse, die durch die Software automatisiert werden sollen.

Idee der OO ist es, quasi die „Realität" bzw. das, was durch die Software gemacht werden soll, mit Objekten „nach zu bauen" also quasi zu „simulieren". Die Objekte bzw. deren Klassen sollten sich direkt aus der Problemdomäne ergeben. Möchte man beispielsweise Datenbankverbindungen verwalten, so ist „Datenbankverbindung" eine Klasse und nicht etwa „Datenbankverbindungsverwaltung". Auch wenn die Klasse natürlich nicht die Verbindung selbst ist, so repräsentiert sie diese dennoch. Von einem konzeptionellen Standpunkt aus, *ist* die Klasse (bzw. deren Instanz) die Datenbankverbindung.

Solche Verwaltungs-, Manager- und -Controller-Klassen sind oft ein Anzeichen dafür, dass man prozedural denkt. Prozedurales Denken ist „Ich brauch jetzt etwas, das XY tut. Also mach ich daraus eine Prozedur." Wenn man bei dieser **Denkweise** bleibt und nur „Prozedur" durch „Klasse" ersetzt, wird das Resultat nicht wirklich objektorientiert sein.

Negativbeispiel: Eine Klasse `SelectStatementBuilder` baut aus verschiedenen Informationen, wie anzuzeigende Felder, Sortierkriterium und diversen Filterkriterien ein SQL-SELECT-Statement, das Bücher aus einer Datenbank abfragt.

Single Responsibility Principle (SRP) [Folie 22]

Eine Modul sollte genau *eine* Aufgabe haben

- Ein „Modul“ ist hier typischerweise eine Klasse
  - Ähnliches gilt auch für Methoden und Subsysteme
  - Aber auf einer anderen Abstraktionsebene
- Robert C. Martin in [18]: „There should never be more than one reason for a class to change.“
  - In Martins Artikel sieht man auch, dass das Prinzip mehr in sich hat, als man auf den ersten Blick vermutet.
  - Eine Klasse hat genau dann mehrere Verantwortlichkeiten, wenn es mehrere Gründe gibt, sie zu ändern.
- Typisches Beispiel: Eine Klasse sollte nicht gleichzeitig Benutzerinteraktion und Datenhaltung übernehmen.

Kapselung und Information Hiding [Folie 23]

Objekte sind wie Abwasserleitungen

- Objekte sind wie Abwasserleitungen: Ich will, dass sie zuverlässig funktionieren; mit den Details will ich aber nichts zu tun haben.
- Definitionen zu Kapselung und Information Hiding: Siehe SE1.
- Eine leicht andere, aber vermutlich verbreitetere Definition bzw. Erklärung findet sich in [23].
- Siehe hierzu auch die Vorlesung *Fortgeschrittene Aspekte Objektorientierter Programmierung* (FAsOOP)

Encapsulate the Concept that Varies [Folie 24]

Encapsulate the Concept that Varies
Überlege bei der Modellierung, was sich ändert oder wahrscheinlich ändern wird. Das sollte dann in einem separaten Modul gekapselt werden.

Dieses Prinzip ist Grundlage vieler Design Patterns der Gang of Four und wird auch in deren Buch [8] erwähnt (welches übrigens ebenfalls sehr lesenswert ist). Nur das, was auch gekapselt ist, lässt sich auch leicht verändern, austauschen und wiederverwenden. Deshalb ist es wichtig, dass alles, was sich ändert auch gekapselt ist.

Beispiel: Eine Software zur Produktionsoptimierung soll verschiedene Strategien/Algorithmen verwenden können, um das Optimierungsproblem zu lösen. Hier kann es hilfreich sein, den Algorithmus in eine separate Klasse auszulagern[8].

Speculative Design [Folie 25]

Was wäre wenn...

- Diese Denkweise führt zu „spekulativem Design“.
  - Man überlegt, wohin sich die Software einmal hin entwickeln könnte.
  - Erweiterungs-Szenarien
- Achtung! Nicht übertreiben!
  - Tendenziell komplexere Designs, aber womöglich kein Nutzen daraus, wenn vermutete Änderungen nicht eintreten
  - Traditionelle Softwareentwicklung setzt verstärkt auf spekulatives Design, agile Softwareentwicklung argumentiert dagegen
  - Ein gesundes Maß sollte nicht überschritten werden

Die MP-Gruppen haben explizit die Aufgabe, spekulatives Design zu betreiben: In deren Szenario ist „bekannt“, dass die Software zukünftig um weitere Kartenspiele erweitert wird.

[8] Das wäre das Strategy-Pattern [8] [29]; Mehr zu Patterns später

Tell, don't Ask! oder: Do It Myself [Folie 26]

Objekt: „Das kann ich alleine!“

Der prinzipielle Gedanke der Objektorientierung ist ja, dass man die Funktionalität auf verschiedene Objekte verteilt und diese dann über „Nachrichten“ (technisch gesehen Methodenaufrufe) kommunizieren. Dabei hat jedes Objekt seine klar definierte Aufgabe. Wenn man jetzt aber nur Daten aus einem Objekt holt, diese Verarbeitet und wieder zurück schreibt, so ist das Objekt quasi ein toter Datenbehälter. Das ist die Vorgehensweise wie man sie bei prozeduralen Programmen findet. In der Objektorientierung sollte man Aufgaben delegieren. „Objekt, mach mal!“ *Wie* es letztendlich seine Aufgabe erledigt, ist prinzipiell egal. Wichtig ist nur, *dass* die Aufgabe erledigt wird. In der OO sagen die Objekte quasi immer „Das kann ich alleine!“. Und dann sollte man sie auch alleine machen lassen.

