Objektorientierte Softwarekonstruktion

WS 2003/04

- Lernskript -

Rene.Reiners@post.RWTH-Aachen.de

Hinweis

Bei dem vorliegenden Dokument handelt es sich lediglich um eine *persönliche* Zusammenfassung, wobei Formulierungen teilweise aus dem Skript übernommen, oder aber auch persönlich erstellt wurden. Inhaltliche Gewichtungen oder Interpretationen stimmen nicht unbedingt mit den Inhalten und Aussagen der Vorlesungen überein.
Daher ist das vorliegende Lernskript lediglich als Hilfestellung bei der Strukturierung der Vorlesungsinhalte zu verstehen. ***Es ersetzt keinesfalls die Vorlesung oder die Bearbeitung der Skripte und erhebt in keinster Weise Anspruch auf eine der genannten Eigenschaften sowie Vollständigkeit und Korrektheit!***

Trotz allem hoffe ich, mit diesem Skript ein wenig Unterstützung beim Erarbeiten der Veranstaltungsinhalte oder einer evtl. Prüfungsvorbereitung geben zu können.
Für Verbesserungsvorschläge und Korrekturen bin ich jederzeit dankbar.

28. Januar2005

Vorlesungsbezug:

- Objektorientierte Softwarekonstruktion, Professor Lichter, Wintersemester 2003 / 04

Inhaltsverzeichnis

1. Konzepte der Objektorientierung

Struktur – Paradigmen:

- Unstrukturierte Programmierung: Lineare Folge von Befehlen, Sprünge, Marken
- Strukturierte Programmierung: Blöcke, Prozeduren, Fallunterscheidung, Schleifen → Ein Eingang und ein Ausgang für jeden Block
- Modulare Programmierung: Sammlung von Typdeklarationen und Prozeduren → Klare Schnittstellen zwischen Modulen
- Objektorientierte Programmierung: Kapselung von Daten und Operationen in Objekten (d.h. die Datenimplementierung ist verborgen), Klassen, Vererbung und Polymorphie (Verschmelzung einiger modularer und strukturierter Aspekte)

Klassifikation und Merkmale objektorientierter Sprachen:

- *Objektbasiert:* Möglichkeit, Objekte im Sinne einer **Datenkapsel** resp. eines Objektnoduls zu realisieren
- *Klassenbasiert:* Möglichkeit, Objekte in Form von Objekttypen (**Klassen**) zu beschreiben. Von diesen Klassen können **beliebig viele Objekte dieses Typs erzeugt** werden (bei objektbasierte Systemen mußte jedes Objekt einzeln beschrieben werden)
- *Objektorientiert:* Klassen können mit Hilfe von **Vererbungsbeziehungen** strukturiert werden → Objekthierarchien im Sinne der Spezialisierung resp. Generalisierung können nun modelliert werden

Objekt-Metamodell: Beschreibung der Elemente und deren Verknüpfungen die für die objektorientierte Modellierung zur Verfügung stehen. Zentrale Elemente: Objekte und Klassen. Verknüpfungen: Benutzt-Beziehung und Vererbung.
Darüber hinaus Festlegung von Bildungsregeln, die die Modellierung anleiten.
Programmiersprachen definieren teils unterschiedliche Objekt-Metamodelle (unter anderem bspw., ob Einfach- oder Mehrfachvererbung oder echter Polymorphismus möglich ist).

Ein Softwaresystem, welches aus Objekten besteht, beschreibt jede Systemaktivität als eine Aktivität eines Objektes. Objekte haben eine Lebensdauer (sie werden erzeugt und wieder vernichtet) und eine Identität (diese muß nicht durch den Programmierer hergestellt werden). Objekte verändern sich (ihren Zustand) während ihrer Lebensdauer und **senden** und **empfangen** Nachrichten.

Objekte

- entsprechen den für die Anwendung relevanten Gegenständen; Diese Gegenstände werden charakterisiert durch ihre Umgangsformen, d.h. die Art und Weise, wie mit ihnen gearbeitet werden kann. Unterscheidung: *Welche Information läßt sich an ihnen ablesen? Welche Aktionen können an ihnen ausgelöst werden?*
- Kapseln die fachlich zusammengehörigen Umgangsformen und Informationen und bieten diese nach außen in einem Satz von Operationen an (Schnittstelle der Objekte)
- Diese Datenkapsel besteht aus zwei Teilen:
    - **Daten** (Deren Wert repräsentiert den Zustand des Objektes)
    - **Operationen** (mit deren Hilfe auf die Daten des Objektes zugegriffen werden kann). Eine Operation wird ausgeführt, wenn ein Objekt eine entsprechende Nachricht erhält → Der Objektzustand kann sich verändern.
- jedem Objekt ist ein Typ zugeordnet → die möglichen Operationen sind durch den Objekttyp bestimmt. In den meisten statisch typisierten OO-PS ist der Typ durch die Klassenzugehörigkeit festgelegt.

Objekt-Schnittstelle (Menge alle öffentlichen Operationen) legt

- den **Namen** der Operation
- die **Argumentobjekte**
- die **Rückgabeobjekte** fest

→ Signatur der Operation

- legt die Dienstleistungen (Services) fest

→ Forderung: Ein Objekt muß einen Satz fachlich motivierter Dienstleitungen anbieten.
→ Anbieter- / Klient-Beziehung

Klassen

- Gemeinsame Umgangsformen oder Ähnlichkeiten von unterschiedlichen Gegenständen werden abstrahierend in einem **Begriff** und dem dahinterstehenden **Konzept** ausgedrückt.
- Die Begriffe lassen sich innerhalb so entstehender Begriffshierarchien spezialisieren oder generalisieren. → Entstehung einer Fachsprache, welche die Grundlage zur Zusammenarbeit zwischen den beteiligten Gruppen bildet und das Verständnis eines Anwendungsbereichs fördert → Fachvokabular
- Technisch werden Begriffe als **Klassen** implementiert.
- Sie definieren für ihre Objekte
    - Ihren **Namen**
    - Die **Vererbungsbeziehung** zu ihren **Oberklassen** (die erbt-von-Beziehung muß Zyklenfrei sein)
    - Die **Speicherstruktur** (Daten, Attribute, Exemplarvariablen)
    - Die **Operationen** (Methoden, Routinen) von denen ein teil exportiert wird (**public)** und ein teil nur intern benötigt wird (**private)**.
- von einer Klasse können beliebig viele Objekte erzeugt werden
- bilden die Erzeugungs- und Verhaltensmodelle für Objekte
- Objekte einer Klasse kennen dieselben Operationen, unterscheiden sich aber in den Werten ihrer Daten
- Entspricht einem ADT
- Nach außen hin sichtbar
    - Klassenname
    - Exportierte (öffentliche) Operationen

Objekt-Erzeugung

- jedes Objekt muß zur Laufzeit explizit erzeugt werden →Aufruf einer Erzeugungsoperation
- Ein Objekt ist durch seine Erzeugung ein Exemplar einer Klasse, solange es lebt
- Der Typ eines Objekts bleibt währen der gesamten Lebensdauer gleich
- Objekte werden im Speicher angelegt, evtl. initialisiert und an einen Bezeichner gebunden
    - Bezeichner erhält in statisch typisierten Sprachen durch Deklaration einen Typ (statischer Typ)
    - Zur Laufzeit können durch Zuweisung oder Parameterübergabe auch Objekte eines Subtyps an einen entsprechenden Bezeichner gebunden werden (aufgrund der Polymorphie) → dynamischer Typ eines Bezeichners
- Programmiersprachen, bei denen Klassen selbst als Objekte im System verfügbar sind
    - Erzeugen wird meist als Klassenoperation realisiert (Konstruktor)
- Compiler-Sprachen, in denen Klassen nicht als Objekte repräsentiert sind

- Code für Objekterzeugung wird vom Compiler generiert und die Instanziierung wird vom Laufzeitsystem übernommen. Die erzeugende Prozedur wird dabei meist als eine ausgezeichnete Operation innerhalb des Klassentextes definiert. Vielfach bewährt: Weitere Operation, die Objekt in einen initialen Zustand versetzt.

Vererbung

- Vererbungsbeziehung: technisch ein wesentliches Merkmal, das OO-Sprachen von konventionellen unterscheidet, Mit ihr lassen sich Klassenbeschreibungen zu hierarchischen Strukturen anordnen.
- Vererbung bedeutet technisch:
  - Alle Beschreibungen der Oberklasse gelten auch für die Unterklassen
  - Vererbungseinheit ist die Klasse
  - Statische Festlegung, von welchen Oberklassen geerbt wird
  - Vererbungsbeziehung bleibt bei statisch typisierten Sprachen während der Laufzeit unveränderlich
  - In einer Unterklasse können Beschreibungen einer Oberklasse **spezialisiert** werden
- Gemeinsame Eigenschaften einer verschiedener Klassen werden in einer eigenen Klasse zusammengefaßt und definiert. Anschließend an spezielle Klassen vererbt (Beispiel: Geo-Object → Rectangle und Geo-Object → Circle)
- **Fachliche Regel:** *Eine Klasse A erbt von einer Klasse B genau dann, wenn A eine* ***Spezialisierung*** *(ein Unterbegriff) von B ist.*
- Beliebig viele Oberklassen → Mehrfachvererbung
- Nur eine Oberklasse → Einfachvererbung

Abstrakte Klassen

- einige Operationen sind noch nicht implementiert, sondern nur deklariert
- es können noch keine Objekte erzeugt werden.
- Erst müssen abstrakte Klassen an konkrete Vererben (alle Operationen sind verfügbar, Objekte können abgeleitet werden)

Abstrakte Operationen

- sie sind nicht implementiert → Aufgabe der Unterklassen. Aber die verbindliche Schnittstelle für alle Unterklassen wird spezifiziert.
  - *Einschuboperation:* Operation gibt nur eine Standardimplementation vor, ist aber für die Redefinition in Unterklassen vorgesehen
  - *Schablonenoperationen:* Sie implementieren einen Algorithmus basierend auf abstrakten Operationen. Vollständige Angabe des Algorithmus, zur Ablauffähigkeit fehlen ihm allerdings die Implementationen der abstrakten Operationen
  - *Basisoperationen:* Operationen, die bereits in der abstrakten Klasse vollständig und ablauffähig implementiert sind

Modifikation geerbter Eigenschaften

- *Erweiterung:* Neues hinzufügen
- *Redefintion:* ähnliches Verhalten (→ nicht-strikte Vererbung)
- *Definition:* Realisation von etwas versprochenem

2. Polymorphismus und Vererbung

2.1. Grundlagen

Polymorphismus: Fähigkeit von Etwas, von verschiedener Gestalt zu sein.
In Programmiersprachen sind damit die Variablen, Funktionen und Prozeduren gemeint. Die unterschiedliche Gestalt äußert sich im jeweiligen Typ. Eine polymorphe „Entität" kann in verschiedenen Kontexten verwendet werden, die unterschiedliche Typen verlangen. Zum Beispiel Zahlen können als *Integer*, *Real* oder *Zeichen* interpretiert werden. Eine entsprechende Prozedur „*write*" muß je nach Typ anders „reagieren".

Monomorphe Sprachen: Funktionen, Prozeduren und damit auch Operanden haben einen eindeutigen Typ → Jeder Wert und jede Variable haben genau einen Typ.

Polymorphe Sprachen: Werte und Variable können mehr als einen Typ haben.

Polymorphe Funktionen: haben **aktuelle Parameter** mit potentiell mehreren Typen.

