Foliensatz 1: Einführung, Motivation, Organisation

In einer komplexen Situation ist es fast immer notwendig, sich nicht nur um ein Merkmal der Situation zu kümmern, also ein Ziel anzustreben, sondern man muss viele Ziele gleichzeitig verfolgen. Wenn man aber mit einem komplizierten, vernetzten System umgeht, so sind die Teilziele kaum je ganz unabhängig voneinander. Es ist vielmehr oft der Fall, dass Teilziele in einem kontradiktorischen Verhältnis zueinander stehen. (D. Dörner, 1989)

Folgen komplexer Systeme für Software-Projekte

- Genaue Analyse zu Beginn
- Projekte mit vielen Mitarbeitern brauchen eine Projektorganisation mit Festlegung von Zuständigkeiten und Terminen
- Unabhängig voneinander arbeitende Gruppen brauchen eine genaue Schnittstelle
- ➔ Unmöglich ein komplexes System in einem Wurf zu bauen

Spezifikation

1. Planungsphase

- prüfen ob Produkt durchführbar ist, entsprechend im Lastenheft festgehalten

Lastenheft (enthält Basisanforderungen): Zielbestimmung, Produkteinsatz, Produktübersicht (graphischer Überblick), Produktfunktionen (nummeriert für spätere Bezüge), Produktdaten, Produktleistungen (nummeriert), Qualitätsanforderungen (Benutzbarkeit, Effizienz), Ergänzungen

Pflichtenheft (enthält fachliche Anforderungen i.d.R. Konkretisierung der Lastenheft-Inhalte)
(Struktur: Zielbestimmung (Musskriterien, Wunschkriterien, Abgrenzungskriterien)
Produkteinsatz (Anwendungsbereiche, Zielgruppen Betriebsbedingungen)
Produktübersicht (wie Lastenheft)
Produktfunktionen (Konkretisierung & Detaillierung gegenüber Lastenheft)
Produktdaten/Produktleistungen/Qualitätsanforderungen
Benutzungsoberfläche: Layout, Style Guides, Zugriffsrechte
Nicht funktionale Anforderungen: Gesetze Normen, Revisionsfähigkeit
Technische Produktumgebungen/Spezielle Anforderungen: Was wird an Soft-/Hardware benötigt, Unterschiede Entwicklungs- /Zielmaschine
Gliederung in Teilprodukte/Ergänzungen)

2. Use Cases

- Soll Leistung als „erwartetes Verhalten“ beschreiben → modelliere Systemverhalten
- Beschreibt noch nicht, wie das System das gewünschte Verhalten implementiert
- Jeder Use Case beschreibt einen Gewinn für den jeweiligen Benutzer

Foliensatz 2: Design, Architektur, Netzwerke & Marshalling

Responsibility-Driven Design: Jede Komponente übernimmt Zuständigkeiten (responsibilities) und diese kollaborieren, um die Gesamtfunktionalität herzustellen
Arten von Zuständigkeiten: Etwas zu wissen, Etwas zu tun, Etwas zu entscheiden (nach außen sichtbar)

```
ServerSocket serverSocket = new ServerSocket(7000);
while (true) {
  com = new Comm(serverSocket.accept());
  try {
    while (true) {
      Message msg = com.recvMessage();
      if (msg.getClass() == Login.class)
          serveLogin((Login) msg);
    }
  } catch (EOFException eof) { }
  com.close();
}
```

Single Responsibility Prinzip: Kleine Komponenten, wenige übersichtliche Methoden in Schnittstelle, Jede Komponente besitzt eine Haupt-Zuständigkeit

Netzwerkkommunikation in eine eigene Komponente gliedern, die jeweils nur die Methoden zur Kommunikation bereitstellen. In der Komponente wird dann Marshalling/Serialisierung (Objekt-Daten werden als Byte-Folge kodiert) betrieben. (DataOutputStream, DataInputStream für primitive Typen, Objekte werden über sogenannte Header/Body Übertragungen ermöglicht, verallgemeinert im Composite Message Pattern)