Der Artikel von Andrew Hunt und David Thomas zu „Tell don't ask!“ [13] ist ausgesprochen lesenswert und enthält noch weitere Infos. Die Geschichte **„The Paperboy and the Wallet“** findet sich auch in [2]. Dort wird auch das Gesetz von Demeter erläutert, das ein verwandtes Prinzip darstellt.

Tell, don't Ask! – Gegenbeispiele [Folie 27]

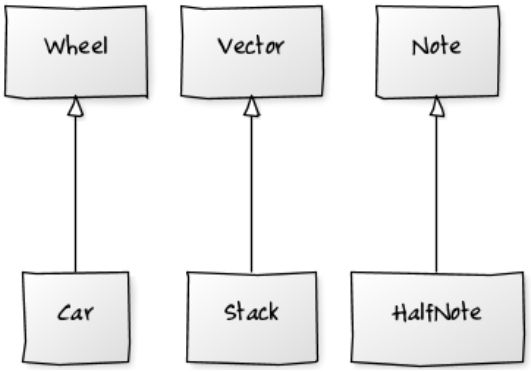
```
// ganz typisch:
myObject.setValue(myObject.getValue() + 1);

// oder auch:
if (myTime.getMinute() == 59)
{
    myTime.setMinute(0);
    myTime.setHour(myTime.getHour() + 1);
}
else
{
    myTime.setMinute(myTime.getMinute() + 1);
}
```

Delegation vor Vererbung [Folie 28]

Vererbung wird oft überverwendet

- Das, was man sich an der OO am leichtesten merkt, ist vermutlich die Vererbung. Sie ist *scheinbar* zentral, intuitiv und einfach.
- Oft führt das dazu, dass Vererbung zu oft, unnötig und falsch eingesetzt wird[9]
  - Delegation ist oft besser. Und manchmal braucht man gar keine zusätzliche Klasse.

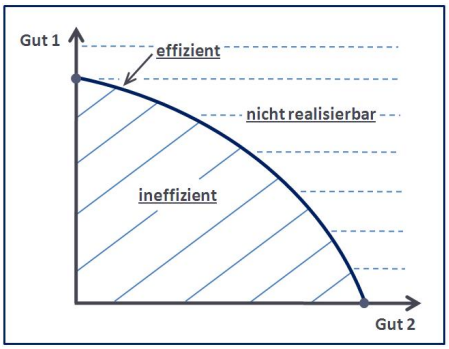
Vererbung – Negativbeispiele [Folie 29]



- Niemand würde auf die Idee kommen, eine Klasse Car von Wheel abzuleiten
  - Ein Auto *ist* kein Rad, sondern *hat* eines (bzw. vier)
- Aber die Java-Entwickler sind auf die Idee gekommen Stack von Vector abzuleiten. Das ist in etwa der selbe Fehler.
- HalfNote sollte man nicht von Note ableiten, da kein neuer Code in der neuen Klasse ist
  - Die Notenlänge ist ein Datum, wird also zum Attribut der Klasse Note

[9] Zu Vererbung siehe auch die Vorlesung *Fortgeschrittene Aspekte Objektorientierter Programmierung* (FAsOOP)

Prinzipien nutzen [Folie 30]



- Prinzipien widersprechen sich (z. B. KISS vs. Abstraktionsprinzip oder Speculative Design vs. KISS) ⇒ Ein Mittelweg (Tradeoff) ist zu finden.
- Anders gesagt: Gesucht ist ein Pareto-Optimum [28]
- Meinungsverschiedenheiten im Team über einzelne Entwurfsentscheidungen entstehen meist dadurch, dass Prinzipien unterschiedliche *gewichtet* werden.
  - Meinungsverschiedenheiten und die Diskussionen, die sie auslösen, sind fast unausweichlich.
  - Sie sind zwar anstrengend aber *gut*! Denn sie helfen dabei, die beste Lösung zu finden und sie sind meistens sehr lehrreich.
  - Lernt damit umzugehen. Nicht in Kleinkram verrennen, nicht verärgert werden, Meinungsverschiedenheiten von Missverständnissen unterscheiden, Entscheidungen fällen. Das ist wichtig.
- Mit der Zeit verinnerlicht man die Daumenregeln so, dass man in einfachen Fällen gar nicht mehr darüber nachdenkt.
- Je komplizierter es wird, desto eher muss man Aktiv einzelne Daumenregeln abwägen oder auf zusätzliche Szenarien auswichen.
- Prinzipien und Szenarien eignen sich gut um Entwurfsentscheidungen zu diskutieren und zu kommunizieren.

3. Abhängigkeiten und Schichten

3.1. Abhängigkeiten

Ripple Effects [Folie 32]

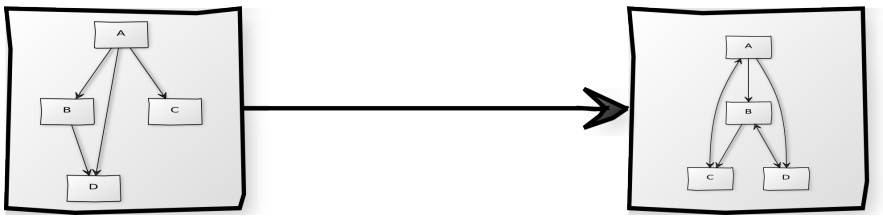
[10]

Ripple Effects

Änderungen an einer Stelle machen Änderungen an weiteren Stellen nötig, die wiederum weitere Änderungen nötig machen.

- Analogie: Ein Wassertropfen erzeugt sich allmählich ausbreitende kleine Wellen (engl. *ripples*).
- Ripple Effects entstehen durch zu viele und zu starke Abhängigkeiten

Bindung und Kopplung [Folie 34]



[10] CC-BY-SA Rainer Zenz http://commons.wikimedia.org/wiki/File:2006-01-14_Surface_waves-2.jpg

Bindung
Bindung (engl. *cohesion*) ist ein Maß für den inneren Zusammenhalt eines Moduls (einer Klasse). Die Bindung sollte immer möglichst hoch sein.