Polymorphe Typen: haben **Operationen**, die auf Operanden anwendbar sind, die mehrere Typen haben können (siehe oben Zahlenbeispiel).
Polymorphismus wird mit Hilfe der Vererbung realisiert. Es existiert folgende Klassifikation:

- **Universeller Polymorphismus**
  Echter Polymorphismus, polymorphe Operation arbeitet für eine (potentiell) unendliche Menge von Typen. Die unterschiedlichen Typen besitzen eine teilweise gemeinsame Struktur. **Derselbe Code** wird für alle Argumente der zulässigen Typen ausgewählt.
  - **Parametrisierter Polymorphismus**
    Funktion besitzt i.d.R. einen oder mehrere Typparameter und die zulässigen Typen besitzen i.d.R. eine gemeinsame Struktur. Polymorphe (generische) Funktionen haben implizite oder explizite Typparameter, die den Typ des Funktionsergebnisses für jede Anwendung der Funktion festlegen.
  - **Inklusionspolymorphismus**
    Ein Objekt kann zu mehr als einem Typ gehören, diese müssen aber nicht disjunkt sein → Subtyping ist eine Form von Inklusionspolymorphie; jedes Objekt eines Subtyps ist auch Objekt des Supertyps (Beispiel: VWs sind Autos, Darstellbare Rechtecke erben von Darstellbaren Objekten und Rechtecken (nicht disjunkt).)
- **ad hoc Polymorphismus**
  scheinbarer Polymorphismus; Operationen auf verschiedenen Typen zeigen nicht notwendigerweise gleiches Verhalten (Beispiel „+" bei Zahlen und Strings)
  → Operationen können verschiedenen Code je nach Argumenttyp ausführen. Die verschiedenen Typen müssen keine Gemeinsamkeiten haben. Diese Art von Polymorphismus kann als endliche Menge von monomorphen Operationen betrachtet werden. (Beispiel: 3.0 + 5 → mehrere Möglichkeiten, dies zu realisieren).
  - **Überladen von Operationen**
    Derselbe Bezeichner wird für verschiedene Operationen verwendet. Kontextinformationen (Parametertypen) sind notwendig, um zu entscheiden, welche Operation gemeint ist. (Kann als Form von syntaktischer Abkürzung verstanden werden, die zur Umsetzungszeit aufgelöst wird).
  - **Typanpassung**
    (Konvertierung) semantische Operation, mit der der vorhandene Typ in den erarteten Typ konvertiert wird. Ohne diese Typanpassung würden Typfehler

auftreten. Die Anpassung kann statisch oder dynamisch zur Laufzeit geschehen.

Dynamisches Binden
Die richtige Implementierung zur Laufzeit finden.
Beispiel:

g : Geo_Object; → ***statischer Typ***
r : Rectangle; → ***statischer Typ***
c : Circle; → ***statischer Typ***

r.Create;
c.Create;

g := r; → ***dynamischer Typ***
g.contains(p);

g := c; → ***dynamischer Typ***
g.contains(p);

Unterschiedliche Implementierungen werden gewählt, abhängig von der Zuweisung. *g* kann beides sein, da es ein *Geo_Object* ist und somit Supertyp von *Rectangle* und *Circle*.

Statische und dynamische Typisierung
Durch Zuweisung und Parameterübergabe können dynamische Objekte anderer Typen an einen Bezeichner gebunden werden. Aufgabe des Typecheckers ist e, bei einer polymorphen Zuweisung nur Typen zuzulassen, die konform zum statischen Typ sind (s.o.).
Dies garantiert, daß alle Operationen, die aufgerufen werden, auch vorhanden sind.
Die Aufgabe des **Laufzeitsystems** ist es, die richtige Implementierung (passend zum dynamischen Typ) zu binden.
Bei der **polymorphen Zuweisung** dürfen nur die vereinbarten Nachrichten, die für alle Klassen gelten, geschickt werden.
Polymorphismus und offene Konstruktion besteht, wenn eine Operation nur von einer Klasse abhängt und für andere erweitert werden kann (nachlesen!).

Es entsteht erhöhte **Sicherheit** durch das Typkonzept → keine Laufzeitfehler.
Außerdem besteht **Flexibilität** durch Polymorphie und dynamisches Binden (einfaches Erweitern ist möglich).
In Java, Ada 95 und Eiffel ist beides sehr hoch. Bei Smalltalk hingegen herrscht sehr hohe, aber unsichere Flexibilität.

Vorteile von Klassen als implementierte ADTs:
- **Kapselung** für die Verständlichkeit und Erweiterbarkeit
- **Typisierung** für die Sicherheit
- **Polymorphie und dynamisches binden** für die Flexibilität

Use und inherit – Beziehung
Use: Objekte und Operationen der Klasse X werden verwendet, um die Leistungen von Y zu erbringen (können Teile von X gebraucht werden?)
Komposition: Teil – Ganzes – Beziehung

Inherit: Spezialisierung; Y ist ein spezielles X (Nicht zum Zusammenkopieren von Code verwenden). Sinnvolle Anwendung, nicht Erzwingen, nur da, wo die Vererbung in oben

genanntem Kontext noch Sinn macht (Die Klassen müssen etwas mit einander zu tun haben, Spezialisierungs-Hierarchie soll vorhanden sein).

2.2. Klassifikationsschema der Vererbung nach B. Meyer:

Wie kann Vererbung eingesetzt werden?

- **Model Inheritance**
  - **Subtype Inheritance**
    Entspricht im wesentlichen „is-A-Modellierung". Oberklasse häufig abstrakt, Unterklasse abstrakt oder konkret. (Geo_Object ← Rechteck)
  - **Restriction Inheritance** (Spezialisierungssemantik)
    Existiert, wenn es eine Untermenge der Objektmenge A gibt, die genau einer definierten Bedingung genügen (Bsp.: Geo_Object ← Rechteck ← Quadrat)
  - **Extension Inheritance**
    Unterklasse B führt neue Eigenschaften ein, die in Oberklasse A nicht vorhanden oder nicht auf sie anwendbar sind.
  - **View Inheritance**
    Zu einer Klasse A gibt es Unterklassengruppen B1 bis Bn, die diese im Sinne von unterschiedlichen Klassifikationsaspekten spezialisieren. Beispiel: Klassifikation eines Angestellten nach Art der Anstellung und Art der Aufgabe.
- **Variation Inheritance** (Redefinition von geerbten Eigenschaften und keine Neueinführung von Eigenschaften außer denen, die für die Redef. Notwendig sind)
  - **Functional Variation Inheritance**
    Rumpf der redefinierenden Operation wird verändert (Ablauf-Code)
  - **Type Variation Inheritance**
    Redefinition beschränkt sich nur auf die Signaturen der Operationen (Ausführungsablauf ändert sich nicht, nur Veränderung der Parameter und Rückgabetypen)
  - **Uneffecting Inheritance**
    Redefinition von Eigenschaften einer Oberklasse, so daß sie virtuell werden (ungewöhnliche Art der Vererbung??)
- **Software Inheritance**
  - **Reification Inheritance**
    „Vergegenständlichung": Klasse A definiert eine allgemeine Datenstruktur und die Unterklasse B liefert eine Implementierungsalternative dafür. **A ist immer abstrakt, B kann abstrakt oder konkret sein.**
  - **Structure Inheritance**
    Auch: *Aspect Inheritance:* Eine abstrakte Klasse A definiert eine allgemeine strukturelle Eigenschaft und eine Unterklasse B ist so realisierbar, daß ihre Exemplare diese strukturelle Eigenschaft besitzen. Beispiel: Eigenschaft *comparable* -> Realisierung in *Integer* oder *Point*, ebenso mit *printable* und dann *Letter, Drawing*.
  - **Implementation Inheritance**
    Eine Unterklasse B erbt eine Menge von Eigenschaften von A. Diese sind ausschließlich notwendig, um die zu B gehörende Abstraktion zu implementieren (Mit Vorsicht zu genießen). → Beide Klassen müssen konkret sein. → Zusammensetzen, was gebraucht wird.

- **Facility Inheritance**
  Liegt vor, wenn eine Klasse nur zu dem Zweck konstruiert wurde, um zusammengehörende Eigenschaften zu definieren, die anderen Klassen mittels Vererbung zur Verfügung gestellt werden sollen.
  - **Constant Inheritance**
    Die Klasse definiert global nutzbare Konstanten
  - **Machine Inheritance**
    Die Klasse definiert Eigenschaften einer (abstrakten) Maschine
    Eigentlich gegen den Grundgedanken der OO. Man sollte solche Probleme nicht mit Vererbung lösen.

2.3. Das Substitutionsprinzip nach Liskov

Ein ungezügelter Einsatz der Vererbung im Sinn der Wiederverwendung von Code führt zu degenerierten und nicht mehr wartbaren Programmen → Verstärkung bei Mehrfachvererbung. Solche Programme sind nicht mehr testbar.

Das Substitutionsprinzip
Ein Objekt der Unterklasse B muß zu jeder Zeit ein Objekt der Oberklasse A ersetzen können.
(Gegenbeispiel: Verfeinerung der Klasse Rechteck zu einem Quadrat. Gibt es in der Klasse Rechteck eine Operation, mit der nur eine Seite gestreckt werden kann, die andere aber nicht verändert wird, kann die Unterklasse Quadrat diese Operation so nicht durchführen und somit Rechteck nicht ersetzen.)

Regeln, die das Einhalten des Substitutionsprinzips garantieren

- *Signatur – Regel:* Unterklasse muß alle Methoden der Oberklasse kennen, d.h. die Signaturen der Methoden der Unterklasse müssen kompatibel mit den Signaturen der der Methoden der Oberklasse sein.
  (Sicherstellung durch den Übersetzer einer typisierten OO-Sprache)
- *Methoden – Regel: Nachbedingungen* einer redefinierten öffentlichen Methode können in der Unterklasse unverändert bleiben oder verschärft werden (keine Lockerung!)
  Vorbedingungen können erhalten bleiben oder abgeschwächt werden (keine Verschärfung!)
- *Eigenschaften – Regel:* Unterklasse muß alle Eigenschaften der Oberklasse bewahren (Gegenbeispiel: siehe Rechteck – Quadrat). Dies gilt insbesondere für Klasseninvariante. (Entwicklerweitsicht gefragt).

2.4. Vererbung im Einsatz

Fachliche Modellierung: Bei der fachlichen Modellierung von Konzepten und Begriffen wird durchgängig die Einfachvererbung eingesetzt → Klarheit der Begriffsbildung

Technische Konstruktion: Mehrfachvererbung kann sich als nützlich erweisen, wenn sie diszipliniert eingesetzt wird.

Softwaretechnisch: Trennung von Spezifikation und Implementation in Ober- und Unterklasse. Spezifikation (Verhaltensvorgabe) in der Oberklasse, konkrete Realisierung des Verhaltens auf konkreter Ebene.
Technisch: Abstrakte Klassen: Spezifikationen → große Bedeutung beim Bau von Frameworks → In gewissem Umfang wird der Umgang mit Ihren Unterklassen festgelegt und damit in Grenzen auch der Entwurf des restlichen Systems definiert.

3. OO Analyse und Use Cases

3.1. OO – Analysemethoden

Eine Analysemethode ist eine Software-Entwicklungsmethode, die in der Analysephase eingesetzt wird, um die Anforderungen an ein Software-System vollständig zu analysieren und um widerspruchsfrei und präzise zu definieren, was die Software leisten soll.
Eine ***OO-Analysemethode*** ist eine Analysemethode, die die Modellierungskonzepte der ***Objektorientierung*** zur Verfügung stellt. Das heißt, es wird von **Attributkapselungen (statisch und dynamisch) in Objekten**, **Klassenbildungen** und **Spezialisierungsbeziehungen** gebrauch gemacht.