Refactoring: Verbesserung der Code-Struktur ohne Änderung der Funktionalität

Architektur-Sicht abstrahiert über die Details der Implementierung und konzentriert sich auf das Verhalten und die Interaktion von Elementen. Programmierer konzentrieren sich auf „ihre“ Komponenten

Foliensatz 3: Grundlegende Patterns

Was sind Patterns? Sie halten ausgereifte Lösungen fest und repräsentieren Anstrengungen im Re-Design und Re-Coding, mit denen die Software flexibler/wiederverwertbar wird.

Composite Pattern

Interface für Clients, das alle notwendigen Operationen deklariert, bei neuen Operationen in Interface deklarieren, dann wird automatisch ersichtlich wo die Methoden noch Implementiert werden müssen, Durchlauf durch Baum braucht Unterstützung (z.B. bei Formeln, Widget-Bäume -> Swing), Clients sind unabhängig von konkreten Objekten.

Erreicht: Rekursion sehr elegant, Erweiterbarkeit um neue Knotentypen, Clients arbeiten nur mit Basisklasse

Probleme: Menge der Operationen aufwendig zu erweitern (Teillösung Visitor Pattern)

Observer Pattern

Hilft in folgenden Situationen:

- Änderungen an einem Objekt erfordern Änderungen an einer unbekannten Anzahl anderer Objekte
- Ein Objekt soll andere Objekte benachrichtigen, ohne die konkreten Klassen zu kennen
- Zustand mehrere Objekte muss konsistent gehalten werden
- Zusammenhängende Abstraktion besteht aus zwei voneinander abhängigen Teilen

```
public interface Observer extends EventListener {
  void update(Subject s);
}
```

```
public class Subject {
  private EventListenerList listeners = new EventListenerList();
  private int state;
  public void modify(int i) {
    state += i;
    notify();
  }
  public void addObserver(Observer l) { listeners.add(Observer.class, l); }
  public void removeObserver(Observer l) { listeners.remove(Observer.class, l);}
  private void notify() {
    for (Observer l : listeners.getListeners(Observer.class)) {
      l.update(this);
    }
  }}
```

Variante Push-Modell:

- Es werden nur die geänderten Daten übertragen
- Effizient und erspart unnötige Berechnung

Adapter

Konvertiert die Schnittstelle einer Klasse in eine andere Schnittstelle, die von Clients erwartet wird. Das Adapter-Pattern lässt Klassen zusammenarbeiten, die dies sonst wegen inkompatibler Interfaces nicht könnten. (z.B. Bibliotheksklasse muss mit einer neuen, unerwarteten anwendungsspezifischen Klasse kollaborieren, Schnittstelle zwischen Subsystemen unterschiedlicher Teams)

Object Adapter: Adapter hält Referenz auf Adaptee, funktioniert für Subklassen von Adaptee (schwierig)

Class Adapter: Adapter erbt von Adaptee, funktioniert nur für eine Klasse (keine Subklassen), muss direkt bei Instanziierung angelegt werden

Layer

Einteilung von Komponenten nach Abstraktionsgrad (Abhängigkeit von Hardware, Wahrscheinlichkeit von Änderungen, Komplexität, …), Komponenten bilden Schichten und kommunizieren nur mit der übergelegenen oder der darunterliegenden. (z.B. Bildschirmdarstellung, Applikationslogik, Netzwerke)

- dadurch Problemloser Austausch von Komponenten möglich
- mögliche Standardisierung von Verhalten
- Zuständigkeiten klar verteilt, Abhängigkeiten auf lokale Nachbarn beschränkt
- Effizienzverlust/Overhead