Kopplung
Kopplung (engl. *coupling*) ist ein Maß für die Abhängigkeiten zwischen den Modulen. Die Kopplung sollte möglichst lose sein, d. h. es sollten möglichst wenige Abhängigkeiten bestehen.

- Eine zu geringe Bindung bedeutet, dass ein Modul mehr als eine Aufgabe hat bzw. Dinge tut, die wenig miteinander gemeinsam haben.
  - Das Modul sollte dann aufgeteilt werden.
- Eine zu hohe Kopplung bedeutet, dass starke Abhängigkeiten bestehen und RippleEffects wahrscheinlicher werden.
- Die Unterscheidung, was Bindung und was Kopplung ist, hängt vom Abstraktionslevel ab.
  - Das, was auf Paketebene Bindung ist, ist auf Klassenebene Kopplung.
  - Das, was auf Klassenebene Bindung ist, ist auf Methodenebene Kopplung.
  - Natürlicherweise nehmen die Abhängigkeiten zu, je näher man der konkreten Realisierung kommt.

Traditionell unterscheidet man die folgenden Kopplungsarten:

Kopplungsarten [Folie 35]

Kopplungsarten (von stark bis schwach)

1. Content coupling (Inhaltskopplung)
2. Common coupling (Bereichskopplung)
3. External coupling (Externdatenkopplung)
4. Control coupling (Kontrollkopplung)
5. Stamp coupling (Datenstrukturkopplung)
6. Data coupling (Datenkopplung)
7. Call coupling (Aufrufkopplung)

- Inhaltskopplung am stärksten und die Aufrufkopplung am schwächsten
- Kontrollkopplung bildet in etwa die Grenze zwischen den „guten“ (schwachen) und den „schlechten“ (starken) Kopplungen
- Bezug insbesondere zu prozeduraler Programmierung
  - Definition in den 70er Jahren
  - Auch noch für OOP gültig
  - Teilweise aber leicht andere Gewichtung

Content Coupling (Inhaltskopplung) [Folie 36]

Beispiel:

```
MyFancyOpenDialog dialog = new MyFancyOpenDialog();
dialog.setVisible(true);   // modal

if (dialog.getTextField().getText() != "")
{
    System.out.println("Die folgende Datei wurde ausgewählt: " +
        dialog.getTextField().getText());
}
```

- Auch: „Pathological coupling“ (pathologische Kopplung)
- Abhängigkeit von konkreter Implementierung
- Hier: Was, wenn in Zukunft kein Textfeld mehr verwendet wird?
- Besser: Dem Dialog ein getFileName verpassen

Common Coupling (Bereichskopplung)

- Kommunikation über einen geteilten Speicherbereich
- Kurz: Globale Variablen sind bööööse!

External Coupling (Externdatenkopplung)

- Kommunikation über einen externen Mechanismus
- Dateien, Datenbank, Netzwerk, etc.

Control Coupling (Kontrollkopplung) [Folie 37]

Beispiel:

```
class MyFeedReader
{
    void deleteItem(FeedItem item, boolean recycle);
    {
        if (recycle)
            // throw in dust bin
        else
            // delete completely
    }
}
```

- Parameter ist Kontrollflussinformation
- Besser: zwei getrennte Methoden
- „Gute“ Variante der Kontrollkopplung: Rückgabewert ist Kontrollflussinformation

Die folgenden drei Kopplungsarten sind „gute“ Kopplungen und zeugen i. d. R. *nicht* von Problemen.

Die „guten“ Kopplungen [Folie 38]

Stamp coupling (Datenstrukturkopplung)
Funktionsaufruf mit Übergabe einer komplexen Datenstruktur als Parameter

Data coupling (Datenkopplung)
Funktionsaufruf mit Übergabe von einfachen Parametern (`String`, `int`, etc.)

Call coupling (Aufrufkopplung)
Funktionsaufruf ohne Parameter

Achtung: Auch wenn eine Datenkopplung eigentlich schwächer und damit besser ist als eine Datenstrukturkopplung, ist es keine gute Idee, `String`- und `int`-Werte zu übergeben, wenn logisch gesehen ein Objekt verlangt wird. Aber man sollte natürlich nicht `JTextField` als Parameter definieren, wenn die Funktion prinzipiell mit jedem String zurecht kommen würde.

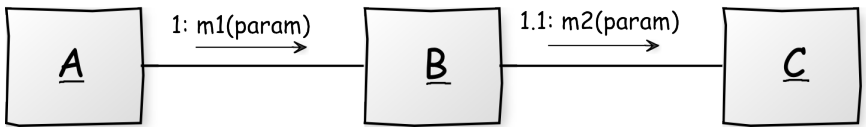
Oft viel wichtiger als die klassischen Kopplungsarten sind die folgenden Sonderformen:

Kopplungsarten – Sonderformen [Folie 39]

Sonderformen

- Tramp coupling
- Hybrid coupling (Hybridkopplung)
- Temporal coupling (zeitliche Kopplung)
- Logical coupling (logische Kopplung)
- Cyclic depepndency (zyklische Abhängigkeit)

Tramp Couplung [Folie 40]



- Ein Parameter wird einfach „durchgereicht“
  ⇒ Unnötige Kopplung durch Kenntnis einer Information, die gar nicht benötigt wird
- Achtung: Manchmal ist eine Art tramp coupling auch gewünscht. Ein Beispiel wäre das Proxy Pattern. [8] [27]

Hybrid Coupling (Hybridkopplung)

```
public void setUpdateInterval(int seconds)
{
    if (seconds == -1)
        // deactivate update
    else
        //set value
}
```