Die Konzepte der Analyse der Problemwelt, des Softwaredesigns und der Implementierung sollten durchgängig objektorientiert sein, wenn man sich für den Ansatz in einem der drei Gebiete entscheidet.

Typisches OO-Analyse-Vorgehen

1.) Objekte und Klassen identifizieren
2.) Verantwortlichkeiten identifizieren und den Klassen zuordnen
3.) Zusammenarbeit zwischen den Klassen identifizieren
4.) Hierarchien definieren
5.) Subsysteme definieren
6.) Schnittstellen definieren

Problem: ***Woher bekommt man die Ansätze für Objekte und Klassenhierarchien?***

- Eine Lösung bilden ***Use Cases***, die bei der Anforderungsanalyse helfen, Strukturen herauszufinden und weiterhin eine natürlichsprachliche Kommunikation mit dem Kunden ermöglichen.

3.2. Use Cases

Analysetechnik, vorgestellt 1992 von Jacobson.

Definition: Eine Sequenz von Interaktionen zwischen einem oder mehreren Akteuren (Benutzern, externen Systemen) und einem System. Dieses System reagiert auf den oder die Akteure und produziert ein entsprechendes Ergebnis.

Ziele:

- Aufnahme des Ist-Zustandes (Arbeitsablauf ohne das zukünftige System)
- Beschreibung des Soll-Zustandes (Arbeitsablauf mit dem zukünftigen System)
- Identifizierung der relevanten Gegenstände und Funktionalitäten des Anwendungssystems
- Basis für den Entwurf
- Kommuniaktionsmedium zwischen Entwickler und Kunden (sehr wichtig)

Mit Use Cases werden funktionale Anforderungen erfaßt. Use Case – Diagramme finden Eingang in Use Case Dokumente. Aus diesen werden dann Prototypen erstellt, die Dokumente selbst können in ein Glossar und die einzelnen Use Cases (textuelle Beschreibungen der Diagramme bzw. eines Handlungsszenarios).

Use Cases werden i.d.R. von den Entwicklern erstellt, mit Fachleuten der Anwendung diskutiert und bilden eine sehr gute Basis, Prototypen zu erstellen. Sei beschreiben die extern

sichtbare Funktionalität vollständig und bilden die Grundlage für das Begriffslexikon (Projektsprache) – somit lernen auch die Entwickler, sich in die Fachsprache der Benutzer einzuarbeiten. Letztere sehen ihre Arbeitshandlungen aus einer anderen Sicht (Betriebsblindheit).

Vorgehensweise der Use Case - basierten Analyse:

- Identifiziere wichtige Anwendungsfälle (dies sind die Use Cases)
- Identifiziere Beziehungen zwischen den Akteuren und den Use Cases
- Identifiziere Beziehungen zwischen den Use Cases
- Beschreibe Use Cases in strukturierter Form in Umgangssprache
- Use Cases werden auf der Basis der Benutzer-Interaktionen, Leistungen, die das System erfüllen soll und Zielen, die erreicht werden sollen, gefunden
- Ein Use Case beschreibt eine Interaktion, die endet, sobald der Akteur wechselt oder eine Zeitverzögerung auftritt (das System wartet). Nachteil der Benutzer-Interaktionsbeschreibung ist, daß diese sehr stark abhängig von der GUI ist, die sich häufig ändern kann.

Die Identifizierung von Use Cases ist zielorientiert:

- Beschreibung, wie ein Benutzer ein Ziel erreichen kann
- Ein Use Case wird durch ein bestimmtes Ereignis angestoßen
- Ein Use Case endet, wenn das angestrebte Ziel erreicht oder nicht erreicht wurde.
- Wichtig ist es, die top-level-Use Cases zu finden, um die Struktur des Systems zu ermitteln (man soll sich nicht in Einzelheiten verlieren, denn Use Cases können auf verschiedenen Ebenen gefunden werden → Basisfunktionen, höhere Funktionen, top-level-Funktionen) → Hierarchische Struktur der Use Cases.

Use Case – Notation basiert auf der UML:

- *Use Case Diagramme (statische Modellierung):* Beschreibung der Funktionalität des Systems aus Sicht des Benutzers in Form von Anwendungsfällen.
  Beschreiben, welche Akteure an welchen Use Cases beteiligt sind, Beziehungen zwischen den einzelnen Use Cases und geben einen Überblick über alle Use Cases.
  Wichtig: Use Case – Diagramme zeigen nicht den inneren Ablauf eins Use Cases; Die detaillierte Beschreibung erfolgt mittels strukturierten, textuellen Darstellungen und dynamischen Modellierungstechniken (Sequenzdiagrammen)
- *Klassendiagramm (statische Modellierung):* Beschreibung der Statik des Systems in Form von Klassen, Objekten, Schnittstellen, Paketen und deren Beziehungen. Spezialform: Objektdiagramm
- *Komponentendiagramm (statische Modellierung):* Zeigt die Implementierungsabhängigkeiten von Implementierungskomponenten.
- *Verteilungsdiagramm (Statische Modellierung):* physische Verteilungsstruktur des Systems im Sinne Client/Server, Prozessoren, Netzwerken, etc.
- *Sequenzdiagramm (dynamische Modellierung):* Beschreibung der notwendigen Interaktionen zwischen Objekten zur Erfüllung einer bestimmten Aufgabe (def. Durch einen Use Case). Im Vordergrund steht die zeitliche Ordnung der Nachrichten zwischen den Objekten.
- *Kollaborationsdiagramm (dynamische Modellierung):* s.o. Vordergrund: Strukturelle Organisation der Objekte.
- *Zustandsdiagramm (dynamische Modellierung):* Beschreibung des erlaubten Verhaltens von Objekten einer Klasse in Form von Zuständen und

Zustandsübergängen. Es werden die zeitlichen und kausalen Abhängigkeiten der Operationen einer Klasse modelliert.
- *Aktivitätsdiagramm (dynamische Modellierung):* Beschreibung des Zusammenspiels einzelner Aktivitäten (Kontroll- und Objektfluß), um eine Aufgabe zu erfüllen

Beziehungen
- *Zwischen Akteuren:* Vererbung: Ist ein Akteur B eine Spezialisierung eines Akteurs A, so nimmt auch er an allen Use Cases in der Rolle von A teil.
- *Zwischen Use Cases:*
  - *Extend:* B extends A bedeutet, daß das Verhalten von B in A eingefügt werden **kann**.
  - *Include:* A includes B bedeutet, daß das in B beschriebene Verhalten in A eingefügt **wird**. Zweck: Wiederverwendung von Funktionalität und schrittweises Spezifizieren neuer Funktionalität.
  - *Generalisierung:* A ist Generalisierung von B bedeutet, daß B das Verhalten von A erbt und es überschreiben und ergänzen kann.

3.3. Use Case basiertes Testen

Die SCENT – Methode:
SCENario.based validation and Test of Software
Scenraio = Use Case

Die ermittelten Use Cases werden auch beim Testen verwendet: Die Use Cases bzw. Zustandsdiagramme werden mit Anmerkungen zu Testfallherleitung versehen:
- Vor- und Nachbedingungen
- Datenbereichen, Datenwerten, erwarteten Resultaten
- Leistungs- und nicht-funktionalen Anforderungen

Testfälle werden nun bestimmt, indem Pfade in den Zustandsdiagrammen durchschritten werden.
Die SCENT – Methode ist ein iterativer Prozeß:
- Akteure identifizieren
- Systemgrenzen festlegen
- Toplevel Use Cases auf Geschäftsprozeßebene erstellen
- Apriorisieren und verfeinern (Übersichtsdiagramm, Modellierung von Alternativen und Behandlung von Ausnahmen)
- Identifizieren wiederverwendbarer Use Cases (miteinbeziehen von nicht-funktionalen und Leistungsanforderungen)
- Benutzer reviewen und validieren Use Cases.

Nutzen: Probleme natürlichsprachlicher Spezifikation (Ungenauigkeit, Mehrdeutigkeit, Widersrpüchlichkeit) werden durch die Use Case – Formalisierung abgeschwächt. Dokumente der Anforderungsermittlung werden systematisch für den Test genutzt.

Ein Testauswahlkriterium: ***Ablaufüberdeckung***:
In der Testfallmenge für einen Use Case sind alle die Testfälle enthalten, die benötigt werden, um alle möglichen Schrittfolgen durch den Use Case (Normalfall, Alternativen, Sonderfälle) einmal zu testen.

Testfallherleitung:

- Basis: Zustandsdiagramme
- Bestimmung der Testfälle, indem alle Pfade in den Zustandsdiagrammen durchschritten werden.
- Knoten-Kanten-Überdeckung (oder andere).
- Normalablauf zuerst
- Anschließend Alternativen und Ausnahmen

→ Datenbereiche und Datenwerte: Äquivalenzklassenbildung, Grenzwertbetrachtung
→ Zeit- und Leistungsanforderungen

Probleme:

- Transformation selbst ist nicht formalisiert, nur Richtlinien
- Keine automatische Generierung von Zustandsdiagrammen aus Use Cases
- Ein weiterer Modellierungsschritt → mögliche Fehlerquelle

4. Software-Entwurf

Der Entwurf legt die Architektur des Systems aus statischer und dynamischer Sicht fest. Er soll eine stabile Grundlage für die Implementierung und spätere Tests darstellen, auf die das System aufgebaut wird. Gleichzeitig werden viele Qualitätseigenschaften durch den Entwurf beeinflußt. Um die Architektur zu beschreiben, werden ADLs benutzt.
Der objektorientierte Entwurf besteht aus der Modellierung von Klassen, Paketen, Bibliotheken und Frameworks. Weiterhin kann die UML als eine ADL verwendet werden, da sie Klassendiagramme, Sequenz- und Kollaborationsdiagramme zur Verfügung stellt.

Grundlegende Entwurfsprinzipien:

- ***Abstraktion***
  Hilfe beim Umgang mit Komplexität → Nicht benötigte (Detail-) Informationen werden ausgeblendet. Beispiele: Kapselung, Generalisierung, virtuelle Maschinen
- ***Teile und herrsche***
  Zerlegung eines Systems oder einer Komponente in kleinere, relativ unabhängige Teile → Separater Entwurf, Abstraktion der Komponenten auf ihre Schnittstelle, Erzeugung von Hierarchien oder Baumstrukturen von Komponenten (Struktur, Übersicht)
- ***Modularisierung***
  Aufteilung des Systems in sinnvolle Subsysteme und Module (Behälter für Funktionen oder Zuständigkeiten des Systems)
- ***Kapselung***
  Zusammengehörige Bestandteile einer Abstraktion werden zu einem Ganzen zusammengefaßt und von anderen abgegrenzt
- ***Information Hiding***
  Alle Entwurfsentscheidungen werden in Modulen verborgen, Zugriff auf Interna nur über wohldefinierte Schnittstellen. Verwender erfährt nur, was er wissen muß, um mit dem Modul umzugehen.
- ***Kopplung und Zusammenhalt***
  Zusammenhang zwischen unterschiedlichen Entwurfs- oder Konstruktionseinheiten. Eine *minimale Kopplung* zielt auf die Reduktion der Bindung zwischen den Einheiten, insbesondere die Vermeidung zyklischer Benutzung.