Foliensatz 4: Java-Bibliothek

- Kommunikation mit der Umgebung: I/O Dateien, Image Reader, Netzwerke, XML-Verarbeitung
- Standard-Datenstrukturen: Linked List, Vector, HashMap, Stack, Queue, Iterator, Enumeration
- Hilfsklassen: Logging, Formatter, Reguläre Ausdrücke
- Nebenläufigkeit: Locks &Semaphoren (Objekte schützen), Blocking Queues
- Swing & UIs
- Nie Referenz rausgeben auf die Datenstruktur rausgeben (diese könnten sonst den Inhalt ändern)
- Work-Queues: explizite Festlegung der Reihenfolge mit Stack/Queue ermöglicht debugging
- HashMaps: Speichert Zusatzinformationen auf die häufig zugegriffen werden muss

Foliensatz 5: Umgebung (Eclipse, Debugger, SVN)

- Bei Fehlerhaftem Code Inspektion des laufenden Programms mit Debugger
- Test schreiben, die den Bug auslöst, damit schnell testbar
- **Debugger**: Stapel der Methodenaufrufe, Variableninhalte, Geworfene Exceptions, …
- **Hot Code Swap**: Austausch von Methoden ohne Neustart
- Tests zu Beginn jeder Methode mit assert(b), um auf gültige Eingaben zu prüfen, wird dann nur mit JVM-Flag –ea getestet und verlangsamt die normale Ausführung nicht

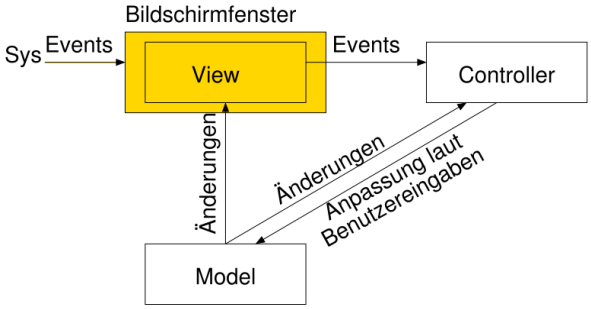
Foliensatz 6: GUI & Swing

- Design legt großen Wert auf Flexibilität (Anpassbarkeit der Darstellung)
- Hierarchischer Aufbau (Composite Pattern)
- **Layout-Manager** entscheidet über Position der Kinder (GridBagLayout)

- Üblicherweise single-threaded, Zugriff auf Widgets nur in Event-Thread (vermeidet Threading-Probleme (Synchronisation, Deadlocks)
- Interaktionen über ActionListener/DocumentListener/… der alle relevanten Objekte informiert
- Fasse logisch zusammengehörige Widget-Gruppen in einem neuen Widget zusammen

Foliensatz 7: Model-View-Controller

Verteile die Funktionalität für die Anzeige und für die Datenhaltung strikt auf unterschiedliche Komponenten, so dass die Datenhaltung für sich alleine Sinn macht und unabhängig vom GUI implementiert und getestet werden kann.



(Betriebssystem liefert Events immer an Views aus.)

Model

- Implementiert funktionalen Kern der Anwendung (Datenstrukturen, Algorithmen)
- Kann mit verschiedenen Views verwendet werden (GUI unabhängig)
- Ändert Daten gemäß Business-Logik (Zugriff auf GUI nur durch Observer-Interface)

View

- Bietet Fenster für graphische Darstellung, registriert sich beim Model als Observer
- reagiert auf Änderungen des Modells
- Erzeugt Controller (createController()) und leitet Events im Fenster an Controller weiter

Controller

- Verarbeitet die Events indem es Schnittstellenfunktionen im Modell aufruft
- Optional: Ändert Darstellung in der View, Beobachtet Modell (→ Observer)

Variante: Document-View

- View und Controller sind eng gekoppelt ➔ Lege View und Controller zusammen zu View
- Bringt reduzierte Komplexität/Kommunkation zwischen Objekten, aber schlechtere Wartbarkeit

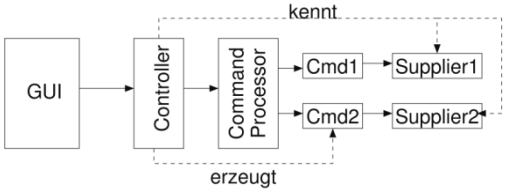
Foliensatz 8: Undo & Command Processor

- Kapsele Änderungswünsche des Benutzers als Objekte ein, um erweiterbare Benutzerschnittstellen und Undo/Redo-Funktionalität zu erhalten
- Objekte enthalten selbst die auszuführende Aktionen als Methode
- Wichtig ist dass undo/redo stets zum gleichen Zustand führt (für alle Fälle!)