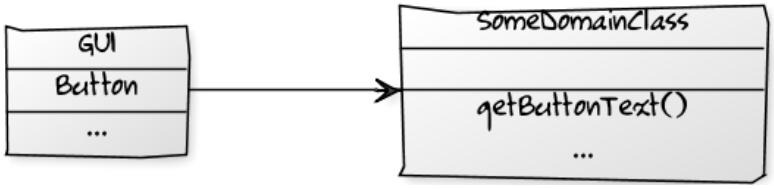
- Ein Parameter oder Rückgabewert ist gleichzeitig Datum und Kontrollflussinformation
- Ähnliche Probleme wie bei der Kontrollkopplung
- Besser: Methoden trennen

Temporal Coupling (zeitliche Kopplung)

```
myParser.load(input);
myParser.parse();
while (myParser.hasMoreTokens())
{
    System.out.println(myParser.nextToken());
}
```

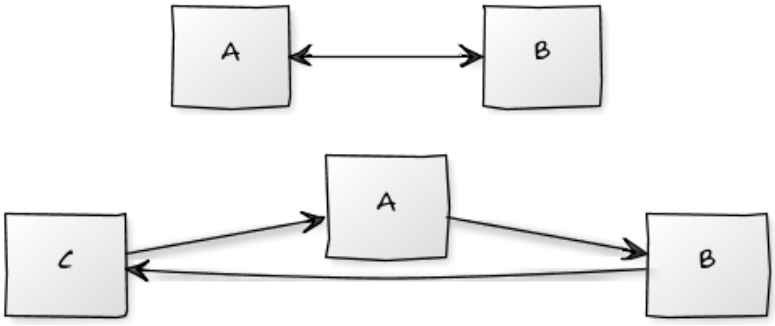
- Die Semantik eines Aufrufs hängt vom Zeitpunkt ab
  - bzw. die Reihenfolge der Aufrufe ist wichtig
- Hier muss nach load() noch parse() aufgerufen werden
- Besser parse intern aufrufen
  - Entweder in load (eager)
  - Oder in hasMoreTokens und nextToken (lazy)
- Es gibt noch weitere Ausprägungen von zeitlichen Kopplungen. Siehe hierzu [12]

Logical Coupling (logische Kopplung) [Folie 41]



- Nicht technische, sondern logische Annahmen
- Hier weiß SomeDomainClass implizit, dass es einen Button gibt
- Oft ist die Abhängigkeit nicht so offensichtlich wie hier

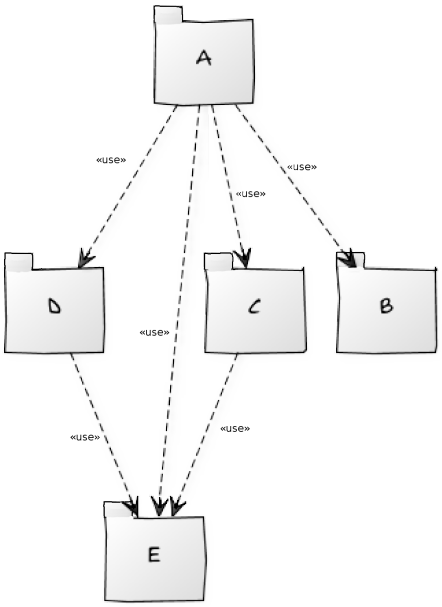
Cyclic Dependency (zyklische Abhängigkeit) [Folie 42]



- Zyklische Abhängigkeiten erzeugen starke Kopplungen
- Ein Modul des Zyklus hängt automatisch von allen anderen ab
- Bessere Lösung:
  - Abhängigkeitsbeziehungen sollten azyklisch sein (Abhängigkeitsgraph ist ein DAG)
  - Besser noch: hierarchisch (Abhängigkeitsgraph ist ein Baum)

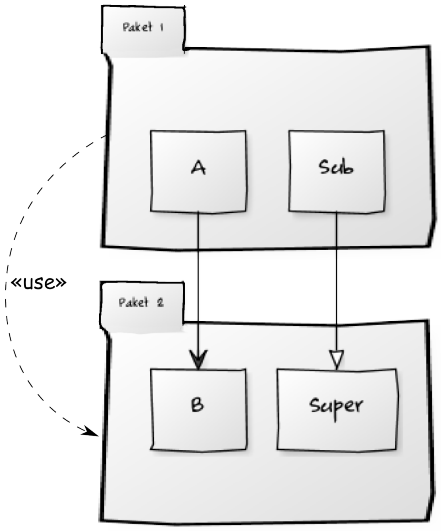
3.2. Schichten

Kommunikation beschränken (1/2) [Folie 43]



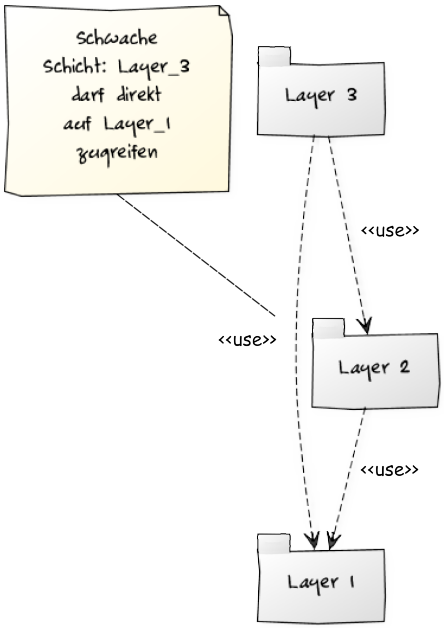
- Eine gute Architektur schränkt die Kommunikation zwischen den Modulen ein und verbietet gewisse Abhängigkeiten.
- Die Architektur definiert Regeln, die für alle Module gelten. Beispiele:
  - „Module aus Paket A dürfen auf Module aus Paket B zugreifen (aber nicht umgekehrt).“
  - „Nur Module im GUI-Paket dürfen direkt mit dem Benutzer interagieren.“
  - „Die Kommunikation zwischen Client und Server geschieht ausschließlich über RMI.“
  - „Ein Modul soll nicht gleichzeitig Datenquelle und Datensenke sein.“
  - ...
- Dadurch, dass die Abhängigkeiten gewissen Regeln unterliegen, wird die Struktur klarer und Abhängigkeiten (und damit auch RippleEffects) werden reduziert.