Maximaler Zusammenhalt fordert eine möglichst hohe Bindungsstärke innerhalb einer Entwurfs- oder Konstruktionseinheit (verschiedene Stufen der jeweiligen Konzepte)

- ***Ausreichend, vollständig, einfach***
  Ausreichend: Minimale Schnittstelle → Verwendung sinnvoll → Umfassen aller Eigenschaften einer Abstraktion, die für eine sinnvolle und effiziente Verwendung der Komponente notwendig sind.
  Vollständig: Komponente umfaßt alle **relevanten** Eigenschaften einer Abstraktion → generelle Schnittstelle erlaubt Wiederverwendung
  Einfach: Alle Operationen einer Komponente könne leicht implementiert werden und komplexe Operationen werden auf eine Kombination einfacherer Operationen zurückgeführt.
- ***Trennung der Zuständigkeiten***
  Aufteilung des Systems sollte anhand von Zuständigkeiten vorgenommen werden:
  - Gruppierung von Komponenten, die an der gleichen Aufgabe beteiligt sind
  - Abgrenzung von anderen Aufgabenbereichen
  - Gruppen bilden ein Subsystem
  - Bei Komponenten, die für mehrere Aufgaben zuständig sind, werden die Bestandteile intern nach Zuständigkeiten gruppiert
- ***Trennung zwischen Schnittstelle und Implementierung***
  Die Schnittstelle definiert die **Funktionalität** der Komponente und ist den **Verwendern** zugänglich.
  Die Implementierung enthält den **tatsächlichen Code** für die Implementierung der Komponente und weitere, nur intern benötigte Funktionalität
  Vorteile: Die Trennung schützt die Verwender vor Implementierungsdetails. Die Implementierung kann geändert oder ausgetauscht werden, ohne daß der Verwender davon betroffen ist.
- ***Open Closed Principle***
  Das Modul kann gefahrlos verwendet werden, da sich seine Schnittstelle nicht mehr ändert. Gleichzeitig kann es aber auch problemlos erweitert werden (Möglich in der OO durch Vererbung → Die Schnittstelle wird in eine abstrakte Klasse umgewandelt, Subklassen können Funktionalitäten erweitern)

Entwurfskriterien

- *Zerlegbarkeit* in kleinere, weniger komplexe Teilprobleme → einfache Struktur und weitgehend unabhängige Konstruktion
- *Kombinierbarkeit:* Die Entwurfseinheiten sollten sich durch einfache Rekombination zu neuen Softwaresystemen in verschiedenen Anwendungsbereichen zusammenfügen lassen**.**
- *Verständlichkeit* einer Entwurfseinheit weitgehend unabhängig von anderen verständlich.
- *Kontinuität:* Änderung des Entwurfsproblems (aus Anwendungsbereich oder Entwurfproblem) sollte Änderungen nur in einer oder wenigen Entwurfseinheiten erfordern.

Regeln, um diese Kriterien zu erfüllen

- Direkte Abbildung
- Wenige Schnittstellen
- Kleine Schnittstellen (geringer Informationsaustausch)
- Explizite Schnittstellen
- Geheimnisprinzip (nur relevante Merkmale sind für den Klienten sichtbar und zugänglich)

Entwurfseinheiten

- *Schichten:* top-level-Entwurfseinheiten; Zusammenfassung logisch zusammengehörender Entwurfseinheiten. Aufbau auf anderen Schichten, Dienstleistungen in Form von Schnittstellen werden angeboten. Elemente einer Schicht sind Rahmenwerke oder Klassenbibliotheken.
- *Klassenbibliotheken:* Zusammenfassung einer Menge von einzeln verwendbaren Klassen zu einer Einheit.
- *Rahmenwerke:* Architektur aus Klassenhierarchien, die eine allgemeine, **generische** Lösung für ähnliche Probleme in einem bestimmten Kontext vorgibt. Konkrete Anwendung: Spezialisierung der vorgegeben Klassen oder Erzeugung vorgegebener Parameterobjekte.
- *Pakte (Cluster):* Gruppierung von Klassen zu höherwertigen Einheiten. Rahmenwerke und Klassenbibliotheken bestehen aus Clustern oder Pakten.
- *Klassen:* Atomare Entwurfs- und Konstruktionseinheiten

Entwurfsnotation:
Erster Abschnitt: Klassenname, Angabe eines Stereotyps, spezielle Merkmale.
Zweiter Abschnitt: Exemplar- und Klassenvariablen.
Dritter Abschnitt: Exemplar- und Klassenmethoden.
(abgleitet /, private -, public +, Klassenmethode _, protected #)

5. Entwurfsmuster

5.1. Grundlagen

Allgemein bedeutet *Muster* eine Vorlage, nach der etwas hergestellt wird. Ähnliche Probleme wurden bereits untersucht und eine vorgegebene Lösung, nach der man sich richten kann, existiert bereits. Eine *Mustersprache* beschreibt ein System von Mustern, welche sich mit demselben Gebiet (in diesem Kontext der Softwarekonstruktion) befassen, sich gegenseitig ergänzen und deren Beziehungen und Kombinationsmöglichkeiten explizit aufgezeigt werden. Durch Kombinationen der Muster können größere Probleme gelöst werden, als das mit einzelnen Mustern möglich ist → auch verwendbar zur Herleitung eines Entwurfs.

Nach Gamma (SE): Ein Entwurfsmuster beschreibt ein bestimmtes, in einem gegeben Kontext immer wiederkehrendes Entwurfsproblem, sowie ein vorgegebenes Schema zu seiner Lösung.

Im Software Engineering entstehen Muster aus typischen Anwendungsproblemen heraus, für die eine allgemeingültige Lösung gefunden wurde. Sammlungen von Entwurfsmustern sind z. B. der GOF- oder der Siemens – Katalog.
Entwurfsmuster werden zur Darstellung von grundlegenden Konstruktionsprinzipien, Dokumentation von Klassenbibliotheken und Rahmenwerken und als Musterlösungen für typische Konstruktionsaufgaben. Weiterhin können Muster auch beim Reengineering als Analysemittel dienen.

Nutzen

- Mikroarchitektur: Entwürfe werden wiederverwendbar
- Unterstützung der Kommunikation im Entwicklerteam (Design-Vokabular)
- Vereinfachung der Einarbeitung
- Erhöhung der Flexibilität von Entwürfen
- Wörter einer Sprache über Softwarearchitekturen

- Einsatz bei der Konstruktion von Frameworks

Entwurfsprinzipien
- Eine Konstruktion soll offen und flexibel gestaltet sein
- Ziel: Wiederverwendbarkeit, Verständlichkeit und Änderbarkeit der Konstruktion
- Konstruktionsprinzpien:
    - Kapselung (information hiding)
    - Lose Kopplung von Komponenten
    - Gut begründete starke Kopplung von Komponenten (bewußt, gewollt)
    - Gleichverteilung der Verantwortlichkeiten
    - Wahrnehmen der Verantwortung und Delegation

Musterbeschreibung (kein einheitliches Format)
- Name
- Problem
- Kontext in dem das Problem auftritt
- Einflußfaktoren die berücksichtigt werden müssen
- Lösung Struktur und Dynamik
- Einschränkungen
- Implementierung
- Beispiele, Varianten, bekannte Verwendungen, verwandte Muster,…

Klassifikation nach
- **Ebene**
    - Class: etabliert durch Vererbung, statisch
    - Object: etabliert durch Assoziationen, dynamisch
- **Zweck**
    - Creational: Einsatz, beim Erzeugen von Objekten
    - Structural: Zusammenbau komplexer Objete
    - Behavioral: Einsatz, wenn Objekte komplexe Aufgaben zusammen ausführen müssen

5.2. Einige GOF-Entwurfsmuster im Detail

5.2.1. Creational

Factory – Method (Class)

Problem: In Frameworks wird vor allem mit abstrakten Klassen gearbeitet, dennoch muß es diesen möglich sein, konkrete Objekte zu erzeugen. Allerdings ist deren Typ bei der Entwicklung des Frameworks noch nicht bekannt, sondern erst bei der Verwendung.

Lösung: Eine abstrakte Klasse definiert eine abstrakte Methode (*Factory Method*), die ein Objekt von einer allgemeinen Oberklasse liefert. Diese ist dann von konkreten Unterklassen so zu redefinieren, daß ein Objekt vom gewünschten Typ erzeugt wird. → Für die konkrete Erzeugung sind also die Unterklassen verantwortlich.

Bemerkungen: Falls es sinnvoll ist, kann in *Creator* eine Default-Implementierung der Factory - Method angegeben werden. Factory Methods können auch parametrisiert werden,

um verschiedene Produkte zu liefern. → spart spezialisierte Factory – Methods, Unterklassen können neue Varianten zulassen oder vorhandene umwidmen.
Alternative zur Factory-Method: Die zu instantiierende Klasse als Parameter einer generischen Klasse Creator angeben. Anschließend können konkrete Creator-Klassen erzeugt werden.

Bewertung:
- *Vorteile:* Das zu entwickelnde Framework kann unabhängig von anwendungsspezifischen Klassen geschrieben werden. Der Frameworkcode hängt nur von der Schnittstelle von *Product* ab. Es wird eine höhere Flexibilität als bei der expliziten Instantiierung einer bestimmten Klasse erreicht.
- *Nachteile:* Um ein konkretes Objekt erzeugen zu können, muß auf jeden Fall eine Unterklasse von *Creator* gebildet werden.

Prototype (Object)
Problem: Ein System, d. h. eine Klassenbibliothek oder ein Rahmenwerk soll unabhängig davon entwickelt werden, wie die Objekte, die es manipuliert, erzeugt, zusammengesetzt und repräsentiert werden. Die Klassen der zu erzeugenden Objekte werden erst zur Laufzeit spezifiziert. Exemplare einer Klasse haben nur wenige unterschiedliche Zustandskombinationen.

Lösung: Bestimme die Arten der zu erzeugenden Objekte durch die Verwendung eines prototypischen Exemplars und erzeuge neue Objekte durch Kopieren dieses Prototyps. Dann kann eine entsprechende Zahl an Prototypen eingerichtet werden, die gleich mit dem richtigen Zustand erzeugt werden.

Bewertung: Prototypen ermöglichen es, neue Klassen in ein System einfach dadurch einzubinden, daß ein prototypisches Exemplar beim Klienten registriert wird. In hochdynamischen Systemen lassen sich effektiv neue Arten von Objekten definieren, indem Objekte existierender Klassen erzeugt, kombiniert und dann als Prototypen bei Klienten registriert werden → Schaffung neuer „Klassen“ ohne Programmierung.
Die Anwendung des Prototyp-Musters kann eine schwierige Aufgabe sein, wenn die zu klonenden Klassen bereits existieren und auf ihren Quellcode nicht zugegriffen werden kann, weil jede Unterklasse von Prototyp die Operation „xxx“ implementieren muß.

Singelton (Object)
Problem: Es soll sichergestellt werden, daß es zu einer Klasse höchstens ein Exemplar gibt. Dieses Exemplar soll global verfügbar sein → es muß eine Möglichkeit für alle Objekte geben, dieses Exemplar anzusprechen.

Beispiele: Drucker-Spooler, Dateisystem, Window-Manager

Lösungsalternativen:
- Instanz in einer globalen Variablen anlegen. Problem: Keine Verhinderung der Mehrfachinstanziierung.
- Deklaration aller Eigenschaften der Instanz als Klasseneigenschaften. Problem: Keine Redefinition möglich.

Lösung: Die Klasse ist selbst dafür verantwortlich, daß es von ihr nur ein Exemplar gibt. Sie stellt auch dieses Exemplar zu Verfügung.