Command Processor Pattern

Trennt die Anforderung eines Services von dessen Ausführung. Die Command Processor Komponente verwaltet Anforderungsobjekte und entscheidet über ihre Ausführung und Umkehrung.

Controller: Anlaufstelle für Service-Anforderungen aus der GUI, übersetzt diese Anforderungen in Command Objekte und übergibt Commands an Command Processor
Command Processor: Führt Command Objekte aus, verwaltet History für Undo/Execute



Bewertung: ermöglicht einfache History-Verwaltung, klare Trennung von Controller und Model, Erweiterbarkeit ohne Änderung an bestehendem Code, unabhängig von Zugriffsart (z.B. Skripting)

Foliensatz 9: Multi-Threading

Ziel ist es das „Einfrieren" der GUI zu vermeiden durch lang dauerende Ausführung und dem Nutzer stets mitteilen, dass noch berechnet/runtergeladen/etc. wird. Als Ansatz dafür wählt man mehrere Threads die Synchron ausgeführt werden, was die Komplexität aber erhöht!!

Swingworker (abstract): ermöglicht Hintergrundausführung und beachtet Threading und Synchronization, doInBackground(), nach Abschluss wird done() ausgeführt, dazwischen publish(e)

Nebenläufige Ausführung auf der Maschine: Scheduler entscheidet wer wie viel Rechenzeit bekommt und kann diese jederzeit wieder entziehen auch mitten in Anweisungen. Problematisch wird es dabei wenn einzelne Threads aufeinander warten müssen oder ungünstig auf dieselben Daten zugreifen und damit inkonsistente Operationen durchgeführt werden.

Java-Library

- Synchronisiert: Hashtable, ConcurrentHashMap, Vector
- Nicht synchronisiert (schneller): HashMap, HashSet, ArrayList, LinkedList

Das Producer/Consumer-Pattern

Ein (oder mehr) Producer stellen Daten in Queue und ein (oder mehr) Consumer holen Daten aus Queue

Zwei Threads greifen synchronisiert auf Queue zu. Schreiben wartet bis Platz ist und lesen bis Daten vorhanden sind. → Datenaustausch zwischen Threads gesichert

Foliensatz 10: Patterns Proxy, Iterator, Visitor

Proxy Pattern

Erzeuge Platzhalter für ein anderes Objekt, um den Zugriff auf dieses Objekt zu kontrollieren.

Remote Proxy: Lokales Objekt, das Methodenaufrufe über das Netzwerk auf ein anderes weiterleitet
Virtual Proxy: Erzeugt teure Objekte wie Bilder ondemand, also erst wenn sie benötigt werden.
Protection Proxy: Prüft Zugriffsrecht vor Weiterleitung der Methodenaufrufe
Smart Reference: Ersatz für normale Objekt-Referenz, der weitere Aktionen ausführt (z.B: Zählen der Referenzen & automatische Objektfreigabe)

Proxy - Struktur

Subject: Gemeinsame Schnittstelle von Proxy & Realsubject
RealSubject: Das Objekt, das der Proxy repräsentiert
Proxy: Hält Referenz auf ein RealSubject, bietet gleiche Schnittstelle wie RealSubject an, kann Erzeugung & Zerstören des RealSubject übernehmen ➔ einsetzbar wie RealSubject