Kommunikation beschränken (2/2) [Folie 44]



- In UML-Klassendiagrammen zeigen alle Pfeile in Richtung der Abhängigkeit.
  - Deshalb zeigen Generalisierungspfeile in Richtung der Superklasse, obwohl anders herum „vererbt“ wird.
- Paketdiagramme zeigen größere (aber ebenfalls statische) Strukturen.
  - UML-Pakete werden zwar nicht notwendigerweise, aber meistens dennoch auf Java-Pakete abgebildet.
- Beziehungen, die zwischen Paketen gelten, müssen von den darin enthaltenen Klassen erfüllt werden.
  - Um das zu verdeutlichen, kann man Klassen in Pakete „schachteln“.

Schichten [Folie 45]

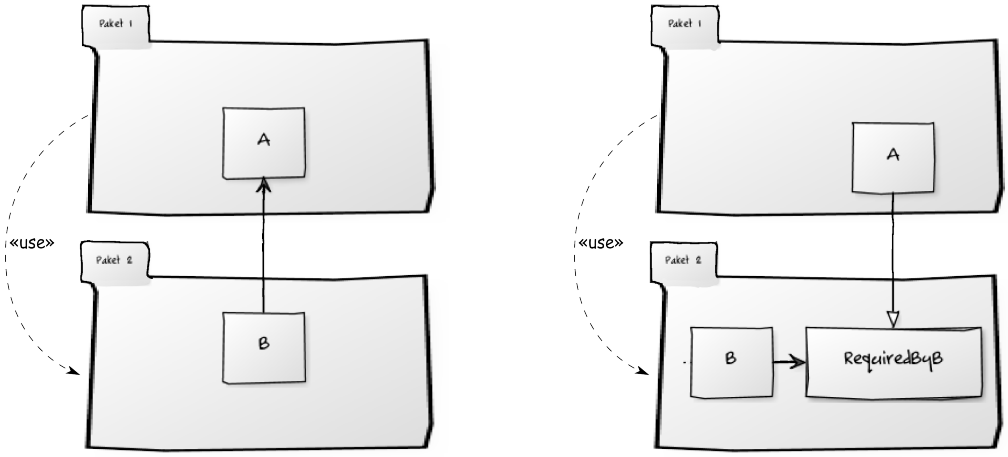


- Häufig benutzt man Schichtstrukturen um die Kommunikation zu begrenzen
  - Obere Schichten dürfen auf darunter liegende zugreifen; aber *niemals* umgekehrt
  - Strengere Variante: Nur auf die direkt darunter liegende Schicht darf zugegriffen werden
- Schichten sind Abstraktionen
  - Obere Schichten (z. B. die GUI) sind konkret und spezifisch für eine Anwendung
  - Untere Schichten werden immer allgemeiner
- Schichten sind logisch gesehen virtuelle Maschinen/Plattformen
  - Das Betriebssystem ist eine Schicht, die von der Hardware abstrahiert. So kann man als Entwickler die Schnittstelle des Betriebssystems als Plattform nutzen.
  - Die Java-VM ist eine Schicht, die vom Betriebssystem abstrahiert. So kann ein Java-Entwickler die Java-API als Plattform nutzen.

  - Auch ein Domänenmodell (natürliche Klassen, die die Anwendungsdomäne modellieren[11]) ist eine Schicht. Darüberliegende Schichten können dieses als Plattform nutzen, können also so tun, als würde die darunter liegende Maschine schon die Anwendungsdomäne kennen[12].

- Häufig werden drei Schichten verwendet. Die Abgrenzung variiert dabei ein wenig. Beispiele:
  - GUI, Anwendungslogik und Basisfunktionalität
  - GUI, Anwendungslogik und Domänenmodell
  - GUI, Anwendungslogik, Datenbank
- Schichten können selbst wieder in Schichten aufgeteilt werden.
- ...[13]

Dependency Inversion [Folie 46]



- Manchmal möchte man Aufrufe von niedrigeren in höhere Schichten realisieren, obwohl man das eigentlich nicht „darf“.

[11] So sind z. B. die Klassen Library, Book, Reader, etc. Teil eines Domänenmodells für Bibliotheken.
[12] Im Beispiel: Wir können uns vorstellen, wir hätten einen Computer, der bereits weiß, was Bücher sind.
[13] Weiteres zu Schichten, Verteilungsaspekten, etc. in den Vorlesungen *Grundlagen des Software-Enigeering* (GSE), *Softwarearchitektur verteilter Systeme* (SAVS), *Middleware für heterogene und verteilte Informationssysteme* (MWHDIS), *Service-Orientierte Architekturen* (SOA), ...