Struktur: Die Instanz wird in einer privaten Klassenvariablen *theInstance* gespeichert. Mit der Klassenmethode *getInstance()* kann sie angefordert werden. Aufruf: *Singleton.getInstance()*

Implemetierung siehe Folien!

Bemerkungen: Eine Redefinition der Singleton-Klasse ist möglich, allerdings muß dazu auch die Methode *getInstance()* überschrieben werden. Eine Alternative besteht darin, im Singleton eine Factory-Method zur Instantiierung zu verwenden. Eine andere Alternative ist die, in der Methode *getInstance()* anhand globaler Informationen (Properties, Umgebungsvariablen) oder Parametern zu entscheiden, welche Klasse zu instantiieren ist.
Wenn ein Programm viele Singletons hat, kann eine Registry eingerichtet werden: Die Singletons können über einen Schlüssel (z.B. Klassenname) von ihr angefordert werden. → Dazu registriert sich jede Singleton-Instanz (im Konstruktor) bei der Registry.
Problem: Der Konstruktor muß von irgendwem aufgerufen werden.

Bewertung: Singleton ermöglicht den überwachten Zugang zu einer einzigen Instanz, welche erst bei Bedarf erzeugt werden muß. Es ist keine globale Variable notwendig. Eine Redefinition der Singleton-Klasse ist möglich, weiterhin kann sie für n Instanzen verallgemeinert werden. Diese Methode ist flexibler als eine Realisierung durch Klassenoperationen.
Nachteile: Alte Gewohnheit der Deklaration vieler globaler Variablen unter dem Deckmantel der Entwurfsmuster kann wieder aufleben.

5.2.2. Structural

Adapter (Class / Object)
Problem: Eine Klasse A soll eine andere Klasse B verwenden, aber deren Schnittstelle paßt nicht. Passiert durch Kombination verschiedener Klassenbibliotheken oder Wiederverwendung verschiedener Klassen.

Lösung: Passe die Schnittstelle einer Klasse an eine von ihren Klienten erwarteten Schnittstelle an → Das Adaptermuster läßt Klassen zusammenarbeiten, die wegen ihrer inkompatiblen Schnittstellen ansonsten nicht dazu in der Lage wären.
Anwendbarkeit:

- Eine existierende Klasse soll verwendet werden, aber die Schnittstelle paßt nicht.
- Eine wiederverwendbare Klasse soll erstellt werden, aber die Klassen, mit denen sie zusammenarbeiten, sind in ihrer Schnittstelle nicht vorhersagbar.
- Verschiedene Unterklassen sollen wiederverwendet werden, aber nicht jede einzelne Schnittstelle durch Spezialisierung angepaßt werden.

Bewertung:

- *Klassenadapter:* Paßt die zu adaptierende Klasse (Adaptee) an genau eine Zielklasse (Target) an. Unterklassen lassen sich so nicht mit anpassen. Er kann das Verhalten der adaptierten Klasse anpassen, da der Adapter Unterklasse ist. Der Adapter verwendet nur ein Objekt und keine „Durchreiche-Operationen“ auf das adaptierte Objekt.
- *Objektadapter:* Ein solcher Adapter kann die adaptierte Klasse und alle Subklassen anpassen. Es sind nur begrenzte Verhaltensänderungen ohne Unterklassenbildung der adaptierten Klasse möglich.

Composite (Object)
Problem: Es sollen hierarchische Teil-Ganzes-Beziehungen modelliert werden, d. h. einzelne Objekte (Primitive) und zusammengesetzte Objekte (Container) sollen für den Verwender die gleiche Schnittstelle haben.
Beispiele: gruppierbare graphische Objekte, Parse-Baum, GUI-Elemente

Lösung: Konstruiere eine gemeinsame Oberklasse für Primitive und Container, die beide Eigenschaften in sich vereinigt.

Bemerkungen: Die Composite – Klasse reicht normale Methodenaufrufe in der Regel an alle Kinder durch, kann aber auch noch zusätzliche Operationen durchführen.
Composite-Operationen in Component: Aufnahme dieser Operationen ist praktisch (einheitliche Schnittstelle) aber auch problematisch, da ihr Aufruf auf einen *Leaf* nicht sinnvoll ist. Zumindest ist eine Default-Implementierung dieser Operationen /z.B. Execption auslösen), ansonsten müßte jede Leaf-Klasse das selbst tun. Ein Verschieben in Composite kann sinnvoll sein → höhere Sicherheit.

Bewertung:
Vorteile:
- Gleiche Schnittstellen für einfache und zusammengesetzte Objekte
- Ermöglichung einfacherer Verwendung
- Möglichkeit der Erweiterung der Struktur um neue Klassen.
- Einzelobjekte und zusammengesetzte Objekte können rekursiv kombiniert werden#

Nachteile:
- Composite-Operationen in der Compinent-Klasse problematisch
- Restriktionen bei der Zusammensetzung lassen sich schwierig realisieren (z.B. Einschränkung auf bestimmte Typen läßt sich nur zur Laufzeit prüfen)

5.2.3. Behavioral

Template Method (Class)
Problem: Ein Algorithmus, zu dem es mehrere Varianten gibt, soll realisiert werden. Weiterhin soll es möglich sein, einfach neue Varianten hinzuzufügen und außerdem soll möglichst wenig Code dupliziert werden.

Idee: Den Code, den alle Varianten gemeinsam haben in ein Algorithmusgerüst packen und die varianten Teile in neue Methoden auslagern, die vom Gerüst aufgerufen werden.
→ Dazu müssen allerdings alle Varianten einem gemeinsamen Schema entsprechen.

Bemerkungen: Template ist eines der wichtigsten Muster zur Realisierung von Frameworks. Die Superklasse ruft Operationen der Subklasse auf („Don't call us, we call you"). Primitive Operationen müssen nicht abstrakt sein, sie können auch eine sinnvolle Default-Implementierung beinhalten (z.B. NOP). Es sollte so wenig wie möglich primitive Operationen geben, die redefiniert werden müssen (*Narrow Inheritance Interface Principle*). Der Algorithmus sollte möglichst allgemein parametrisiert werden können.
→ Eine sinnvolle Konvention ist es, primitive Methoden mit dem Präfix do zu benennen.

Iterator
Problem: Es soll über die Elemente einer komplexen Datenstruktur, z.B. einer Liste oder Baum, iteriert werden. Weiterhin soll es möglich sein, mehrere Iterationen gleichzeitig durchführen zu können. Die interne Struktur soll trotzdem verborgen bleiben.

Lösungsalternativen:
- Erweitere die Schnittstelle der Datenstruktur um Methoden zur Iteration (Cursor-Mechanismus)
- Trenne die Datenstruktur von ihrem Iterator

Bewertung:
Vorteile: Trennung der Zuständigkeiten zwischen Datenstruktur und Iterator, d.h. die Schnittstelle und Implementierung der Datenstruktur werden vereinfacht. Mehrere Iterationen auf derselben Datenstruktur sind gleichzeitig möglich. Polymorphe Iterationen erlauben eine Abstraktion von der konkreten Implementierung der Datenstruktur.
Nachteile: Iteratoren benötigen Zugriff auf den internen Zustand der Datenstruktur → Durchbrechen der Kapselung, enge Kopplung.
Es sind mehr Objekte (und Klassen) im System vorhanden.

Observer
Problem: Die Konsistenz zwischen Objekten soll sichergestellt werden, z.B. zwischen Daten (Modell) und ihren Darstellungen (Views)

Kontext: Der Zustand eines oder mehrerer Objekte (Beobachter) ist vom Zustand eines anderen Objekts (Subjekt) abhängig. Es ist wichtig, jederzeit die Konsistenz der Objekte zu gewährleisten: Alle Änderungen des Subjekts sollen dem Beobachter bekannt werden. Ein Subjekt kann mehrere Beobachter haben.

Einflußfaktoren: Die Objekte sollen trotz der Forderung nach Erhaltung der Konsistenz weitgehend entkoppelt bleiben, d.h. sie sollen möglichst wenig voneinander „wissen“. Die Beobachter eines Subjekts sind a priori nicht bekannt; ihre Anzahl kann sich dynamisch ändern.

Die Beobachter kennen das zu beobachtende Objekt, nicht aber umgekehrt. Das beobachtete Objekt teilt die eigenen Änderungen den Beobachtern mit.
Dazu wird eine Vermittlung benötigt, bei der sich die Beobachter anmelden. Das Beobachtete Objekt informiert somit nur die Vermittlung (changed), diese informiert dann die angemeldeten Beobachter (update your view). (Smalltalk: Changed-Update-Mechanismus).

Strategy

Motivation: In einem Eingabeformular muß in einer Reihe von Textfeldern die Eingabe auf korrekte Syntax geprüft werden (z.B. ganze Zahl, PLZ, E-Mail-Adresse,…).

Lösungsmöglichkeiten:
- *monolithisch:* Beim Bestätigen der Eingaben wird in einer „*submit*“-Methode geprüft, ob die Textfelder korrekt ausgefüllt sind. Problem: Schlechte Änderbarkeit bei Änderungen des Formulars oder der Syntax, Codeduplikation.

- *Faktorisierung:* Die Prüfroutinen für einzelne Eingabeformate werden in spezielle Methoden ausgelagert, die von der „*submit*"-Methode aufgerufen werden. Problem: Schlechte Änderbarkeit bei Änderung des Formulars.
- *Spezialisierung von TextField:* Spezielle Unterklassen von *TextField* einführen, die die Prüffunktion beinhalten. Das Formularobjekt stößt die Prüfung der einzelnen Textfelder an. Problem: Explosion der Klassen.

Strategy-Idee: Es werden spezielle Prüfklassen eingeführt, welche einen Text auf korrekte Syntax prüfen. Ein *TextField* wird mit einer Prüfklasse assoziiert und delegiert die Prüfungen an diese. Da das Prüfobjekt zur Laufzeit ausgetauscht werden kann, können auch Textfelder geprüft werden, deren Semantik sich dynamisch ändert (z.B. Stichwort, Signatur, ISBN,… bei Bibliotheksrecherche).

Anwendbarkeit:
- ♦ Eine Menge verwandter Klassen unterscheidet sich nur in ihrem Verhalten→ Konfiguration einer Klasse mit verschiedenen Verhaltensweisen.
- ♦ Es werden verschiedene Varianten eines Algorithmus benötigt → Auslagerung in separaten Klassen
- ♦ Die von einem Algorithmus (intern) benötigten Daten sollen dem Verwender verborgen bleiben → Dazu wird er in eine Klasse verpackt.
- ♦ Eine Klasse hat je nach Zustand ein unterschiedliches Verhalten, welches durch umfangreiche bedingte Anweisungen realisiert wird → Möglichkeit der Gruppierung und Auslagerung in Verhaltensklassen.
- ♦ Das Verhalten einer Klasse soll sich zur Laufzeit (von außen steuerbar) ändern

Bewertung:
Vorteile:
- Familien verwandter Algorithmen als Strategie-Hierarchie darstellbar
- Alternative zu vielen Subklassen mit unterschiedlichem Verhalten
- Erlaubt Trennung zwischen Vorgehensweise und Implementierung
- Es können unterschiedliche Implementierungen desselben Verhaltens angeboten werden (z.B. zeit- oder platzoptimiert)
- Verwender können eine Klasse mit Hilfe von *Strategies* konfigurieren → Dazu müssen sie allerdings über die Strategie und ihre Semantik Bescheid wissen → Offenlegen eines Teils der Implementierung

Nachteile:
- Kommunikations-Overhead zwischen Kontext und Strategy
- Noch mehr Objekte im System

6. Rahmenwerke, Anwendungsfamilien

Nutzen von Klassenbibliotheken
- Wiederverwendung von vorhandenem Code und Know-How
- Konzentration auf das fachlich Notwendige
- Reduktion der Entwicklungszeit
- Verwenden von erprobte Bausteinen
- Spezialisten entwickeln Klassenbibliotheken
- Standarkomponenten statt projektspezifischem Code (→ geringer Wartungsaufwand)
- Lernen am guten Beispiel (Design, Implementierung, Konventionen)

Definition: Toolbox:
Eine Toolbox ist eine Menge von wiederverwendbaren Klassen, die zueinander in Beziehung stehen. Sie sind so entworfen, daß sie nützliche und allgemein anwendbare Funktionalität zur Verfügung stellen.