Proxy – Konsequenzen

- Proxies führen eine Indirektion in den Objekt-Zugriff ein
- Remote Proxies verbergen physikalischen Ort im Netzwerk

```
class PersonProxy implements IPerson {
  private int id;
  private Person p;
  public PersonProxy(int id) {
    this.id = id;
  }
  public String getName() {
    if (p == null) loadPerson();
    return p->getName();
  }

  private void loadPerson() {
    com.sendMessage(new FetchPersonMsg(id));
    PersonMsg res = (PersonMsg)com.receiveMessage();
    p = new Person(res.getName());
  }
}
```

4. Mai 2012

Visitor Pattern

Repräsentiert eine Operation auf einer Objekt-Struktur. Es ist möglich, eine neue Operation zu definieren, ohne die Klassen der Elemente der Struktur zu ändern. Eine rekursive Operation ist damit in einer Klasse gekapselt.

```
public abstract class BinNode extends Node {
  public void accept(NodeVisitor v) {
    left.accept(v);
    right.accept(v);
  }
}
public class AddNode extends BinNode {
  public void accept(NodeVisitor v) {
    super.accept(v); // Rekursion
    v.visitAdd(this);
  }
}
public class NumNode extends Node {
  public void accept(NodeVisitor v) {
    v.visitNum(this);
  }
}
```

Visitor – Konsequenzen

- Implementierung neuer Operationen wird vereinfacht. (accept())
- Trennung logisch unzusammenhängender Funktionalität
- Hinzufügen neuer Elemente deutlich erschwert!!!

Iterator

Stellt einen Weg bereit, die Elemente eines zusammengesetzten Objekts sequentiell zu durchlaufen, ohne dessen Repräsentation offenzulegen. (z.B. Vector, ArrayList, Knoten eines Baumes)

Iterator – Konsequenzen

- Durchlaufstrategie kann flexibel geändert werden
- Vereinfachte Schnittstelle des zusammengesetzten Objektes
- Parallele Durchläuf möglich (Iterator hält internn „aktuelle“ Position)

Foliensatz 11: Design by Contract

- Notwendige Voraussetzung für Arbeitsteilung, Wartbarkeit, Testen & Debuggen, Dokumentation
- Beschreibung der Zusammenarbeit von Modulen, Korrektheit von Programmen
- **Der konkrete Zustand**: Lokale Variablen, Parameter der laufenden Methode, Feldinhalte von Objekten, Statische Felder von Klassen, Array-Elemente

Information Hiding (IH)

Prinzip: Module (Objekte) verbergen ihre Repräsentation (Felder sind private, Datenstrukturen nicht von außen erreichbar, gibt keine Referenzen auf Hilfs-Objekte heraus)
Ziel: Interne Verarbeitung kann jederzeit geändert werden

Assertions (Zusicherungen)

- Logische Aussagen über den Zustand, die einem Programmpunkt zugeordnet sind. Immer, wenn dieser Programmpunkt erreicht wird, ist die Aussage wahr.
- Am Anfang & Ende einer Methode bleiben die Aussagen über Parameter in der Methode Annahmen.
- Aufrufer kann nicht alle notwendigen Assertions sicherstellen (Information Hiding)
- Public-Methoden nehmen Invariante zu Beginn an & garantieren sie nach Ausführung
- Konstruktoren initialisieren die Felder der Klasse so, dass die Invarianten gelten

Design von Verträgen

- Methoden müssen genügend „wissen“, um zu arbeiten
- Ergebnis so beschrieben, dass Aufrufer es verwenden kann
- Vertrag darf keine Interna preisgeben, Aufrufer kann Repräsentation nicht kontrollieren (IH)

Programmierstile

- **Toleranter Stil** (defensives Programmieren): Methoden haben schwache Vorbedingungen, prüfen für aktuelle Argumente, ob sie anwendbar sind, Signalisieren Fehlschlag mit Default Ergebnis
- **Demanding Style**: Methoden haben gerade so starke Vorbedingungen, dass sie ohne Prüfung der Argumente arbeiten können, sie machen möglichst starke Aussagen über Ergebnis
  (effizienter da keine Laufzeit-Überprüfung, lesbarer da Konzentration auf eigentliche Funktionalität, wartbarer da es eine eindeutigere Dokumentation ergibt, weniger Komplex)