- Lösungen:
  - Callback-Funktionen/Methodenzeiger (C++, Delphi, ...)
  - Closures (LISP, Haskell, Python, Ruby, ...), Delegates (.NET)
  - Dependency Inversion (sprachunabhängig)
    - Generelles Prinzip mit verschiedenen Ausprägungen
    - Eine der Ausprägungen ist das Observer-Pattern (siehe [20])
- Dependency Inversion: Abhängigkeiten lassen sich umkehren, wenn man eine zusätzliche Abstraktion (Superklasse bzw. Interface) einführt.
- Dependency Inversion kann man zu einem Prinzip machen: DIP (Dependency Inversion Principle) [17] „Abstractions should not depend upon details. Details should depend upon abstractions“

4. Patterns

Über Patterns lässt sich viel sagen. Sie sind sehr hilfreich und auch für das SEP/MP kann man einige verwenden. Aufgrund der Kürze der Zeit kann ich hier nur ganz grob darauf eingehen. Für eine detailliertere Beschreibung verweise ich auf [20].

Patterns [Folie 48]

Definition Pattern
Ein „Muster“ oder „Pattern“ ist eine häufig anzutreffende, für gut befundene Lösung zu einem wiederkehrenden Problem.

- Das ist meine Definition. Es gibt noch diverse andere. In [20] liste ich ein paar davon auf.
- Patterns sind so etwas wie fertig ausbalancierte Musterlösungen zu wiederkehrenden Problemen.
  - Statt sich also selbst eine Lösung einfallen zu lassen und dabei die einzelnen Kräfte[14] auszubalancieren, kann man einfach ein bestehendes Muster nehmen.
  - Trotzdem muss man das Muster oft anpassen.
  - Trotzdem muss man die Prinzipien nutzen um heraus zu finden, ob das Muster hier passt.
  - Trotzdem sind Patterns nicht unproblematisch, wenn man sie überverwendet.
  - Trotzdem sind Patterns hilfreich.
- Patterns werden in Musterkatalogen dokumentiert und zu Mustersprachen verknüpft.
- Patterns transportieren Wissen (von Leuten, die wissen, wie man ein Problem löst zum Rest der Menschheit)
- Patterns dienen der projektinternen Kommunikation; sie geben Lösungen Namen
  - Statt „Das Konzept bei dem die eine Klasse die Anfragen entgegen nimmt und sie ggf. an eine andere mit gleichem Interface weiterreicht.“ sagt man „Proxy Pattern“ [8] [27] und jeder weiß, was gemeint ist.

Patterns gibt es auf diversen Abstraktionsebenen und für alle möglichen Arten von Problemen:

[14] im Umfeld von Patterns spricht man häufig auch von „Kräften“ statt von Prinzipien

Unterschiedliche Patterns [Folie 49]

- Architekturpatterns (architectural patterns) bzw. „Architekturstile“ (architectural styles)[15]
- Analysepatterns (analysis patterns)
- Entwurfsmuster (design patterns)
- Idiome (idioms)
- Anti-Patterns
- Code Smells
- ...

Patterns nutzen [Folie 50]

1. Problem haben
2. Sich an Pattern erinnern
3. Pattern nachschlagen
4. Prüfen, ob Pattern passt
5. ggf. Pattern anpassen
6. Pattern anwenden
7. Freuen

- Man muss Patterns nicht auswendig lernen, aber man muss sich daran erinnern können, dass es sie gibt. Einen Überblick über die wichtigsten Patterns zu kriegen, ist also vorteilhaft.
- Patterns definieren oft Rollen. Die Klassen, die in den Diagrammen der Pattern-Beschreibungen gezeigt werden, werden also zuerstmal auf bestehende Klassen abgebildet. Nur, wenn das nicht geht, werden neue Klassen eingeführt.
- Weitere Infos zu Patterns und Links zu wichtigen Musterkatalogen gibt es in [20].

[15] siehe hierzu Vorlesungen GSE und SAVS

Fazit [Folie 51]

- 3 Hilfsmittel um Software zu entwerfen
  - Szenarien, Prinzipien und Patterns
- Nützliches Wissen
  - RippleEffects, Bindung und Kopplung
  - Abhängigkeiten und Schichten

A. Anhang

A.1. Aufgaben

Aufgabe 1 – Datum [Folie 53]

Aufgabe 1

- Bibliothek für Datums- und Zeitwerte
- Für den „alltäglichen Gebrauch“ einsetzbar

Varianten der Aufgabe: Statt „Für den ‚alltäglichen Gebrauch‘ einsetzbar“: Nutzung für Software für a) Historiker b) Astronomen c) Callcenter d) Regelungssystem mit harten Echtzeitanforderungen, e) Navigationssystem, f) Atomuhr.

Hinweise zu den Randbedingungen der Varianten: Siehe Wikipedia-Artikel zu Julianischer Kalender, Gregorianischer Kalender (insbesondere: Reformjahr 1582 und Jahresbeginn), Liste der Kalendersysteme, Schaltjahr, Schaltsekunde, GPS-Zeit, Sommerzeit, Hochsommerzeit, Datumsgrenze und Jahr null. Sehr hilfreich ist auch [11].

Achtung: Die Aufgabe ist *sehr* schwer. Zumindest, was die Varianten angeht. Die Lösung der Java API halte ich im übrigen für schlecht (Zusatzaufgabe: Warum? Was kann man daran kritisieren? Wie ist es dazu gekommen[16]?). Es geht hier auch nicht darum, eine perfekte oder auch nur annehmbare Lösung zu finden. Es geht nur darum, sich Gedanken darüber zu machen, was man alles beachten muss, welche Szenarien man braucht und wie das die Lösung verändert.