Eine **Klassenbibliothek** enthält Klassen, die anwendungsunabhängige Leistungen erbringen. Sie besteht aus einem oder mehreren Clustern. Ihr Schwerpunkt liegt auf der Wiederverwendbarkeit des Codes.

Horizontale Bibliotheken bieten grundlegende Funktionalitäten an (Basis-Klassenbibliothek). Im Normalfall sind die verwendeten Bibliotheken nicht aufeinander abgestimmt, d. h. nicht vom selben Anbieter. Es gibt Varianten, in denen die verwendeten Bibliotheken aufeinander aufbauen → sie sind vom selben Anbieter.

Vertikale Bibliotheken bieten Funktionalität an, die in allen Schichten einer Anwendungsarchitektur verwendet werden können (GUI, Datenhaltung, funktonaler Teil). En Beispiel ist die ILOG – Bibliothek.
Die meisten käuflichen Bibliotheken sind horizontal.

Definition: Rahmenwerk:
Ein Rahmenwerk beschreibt eine Menge kooperierender Klassen die zusammengenommen ein wiederverwendbares Design für eine spezifische Klasse von Software zur Verfügung stellen.

Ein Rahmenwerk schreibt die Architektur einer Anwendungsklasse vor (Makro-Architektur), enthält die Design-Entscheidungen, die generell für die Anwendungsklasse gelten, definiert, welche Klassen angepaßt werden können, implementiert interne Kommuniaktionsmechanismen zwischen Klassen und definiert zentrale Verantwortlichkeiten von Klassen. Rahmenwerke können auf Klassenbibliotheken enthalten oder auf ihnen aufbauen.
Der *Schwerpunkt* liegt auf der *Wiederverwendung* von bereits bewährtem Design.
(Die Compiler-Domäne zum Beispiel ist bereits gut bekannt und beschrieben).

Unterschiede im Kontrollfluß

- *prozedurale Systeme:* Der gesamte Kontrollfluß ist im Anwendungs-Code enthalten, Operationen der darunterliegenden Schichten werden aus dem Anwendungs-Code heraus aufgerufen. Die Operationen der Basismaschine werden nur dann aktiv, wenn sie von der Anwendung gerufen werden. *Der Programmierer ist Herr des Kontrollflusses.*
- *Event Loop:* Interaktiv bedienbare Systeme führen zu einer neuen Strategie für den Kontrollfluß; Der Event-Loop wird von der Entwicklungsplattform bereitgestellt (denke an Macintosh in DIS II). Er kontrolliert die Interaktion mit dem Benutzer und ruft spezifische Teile des Anwendungscodes auf (Callbacks).
- *Rahmenwerke:* Hier wird der Kontrollfluß oft mit *Template* - Methoden realisiert. Der zentrale und überwiegende Teil des Kontrollflusses ist im Rahmenwerk implementiert. Das Rahmenwerk ruft den Code aus seinem Kontrollfluß auf → *Der Programmierer hat keinen Einfluß mehr auf den Kontrollfluß.* → Er muß die zentralen Mechanismen des Rahmenwerks verstehen und kennen.

Whitebox-Wiederverwendung
Ein Rahmenwerk wird zu einer Anwendung instanziiert, indem Unterklassen gebildet und Methoden redefiniert werden. Das Rahmenwerk definiert die zentralen Teile der Anwendung, die Beziehungen dazwischen und die Ansatzpunkte (hot spots).
→ Detailwissen der Implementierung ist erforderlich.
Dieser Ansatz ermöglicht eine mächtige Art der Wiederverwendung, da er sehr flexibel ist. Allerdings ist eine Programmierung erforderlich, die Anpassung kann nicht von einem Anwender vorgenommen werden.

Blackbox-Wiederverwendung
Das Rahmenwerk bietet verschiedene direkt benutzbare Klassen zur Instanziierung oder Parametrisierung an,
→ Wissen, wie die Klassen intern zusammenspielen, ist nicht notwendig. Teils bieten diese Rahmenwerke spezielle Werkzeuge, um die Parametrisierung vorzunehmen.
→ → Blackbox bietet folglich nur eine eingeschränkte Anpassung
Meist erfolgt eine Konvertierung von Whitebox zu Blackbox nach dem Sammeln von Erfahrungen beim Umgang mit abstrakten Rahmenwerken.

Nutzen und Probleme von Rahmenwerken
Nutzen:
- Höhere Produktivität durch Wiederverwendung von vorhandenen Komponenten und Know-How, Konzentration auf das fachlich Notwendige und Reduktion der Entwicklungstiefe
- Höhere Qualität durch Verwendung einer erprobten Plattform, die von Spezialisten entwickelt worden ist.
- Geringerer Wartungsaufwand da mit vielen gemeinsamen Standardkomponenten entwickelt wird. Daher entsteht weniger wartungsaufwendigerer projektspezifischer Code.

Probleme:
- Hoher Entwicklungsaufwand, der wirtschaftlich sein muß
- Rahmenwerk-Evolution; Änderungen in Basisklassen wirken sich auf alle Produkte aus. Es werden hohe Anforderungen an das Konfigurations- und Change-Management gestellt und Änderungen müssen im Kontext einer Produktfamilie betrachtet werden.
- Test framework-basierter Anwendungen
- Wenige Methoden und Werkzeuge, die die Entwicklung mit Frameworks unterstützen; Wenige Notationen → Gegenstand der Forschung
- Dauer des Einsatzes; Je länger ein Rahmenwerk im Einsatz ist, desto mehr wächst es. Diese Änderungen wirken sich auf die gesamte Struktur aus → Evtl. Management-Problem

Taligent's Framework-Technologie
Unterteilung in
- *Application Frameworks*
  Zusammenfassung von Funktionalitäten, die in vielen Applikationen gebraucht werden können → diese Rahmenwerke sind *horizontal* (über Applikationsdomänen) (Beispiel: GUI-Applikationen)
- *Domain Frameworks*

Zusammenfassung von Funktionalitäten, die für eine Applikationsdomäne gebraucht werden kann → *vertikale* Rahmenwerke (innerhalb einer Applikationsdomäne (Beispiel: Multimedia-Rahmenwerke, Kontrollsystem-Rahmenwerke)

- *Support Frameworks*
  Stellen systemnahe Dienste zur Verfügung
  (Beispiel: Dateizugriff, Verteilung)

Klassenbibliotheken vs. Rahmenwerke:

<> Funktionalität für allgemeine Zwecke	<> Applikationsklasse
<> Code-Wiederverwendung	<> Entwurf-Wiederverwendung
<> gleicher Code für mehrere Applikationen	<> Gleiches Design für mehrere Applikationen
<> Kontrolle liegt beim Benutzer der Bibliothek	<> Kontrolle liegt beim Rahmenwerk

Die Abgrenzung zwischen Klassenbibliotheken und Rahmenwerken entsteht evolutionär und verlangt außerdem höchste Kenntnisse im OO-Design (Design-Pattern) und Anwendungs-Know-How.
Um ein Rahmenwerk zu erstellen müssen mehrere Anwendungen der Anwendungsdomäne erstellt worden sein.
Bei Klassenbibliotheken ist das Design nicht trivial, da es für viele Anwendungssituationen gedacht ist.
Der Übergang von einer Klasse zu einer Klassenbibliothek geschieht iterativ und evolutionär, während eine Klasse relativ „einfach“ zu entwerfen und zu implementieren ist, da sie für genau eine Zweck gedacht ist → Die Entwicklung ist nicht so vorausschauend wie es bei Klassenbibliotheken notwendig ist.

7. Metapher Werkzeug & Material

7.1. Grundlagen

Eine Metapher ist ein sprachliches Ausdrucksmittel, bei dem ein bildhafter Ausdruck aus einem Kontext heraus genommen wird, und an die Stelle eines eigentlichen Ausdrucks in einem anderen Zusammenhang gestellt wird, zum Beispiel: Debugging, Garbage Collection.
Im Kontext der Software-Entwicklung können Metaphern eingesetzt werden, um Leitbilder anschaulicher zu machen. Ein Beispiel hierfür ist die *Desktop-Metapher*.

Eine ***Entwurfsmetapher*** ist eine bildhafte, gegenständliche Vorstellung, die ein Leitbild fachlich und konstruktiv ausgestaltet → trägt zur *Begriffsbildung* bei und leitet die Analyse und Modellierung von Systemen.

Ein ***Leitbild*** ist eine benennbare, grundsätzliche Sichtweise, anhand derer ein Ausschnitt der Realität wahrgenommen verstanden und gestaltet werden kann.
Im Kontext der Software-Entwicklung soll ein Leitbild helfen, das fachliche Modell eines Anwendungsbereichs zu erstellen und in den Entwurf eines zukünftigen Systems zu übertragen. So gibt es im Entwicklungsprozeß einen gemeinsamen Orientierungsrahmen vor („woran arbeiten wir eigentlich?“ – Alle sollen dasselbe verstehen und über dasselbe reden).

So ausgestaltete Leitbilder helfen jenseits der eingesetzten OO-Analyse und Design-Methode das Anwendungsfeld zu strukturieren und fachlich sowie softwaretechnisch vernünftige Begriffs- und Klassenstrukturen zu bilden.

Entwurfsmetaphern sind Werkzeug und Material.
Diese Metaphern sind besonders geeignet, um interaktive Arbeitsplatzsysteme objektorientiert zu realisieren. → Als natürlich nachempfundener Umgang mit Arbeitsmitteln und Arbeitsgegenständen. Software, die nach diesem Prinzip entwickelt wird, kann besser verstanden werden.

Werkzeuge sind Gegenstände, mit denen Menschen im Rahmen einer Aufgabe Materialien verändern oder sondieren können. Teilweise eigenen sie sich für unterschiedliche Zwecke, Dazu müssen sie geeignet bedient werden → Lernfaktor. Werkzeuge können selbst wieder Material sein, das mit anderen Werkzeugen bearbeitet wird.
SW-Beispiele: Editoren, Werkzeuge zum Suchen von Dateien, Explorer

Materialien sind Gegenstände, die im Rahmen einer Aufgabe Teil des Arbeitsergebnisses werden. Sie werden durch Werkzeuge (und Automaten) bearbeitet und müssen so beschaffen sein, daß sie von Werkzeugen bearbeitet werden können. Ein Material selbst kann Werkzeugeigenschaften haben.
Beispiele: Rechnung, Mahnung, Papier, Holz, Stahl, etc… (SW: Dateien verschiedener Typen)

Bei der objektorientierten Konstruktion werden interaktive Arbeitsplatzsysteme modelliert, indem Werkzeuge und Materialien identifiziert werden. Softwaretechnisch führt dies zu ***Werkzeug-, Material-,*** und ***Aspektklassen***.