Zwei Prinzipien für den Demanding Style:

- **Non-redundancy Principle**: Methoden prüfen niemals ihre Vorbedingungen
- **Reasonable Recondition Principle**: Vor- & Nachbedingung sind aus offizieller Dokumentation verständlich, Notwendigkeit der Vorbedingung ergibt sich aus der Spezifikation
- **Precondition Availability Principle**: Aufrufer muss die Vor-/Nachbedingung verstehen können

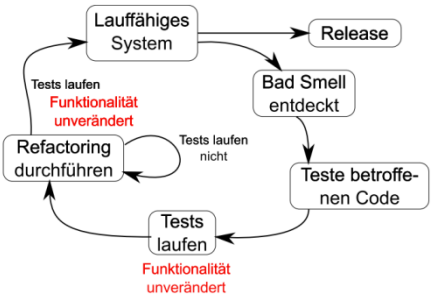
Design an Systemgrenzen

- Rechne nicht damit, dass externe Programme & Benutzer Verträge einhalten ➔ Toleranter Stil
- Beschreibe erwartete Eingaben & garantierte Ausgaben (prüfe ALLE Eingaben des Systems)

Foliensatz 12: Clean Code & Refactoring

Jeder Routine tut das, was man erwartet, offensichtliche Logik, minimale Abhängigkeiten

Namenswahl: sollte die Intention wiederspiegeln, muss man aussprechen können, vermeide Abkürzungen, verwende für ein Konzept immer dasselbe Wort, Klassennamen sollten Nomen, Methodennamen Verben/Hilfssätze sein



Methoden: Schreibe kurze Methoden, jede Methode tut eine Sache, jede Methode arbeitet auf einer Abstraktionsebene, verwende Exceptions statt Rückgaben mit Error-Code, Vermeide zu viele Argumente

Klassen: verwende encapsulation, schreibe kleine Klassen, beachte das Single Responsibility Principle (Beschreibungen von Klassen enthalten kein „und“)

Foliensatz 13: UML Sequence Charts & State Charts

Sequenzdiagramme beziehen sich auf Objekte, nicht auf Klassen wie die meisten anderen UML-Diagrammtypen. Abläuft sind nur möglich, nicht zwingend.

Events: ActionsListener, Aunkft von Daten über das Netzwerk, Multithreading
Action: Eine Aktion ist ein atomarer Berechnungsschritt. (kann nicht von Events unterbrochen werden)
Activity: Eine Aktivität ist eine fortdauernde, nicht-atomare Berechnung
Zustand: Ein Zustand ist eine Bedingung oder eine Situation im Lebenszyklus eines Objekts, in dem es eine bestimmte Bedingung erfüllt, eine bestimmte Aktivität ausführt oder auf ein Event wartet.
Transition: Gibt zwischen zwei Zuständen an, dass ein Objekt, das im ersten Zustand ist, beim Eintreffen eines bestimmten Events (event trigger) und Erfülltsein einer bestimmten Bedingung (guard condition) in den zweiten Zustand übergeht und dabei bestimmte Aktionen (action) ausführt

Events bilden „Zustand“ ab. Abstrakter Zustand ⇔ Bestimmte Belegung der Felder

Eigenschaften von Zuständen

- **Entry action** wird ausgeführt, wenn der Zustand betreten wird
- **Exit action** wird ausgefürt, wenn er verlassen wird
- **Do**: Ausführung benannter Aktivität in diesem Zustand
- Aktion **defer**: speichere das Event für spätere Bearbeitung
- **Internal transitions**: durchgeführt ohne Zustand zu verlassen
- **Self transitions**: Verlassen & Wiedereintritt