Aufgabe 2 – Genealogie [Folie 54]

Aufgabe 2

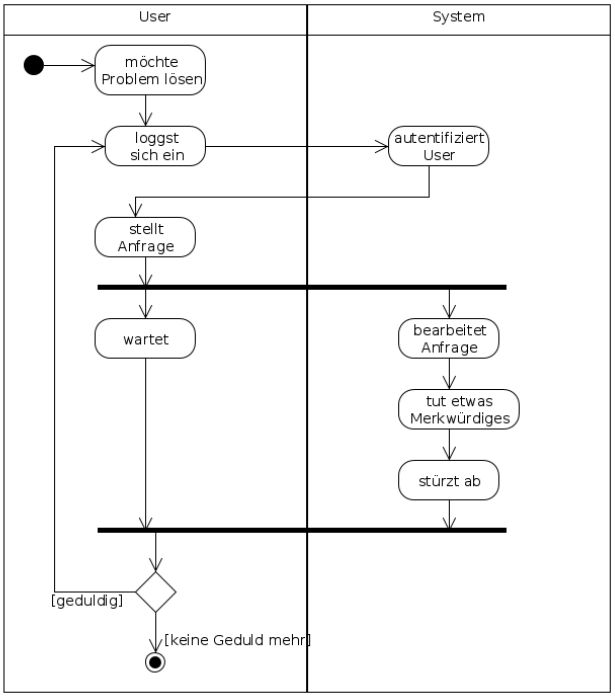
- Software für Genealogie (Ahnenforschung)
- Die Software soll Stammbäume, Verwandtschaftsbeziehungen, etc. graphisch darstellen, durchsuchbar machen, ...
- Für jede Information (Geburtsdatum, etc.) soll der User eine Quelle angeben können
- Außerdem ist zu beachten, dass manche Daten vage sein können

[16] „Die Java-Entwickler waren blöd“, ist die falsche Antwort.

Auch diese Aufgabe ist (im Detail) nicht ganz einfach (wenn auch etwas einfacher als die Datums-Aufgabe). Vor allem aber ist die aufwendig. Es geht aber auch hier wieder nicht darum, eine vollständige, korrekte und gute Lösung zu finden. Vielmehr geht es darum, den richtigen Ansatz zu finden. Details sind nur dann wichtig, wenn man wirklich so eine Software schreiben will.

A.2. Aktivitäts- und Zustandsdiagramme

Aktivitätsdiagramme [Folie 55]



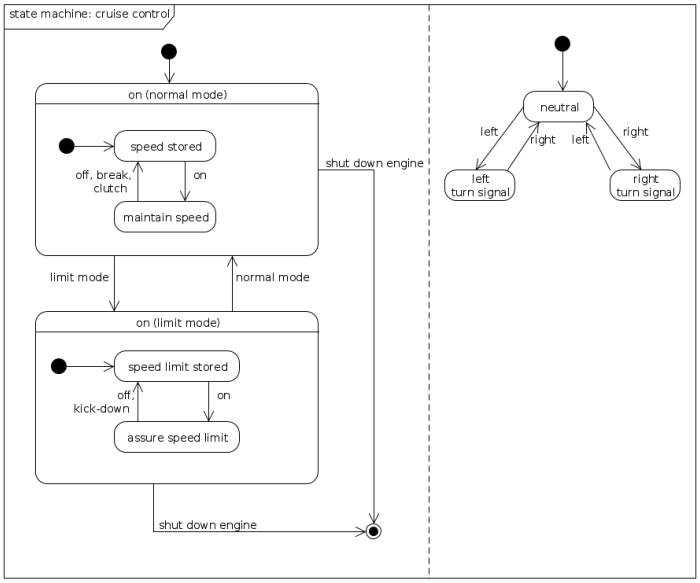
- Aktivitätsdiagramme zeigen Abläufe und sind verallgemeinerte (Kontroll-)Flussdiagramme
- Seit UML 2 haben Aktivitätsdiagramme eine Tokensemantik und sind somit verwandt mit Petrinetzen ⇒ Nebenläufigkeit modellierbar
- Knoten sind Aktionen, Kanten sind Kontrollflüsse
- Aktivitätsdiagramme eignen sich, um Algorithmen zu definieren oder die Behandlung von Szenarien generisch zu beschreiben (also nicht nur einen Durchlauf (trace))
- Durch Aktivitätsbereiche („swim lanes“) lässt sich leicht zeigen wer welche Aktion ausführt ⇒ Interaktion mit dem System modellierbar
- Aktivitätsdiagramme haben noch viele weitere Syntaxelemente. Siehe [24] und [5] für Details.

Künstliche Modelle [Folie 56]

Intuitive Modellierung ist nicht die einzige Möglichkeit

- Neben der intuitiven Modellierung (Modellprinzip) kann es manchmal hilfreich sein, Gegebenheiten mit künstlichen Modellen zu beschreiben
- Diese Modelle haben i. d. R. besondere Eigenschaften (Leichte Übertragbarkeit in Code, maschinelle Analysierbarkeit, beweisbare Korrektheit, etc.) und sind manchmal auch einfacher als natürliche Modelle.
- Zustandsmaschinen sind solche Modelle

Zustandsdiagramme [Folie 57]



- Zustandsdiagramme beschreiben endliche Automaten (siehe Theorievorlesungen)
- Knoten sind Zustände, Kanten sind Aktionen

- David Harel erweiterte die aus der Theorie bekannte Notation, um Sachverhalte bequemer, einfacher und übersichtlicher modellieren zu können [10]
  - verschachtelte Zustände, parallele Automaten, weitere Pseudozustände wie Kreuzungen, Historien, ...
  - Diese erweiterte Notation wurde in die UML aufgenommen (siehe [24], [5])
- Ein Zustand kann entweder ein Datum sein (einfacher Zustand) oder das Systemverhalten grundlegend beeinflussen (solche Zustände nennt man auch „Modi“ (bzw. englisch *modes*))
  - Einfache Zustände werden i. d. R. als Variablen umgesetzt
  - Modi partitionieren das Systemverhalten, sodass man separate Klassen verwenden kann (State Pattern). Das kann vorteilhaft sein, um den Code zu partitionieren und so übersichtlicher und wartbarer zu machen.
- Hinweise zur Modellierung von Zustandsmaschinen finden sich in [14]
- Zustandsmaschinen lassen sich auf verschiedene Arten in Code überführen. Teilweise sogar automatisch.
  - Informationen hierzu finden sich in [14].
  - Neben den in [14] genannten Möglichkeiten gibt es in Java noch eine weitere Möglichkeit der Realisierung von Zustandsautomaten: siehe [4]
  - Außerdem kann man Zustandsautomaten durch Erstellung/Nutzung eines Frameworks in Code überführen (siehe [24])

Literatur [Folie 58]

Literatur

[1] Bass, Len, Paul Clemens und Rick Kazman: *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 2 Auflage, 2003.