Materialklasse:
Fachlich: Werden im Rahmen einer Anwendung zu Arbeitsgegenständen und lassen sich nach ihrer spezifischen Art und Weise bearbeiten. Solche Arbeiten werden mit unterschiedlichen Werkzeugen ausgeführt.
Technisch: Eine Materialklasse besitzt im Rahmen einer Anwendung keine eigenständige Ein-/Ausgaberoutinen. Sie stellt nur die Operationen zur Verfügung, die zum Verändern des Materials notwendig sind.

Werkzeugklasse:
Fachlich: Dinge, die im Rahmen einer Anwendung zum Arbeitsmittel werden, werden in Werkzeugklassen angeordnet. Werkzeuge werden zum Bearbeiten von Material verwendet. Ein Werkzeug kann für unterschiedliche Materialien geeignet sein (Beispiel: Editor, ACDSee).
Technisch: Werkzeugklassen beschreiben die Umgangsformen eines Benutzers mit einer Anwendung, Sie realisieren die interaktive Komponente einer Anwendung.

Bei der Strukturierung von Werkzeug- und Materialklassen kann es zu Problemen kommen, wie das Beispiel eines Druckers (Werkzeug) und verschiedener Dokumente verdeutlicht. Bei einer direkten Modellierung müßte jedes Material eine eigene (angepaßte) Druckroutine beinhalten. Eine Umstrukturierung des Materials hilft hier auch nicht weiter, da die Druckroutine immer noch spezifisch angepaßt werden muß.

→ Hier kommen die ***Aspektklassen*** in Betracht:

fachlich: Es existieren bestimmte Beziehungen zwischen Werkzeugen und Materialien; sie drücken das Zusammenpassen von Werkzeug und Material aus. Dies wird durch Aspektklassen modelliert.
Technisch: Aspektklassen legen Schnittstellen zwischen Werkzeug. Und Materialklassen fest. Jede Werkzeugklasse benutzt nur die Operationen der entsprechenden Aspektklasse, um auf Materialien zuzugreifen. Jede Materialklasse implementiert alle Operationen ihrer Aspektklasse, damit die Werkzeuge diese verwenden können.
→ Aspektklassen führen im allgemeinen zu Mehrfachvererbung.
Oft geben Aspektklassen nur ***Spezifikationen*** an und sind i.a. nicht ausführbar.
→ Somit dienen sie zur Festlegung von Schnittstellen, die das Zusammenspiel Funktionieren) von Werkzeug und Material sicherstellen.

Im Druckbeispiel wird nun die Aspektklasse *druckbar* eingefügt, die virtuelle Methoden zur Verfügung stellt, die das Werkzeug *Drucker* für seine Operationen verwenden kann. Die Ausgestaltung der Methoden geschieht in den einzelnen Materialklassen wie *Brief* oder *Formular*. Hier können die unterschiedlichen Druckersteuerungen ausgestaltet werden.

Mehrfachvererbung: *Brief* erbt zum Beispiel vom Aspekt *druckbar* und *editierbar*.

Geeignete Vorgehensweise:
- Entwirf Materialklassen entlang den vertrauten Gegenständen der realen Welt
- Entwirf Werkzeuge, die den aktuellen Arbeitserfordernissen entsprechen
- Stelle den Zusammenhang von Werkzeugen und Materialien in Aspektklassen dar
- Verwende die Aspektklassen als Rahmen für die Integration neuer Werkzeuge und Materialien

7.2. WAM Entwurfsmuster

WAM ist eine *praxiserprobte* Konstruktionslehre, die Hilfsmittel in Form von Mustern auf verschiedenen Ebenen zur Verfügung stellt → Interpretations- und Entwurfsmuster. Der Ansatz stützt sich auf bekannte (GOF-)Muster und bietet starke Ansätze für die Realisierung von Frameworks. Hierfür sind definierte Muster wie IAK und FK eine Basis. Weiterhin stellt der WAM-Ansatz eine Begrifflichkeit für den Entwurf zur Verfügung.

Mit Hilfe des WAM-Ansatzes sollen Muster und Konstruktionstechniken definiert werden, die dem Entwickler helfen sollen, auf Basis einer Systemanalyse zu adäquaten objektorientierten Software-Architekturen zu gelangen. Der Ansatz bezieht sich auf die Anwendungsdomäne, speziell auf interaktive Anwendungen im Bereich des Arbeitsplatzes für qualifizierte menschliche Arbeitstätigkeiten.
Somit steht der Ansatz zwischen den top-level-Konzeptionsmustern und den codenahen Programmiermustern.

Konzeptionsmuster
Ein Konzeptionsmuster ist ein Muster, welches zur Interpretation und Gestaltung von tatsächlichen oder antizipierten Anwendungssituationen und Softwaresystemen verwendet werden kann.

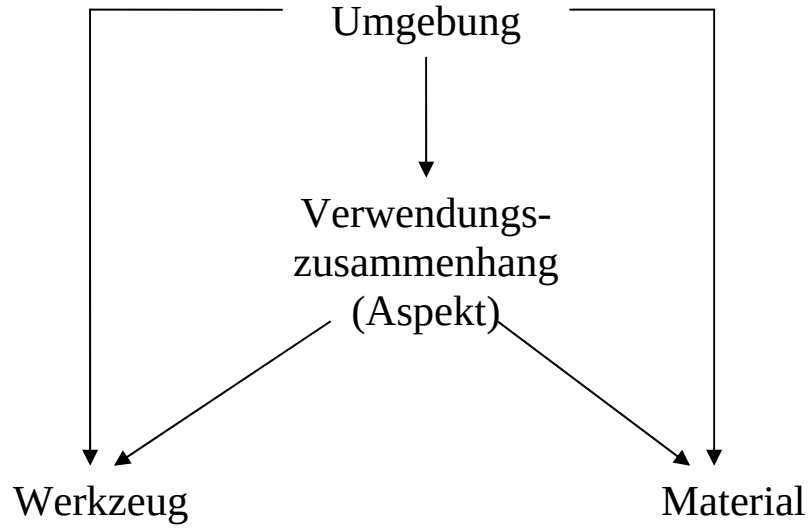
Sie dienen dazu, sich in der Anwendungswelt zu orientieren und erläutern, wie Anwendungssituationen interpretiert werden sollen. Darüber hinaus stellen sie eine passende Begrifflichkeit dazu bereit. So kann die Anwendungswelt mit den Benutzern erfaßt und interpretiert werden. → Ausreichende Nähe zum Software-Entwurf, somit ist ein Übergang leicht möglich.

Entwurfsmuster
Ein Entwurfsmuster ist ein Muster, das als Vorlage für die Konstruktion eines softwaretechnischen Entwurfs dient. Ein Entwurfsmuster besteht aus dem Zusammenspiel technischer Elemente wie Klassen, Objekte und Operationen, Die Struktur und Dynamik eines Entwurfsmusters klärt die beteiligten Komponenten, ihre Zusammenarbeit und die Verteilung der Zuständigkeiten.

Entwurfsmuster sind effiziente vorlagen für den Entwurf und erleichtern die Kommunikation, Sie stellen Mikro-Architektuen dar, die flexibel kombiniert werden können und erleichtern die Einarbeitung und das Verständnis von Software. Sie sind für den Entwickler gedacht.

Programmiermuster
Ein Programmiermuster ist ein Muster, das als Vorlage für die Implementierung eines Entwurfs dient. Programmiermuster basieren auf Erfahrungswissen in der Programmierung und sind in der jeweiligen Programmierkultur weitgehend bekannt.

Ihre Verwendung geschieht teils unbewußt und häufig in Styleguides formuliert. (Apple, Smalltalk).



Umgebung
Werkzeuge befinden sich immer in einer Umgebung (z.B. Schreibtisch). Die Umgebung ermöglicht die Umsetzung gewohnter räumlicher Organisationsformen in logische Dimensionen (Beispiel: Plazierung von Icons).
Die fachlichen Abhängigkeiten zwischen Materialien und ihre Konsistenz werden sichergestellt (Beispiel: Konto und Kreditvertrag).
Die Umgebung kontrolliert auch, ob die Anwendung eines Werkzeugs auf einem bestimmten Material möglich ist (Beispiel: Ziehen eines Dokuments auf ein Drucker Icon).

7.2.1 Entwurfsmuster zur Werkzeugkonstruktion

(vgl. Abbildung auf Folie 10)

Werkzeug. Und Materialkopplung
Werkzeuge werden auf Materialien angewandt, um Arbeitstätigkeiten durchzuführen. Diese Arbeitstätigkeiten werden im Verwendungszusammenhang beschrieben und durch Aspekte

formalisiert. Dieses Muster definiert die software-technische Lösung für die Kopplung von Werkzeugen an Materialien.
→ Aspekte koppeln Werkzeuge an Materialien
→ Aspekte werden als Schnittstellenklassen oder als abstrakte Klassen formuliert.
→ Die Materialschnittstelle ergibt sich als Vereinigung der Aspektschnittstellen, die die möglichen Verwendungszusammenhänge des Materials beschreiben.
→ Die Materialklasse erbt alle Aspektschnittstellen
→ Werkzeuge benutzen ausschließlich Aspekte, um Materialien zu bearbeiten.
Beispiel: *Auflister* (Werkzeug), *Auflistbar*, *Druckbar*, *Speicherbar* (Aspekte), *TerminBuch*(Material).

Werkzeugkomposition
Es ist mühsam, Werkzeuge immer neu zu konstruieren → Wiederverwendbare Zusammensetzung aus Teilwerkzeugen → Grundlegende Werkzeuge, die zur Kombination geeignet sind, müssen vorhanden sein → Strategie: Baue Werkzeuge aus Subwerkzeugen zusammen → → Werkzeugbaum. Eine komplexe Aufgabe wird somit von einem zusammengesetzten Werkzeug erledigt; Es delegiert Aufgaben an Subwerkzeuge und interpretiert deren Ergebnisse (Kontextwerkzeug) und erzeugt, verwendet oder löscht Subwerkzeuge der Aufgabe entsprechend.

Ein Werkzeugbaum kann mit Hilfe des Entwurfsmusters *Composite* beschrieben werden. Es erlaubt die Konstruktion von rekursiven Baumstrukturen und en uniformen Umgang mit den Blättern des Baumes. Einfache Werkzeuge sind die Blätter eines Baumes.

Trennung von Interaktion und Funktion
Aus Erfahrung ist bekannt, daß Probleme bzgl. Der Weiterentwicklung und Anpassung entstehen, sobald die Interaktion und Funktionalität eines Werkzeugs vermischt werden.
→ Daher ist eine Trennung sinnvoll; Die Interaktion ist von der Benutzungsumgebung abhängig (Maus, etc.) und die Funktionalität entspricht der Arbeitsaufgabe eines Werkzeugs. Sie ist auch abhängig von dem zu bearbeitenden Material.

Strategie: Baue ein Werkzeug aus einer oder mehreren Interaktionskomponenten (IAK) und aus einer Funktionskomponente (FK) zusammen. Die FK speichert den Interaktionszustand des Werkzeugs (z.B. Selektion), die IAK verwendet nur die Operationen der FK.

Eine FK kann mehrere IAKs haben. Sie basiert auf mehreren Aspektschnittstellen. Die IAK benutzt atomare Interaktionsformen (Widgets).
Schema: IAK benutzt Widget. FK wird dargestellt durch IAK, basiert selbst auf Aspektschnittstelle.