[2] Bock, David: *The Paperboy, The Wallet, and The Law Of Demeter*. http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf.

[3] Duszynski, Slawomir, Jens Knodel und Mikael Lindvall: *SAVE: Software Architecture Visualization and Evaluation*. In: *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, CSMR '09, Seiten 323–324, Washington, DC, USA, 2009. IEEE Computer Society.

[4] Erb, Benjamin: *Zustandsautomat in Java mithilfe des State Patterns*. http://www.ioexception.de/2009/06/21/zustandsautomat-in-java-mithilfe-des-state-patterns/, Jun 2009.

[5] Fakhroutdinov, Kirill: *uml-diagrams.org*. http://www.uml-diagrams.org.

[6] Fowler, Martin: *Dealing with Roles*. http://martinfowler.com/apsupp/roles.pdf, Jul 1997.

[7] Fowler, Martin: *TechnicalDebt*. http://martinfowler.com/bliki/TechnicalDebt.html, Feb 2009.

[8] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] Garlan, David und Mary Shaw: *An Introduction to Software Architecture*. In: *Advances in Software Engineering and Knowledge Engineering*, Seiten 1–39. World Scientific Publishing Company, 1993. http://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/intro_softarch.pdf.

[10] Harel, David: *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, 8:231–274, 1984. http://www.wisdom.weizmann.ac.il/~harel/SCANNED.PAPERS/Statecharts.pdf.

[11] Horn, Torsten: *Java Date und Calendar*. http://www.torsten-horn.de/techdocs/java-date.htm.

[12] Hunt, Andrew und David Thomas: *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 1999.

[13] Hunt, Andrew und David Thomas: *The Art of Enbugging*. IEEE Software, 20:10–11, Jan/Feb 2003. http://www.ccs.neu.edu/research/demeter/related-work/pragmatic-programmer/jan_03_enbug.pdf.

[14] Jackson, Daniel: *Elements of Software Construction*. http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-elements-of-software-construction-fall-2008/lecture-notes/, 2008. Vorlesung am MIT.

[15] Liggesmeyer, Peter: *Software-Entwicklung 2*. http://agde.informatik.uni-kl.de/teaching/se2/.

[16] Lindvall, Mikael und Jens Knodel: *Software Architecture Visualization and Evaluation*. http://www.fc-md.umd.edu/save/.

[17] Martin, Robert C.: *DIP: The Dependency Inversion Principle*. http://www.objectmentor.com/resources/articles/dip.pdf.

[18] Martin, Robert C.: *SRP: The Single Responsibility Principle*. http://www.objectmentor.com/resources/articles/srp.pdf, Feb 1997.

[19] Rehn, Christian: *Tag: Daumenregeln*. http://www.christian-rehn.de/tag/daumenregeln/.

[20] Rehn, Christian: *Patterns*. http://www.christian-rehn.de/2011/05/21/patterns/, Mai 2011.

[21] Rehn, Christian: *Softwareentwicklungs-Prinzipien: Eine Übersicht*. http://www.christian-rehn.de/2011/05/14/softwareentwicklungs-prinzipien-eine-ubersicht/, 2011.

[22] Rehn, Christian: *The Tradeoff Game*. http://www.christian-rehn.de/2011/04/23/the-tradeoff-game/, 2011.

[23] Rogers, Paul: *Encapsulation is not information hiding*. http://www.javaworld.com/javaworld/jw-05-2001/jw-0518-encapsulation.html, 2001. JavaWorld.com.

[24] Rupp, Chris, Stefan Queins und Barbara Zengler: *UML 2 glasklar*. Hanser, 2007.

[25] Wagenführ, Dominik: *Objektorientierte Programmierung: Teil 1 – OOP in der Praxis*. freiesMagazin, 3:6–10, 2012. http://www.freiesmagazin.de/freiesMagazin-2012-04.

[26] Wagenführ, Dominik: *Objektorientierte Programmierung: Teil 2 – Die richtige Strategie*. freiesMagazin, 4:5–8, 2012. http://www.freiesmagazin.de/freiesMagazin-2012-04.

[27] Wikipedia: *Stellvertreter (Entwurfsmuster) — Wikipedia, Die freie Enzyklopädie*, 2011. http://de.wikipedia.org/w/index.php?title=Stellvertreter_(Entwurfsmuster)&oldid=90992901 [Online; Stand 2. Mai 2012].

[28] Wikipedia: *Pareto-Optimum — Wikipedia, Die freie Enzyklopädie*, 2012. http://de.wikipedia.org/w/index.php?title=Pareto-Optimum&oldid=101987377 [Online; Stand 2. Mai 2012].

[29] Wikipedia: *Strategie (Entwurfsmuster) — Wikipedia, Die freie Enzyklopädie*, 2012. http://de.wikipedia.org/w/index.php?title=Strategie_(Entwurfsmuster)&oldid=102549483 [Online; Stand 2. Mai 2012].

[30] Zakas, Nicholas C.: *Understanding technical debt*. http://www.nczonline.net/blog/2012/02/22/understanding-technical-debt/, Feb 2012.

Lizenz [Folie 59]