Werkzeugkopplung
Besteht ein Werkzeug aus mehreren Subwerkzeugen, so müssen diese an die IAKs und an die FK angebunden werden. Der Werkzeugbaum wird durch die FKs gebildet. Eine KontextIAK muß nicht mit ihren SubIAKs zusammenarbeiten. KontextIAKs und FK beobachten ihre SubFks.

7.2.2 Entwurfsmuster zur Werkzeugintegration

(vgl. Folie 24)

Ab hier können noch mal alle Folien durchgegangen werden. Der Text eignet sich nicht zur Zusammenfassung.

8. Persistenz von Objekten

8.1. Persistenz von Objekten (Serialisierung)

Objektorient konstruierte Anwendungssysteme müssen auch die dauerhafte Speicherung von Objekte berücksichtigen (zum Beispiel Kundendatensätze in einem Betreuungswerkzeug).
Es gibt verschiedene Möglichkeiten, eine solche Speicherung zu realisieren:

- Eigene Datenhaltung auf der Basis des Dateisystems (Serialisierung in JAVA, Bytestreaming in Smalltalk)
- Nutzen einer objektorientierten Datenbank (Objektstruktur wird direkt in der Datenbank gespeichert inklusive den Operationen auf den Objekten)
- Anbindung an eine relationale Datenbank (diese sind in vielen Fällen vorhanden und *müssen* benutzt werden)

JAVA stellt einen Mechanismus zur Verfügung, um Objekte in Byteströme umzuwandeln und diese wieder aus solchen Strömen zu rekonstruieren. Die Ströme können als Dateien abgelegt werden. Sie werden als Mechanismus für die *Persistenz* und *Übertragung* von Objekten in Netzwerken verwendet.
Sollen Objekte einer Klasse serialisiert werden, so muß die betroffene Klasse das Interface *Serializable* implementieren. Dieses Interface dient lediglich zur *Markierung* und enthält folglich keine Methoden und Variablen. Die Implementierung wird von den betroffenen Klassen selbst realisiert. *Alle Variablen, die weder als static noch transient deklariert werden, werden serialisiert.*
Die Serialisierung scheitert, wenn Variablen auf Objekte zeigen, die nicht serialisierbar sind.

Vorgang der Serialisierung:
Um serialisierte Objekte in einer Datei zu speichern, wird ein Objekt der Klasse *FileOutputStream* benötigt. Ein Objekt der Klasse *ObjectOutputStream* legt die serialisierten Objekte auf den *FileOutputStream*. Zur eigentlichen Serialisierung bietet die Klasse *ObjectOutputStream* die Operationen *writeObject, writeBoolean, writeInt*, etc. an.
Jedes Objekt wird nur einmal serialisiert.

Objekte werden nach dem FIFO-Prinzip serialisiert.

Vorgang der Deserialisierung:
Um serialisierte Objekte wieder zu rekonstruieren, wird ein Objekt der Klasse *ObjectInputStream* benötigt, das auf einem *FileInputStream* arbeitet.
Die Deserialisierung geschieht mit Hilfe der Operationen *readObject, readInt, readBoolean,* etc.
Ist die Klasse des wiederherzustellenden Objekts unbekannt, dann wird die Ausnahme *ClassNotFoundExeption*aktiviert.
Wird *readObject* verwendet, so muß eine explizite Cast-Operation durchgeführt werden.

Quellcodebeispiele sind in den Folien zu finden!

Werkzeug zur Speicherung:
Die Art und Weise, wie die Datensätze gespeichert werden, sollte nicht im Kundenwerkzeug bekannt sein → Separation of concerns.
Daher wird ein einfaches Persistenzwerkzeug konstruiert, welches mit dem Material Kundenwerkzeug (als Beispiel) arbeitet.

8.2. Anbindung an relationale Datenbanken

Wie können Objekte und ihre Strukturen in einem relationalen DB-System gespeichert werden? RDBMSe kennen nur einfache Datentypen (Interger, Character, etc.) und die Informationen werden in Tabellen gespeichert.
Als Konsequenz dieser Eigenschaften können nur *Objektzustände* gespeichert werden.
Komplexe Strukturen können *nicht direkt* gespeichert werden.
Wie sollen nun Objektstrukturen in geeignete Tabellenstrukturen transformiert werden, die Vererbung abgebildet und die Eigenschaft, persistent zu sein, abgebildet werden?

Strukturkonflikt

Objekorientiert	***Relational***
Objekt	Tupel, Record, Zeile
Attribut	Attribut, Geld, Spalte
Klasse	Relation, Tabelle
Basisklasse	Domäne, Attributtyp
Systemweite Identität	Primärschlüssel
Objekt enthält Objekte	Tupel enthält Attribute (Fremd-Schlüssel)

Basisklassen und Domänen
Domänen sind vom DB-System vorgegeben und repräsentieren Werte und Wertemengen für Felder (INTEGER, MONEY, DATE, CHAR,…).
OO-Sprachen geben Basisklassen (Basisdatentypen) vor, die Werte repräsentieren.
→ Die Abbildung von Basisklassen in Domänen erfolgt, indem unter Verwendung der Basisklassen fehlende, den Domänen korrespondierende Werteklassen gebildet werden.
Das Ziel ist, bei der Konvertierung eines Objekts in die relationale Welt umgekehrt, sicherzustellen, da0 nur erlaubte Werte existieren.

Modellierung der Persistenz
Annahme: Damit Objekte einer Klasse in einer Datenbank persistent gespeichert werden können, erbt sie das dazu notwenige Verhalten (definiert in der Klasse *DBSpeicherbar*).

Entwurf: Zu jeder speicherbaren Klasse wird eine Klasse definiert die die Konversion von OO-Werten in Domänenwerte definiert und die die Abbildung von OO-Attributnamen auf Feldnamen vornimmt.

Objektidentität
Jedes speicherbare Objekt erhält in der Datenbank einen systemweit eindeutigen Primärschlüssel, welcher vom Datenbanksystem erzeugt wird. Mit der Objekt-ID wir in der der Klasse zugeordneten Tabelle der Datenbank ein Record angelegt. Die Objekt-ID dient zur Rekonstruktion der Objektidentität in der Datenbank. *Nur die Objekt-ID* ist Primärschlüssel des Objekts.

Abbildung der Vererbung
Einfache Klassen bestehen neben den Operationen nur aus Attributen, die Werte von Basisklassen besitzen.
Für die Abbildung wird für jede einfache Klasse eine Tabelle angelegt. Für jedes Exemplarattribut (das nicht abgeleitet ist) wird ein DB-Attribut definiert und die Basisklassen werden auf DB-Typen (Domänen) abgebildet. Jede Tabelle enthält als Primärschlüssel die Objekt-ID.

Klassen mit eingebetteten Objekten enthalten Attribute, deren Wert statisch alloziierte Verbunde sind (also keine Objekte im eigentlichen Sinn).
Die Abbildung erfolgt durch eine rekursive Auflösung dieser Attribute und eine Zerlegung in elementare Komponenten. Für jede elementare Komponente wird ein DB-Attribut in der Tabelle erzeugt.

Klassen mit Objektreferenzen besitzen wenigstens ein Attribut, welches eine Referenz auf ein Objekt einer anderen Klasse ist.
Abbildung: Für jedes Attribut, das eine Objektreferenz ist, wird ein DB-Attribut aufgenommen. Der Bezeichner des Attributs ist der Tabellenname, der Wert ist die entsprechende Objekt-ID dieser Relation.

Beziehungen zwischen Objekten
Objektorientierte Systeme weisen in der Regel eine große Anzahl relativ einfacher Klassen auf, die über sehr komplexe Beziehungen miteinander interagieren (zum Beispiel???).

In relationalen Systemen werden Beziehungen mit Hilfe von Fremd-Schlüsseln hergestellt. Diese werden mit Hilfe von Select- oder Koin-Operationen der Datenbank verfolgt → aufwendig.
→ ! Hier gilt besondere Vorsicht bei Paradigmenwechsel!
→ Abweichen von der Formel: 1 Klasse = 1 Tabelle. Dies ist insbesondere bei Aggregationen sinnvoll.

Beispiele zu den Transformationen finden sich auf den Folien.

Übersicht:

Technik- neutral	Wiederverwendbar in der Domäne	Sehr gut wiederverwendbar
Technik-spezifisch	Schlecht wiederverwendbar, schlecht wartbar	Wiederverwendbar, wenn Technik eingesetzt wird (z.B. immer DBMS,…)
	Anwedungsspezifisch	anwendungsneutral

9. Refactoring

Um die Design-Qualität zu halten oder zu steigern, muß die Architektur stetig verbessert werden, denn diese degeneriert langsam dadurch, daß neue Anforderungen eingebracht werden.
Refactoring ist eine Technik, um das Design in kleinen überschaubaren Schritten zu verbessern. Natürlich darf sich die Funktionalität dabei nicht ändern.
Das ***systematische Refactoring*** ist eingebettet in systematische Tests; Diese müssen regressionsfähig und automatisch durchführbar sein.

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Oft wird Software in ein Unternehmen übernommen, die für das Unternehmen wichtig und wertvoll ist → ***Legacy-Systeme***. Allerdings sind die Entwickler dieser Systeme meist nicht mehr verfügbar, die damals eingesetzten Methoden sind veraltet, das System ist seit seiner Entwicklung heftig modifiziert worden und / oder die Produktdokumentation fehlt oder ist veraltet. Dies führt dazu, daß eine Weiterentwicklung extrem aufwendig und somit auch extrem teuer wird.

Allerdings zeigt jedes erfolgreiceh Softwaresystem nach einiger Zeit Symptome eines Legacy-Systems. Ein OO-Legacy-System ist ein erfolgreiches OO-System, dessen Architektur nicht mehr den Änderungen gewachsen ist und dementsprechend degeneriert. Die von der Objektorientierung versprochenen Vorteile (Flexibilität, Wiederverwendbarkeit oder z.B. Wartbarkeit,…) werden nicht umsonst erhalten. Auch OO-Systeme müssen kontinuierlich re-engineered werden. Nur das schafft die Voraussetzung dafür, daß die Systeme flexibel und wartbar bleiben.

Forward Engineering
Traditioneller Prozeß von high-level Abstraktionen und logischen, implementierungsunabhängigen Entwürfen zur physikalischen Implementierung eines Systems.

Reverse Engineering
Prozeß der Analyse eines Systems, um die Komponenten und ihre Beziehungen untereinander zu identifizieren und um Repräsentationen des Systems in einer anderen Form oder einer high-level Abstraktion zu erstellen.

Ziele:

- *Umgang mit Komplexität:* Notwendigkeiten von Techniken, um große, hoch-komplexe Systeme zu verstehen.
- *Wiederauffinden von verlorenen Informationen:* Welche Änderungen wurden warum gemacht?
- *Identifikation von Seiteneffekten:* Hilfe, Auswirkungen von Änderungen auf das gesamte System zu verstehen.
- *Erzeugen von höheren Abstraktionen:* Identifizierung von im Code versteckten Abstraktionen
- *Verbesserung der Wiederverwendung:* Identifikation von Komponenten, die vielleicht wiederverwendbar sind

Techniken:

- *Neudokumentierung:*
- *Wiederaufdecken des Entwurfs:* Werkzeuge: Software-Metriken, Browser und Visualisierungs-Tools, statische Analysetools, dynamische (trace) Analysetools.

Reengineering

Untersuchung und Veränderung eines Zielsystems um es in einer neuen Form und neuer Implementierung mit evtl. erweiterter Funktionalität neu aufzubauen.

Ziele:

- *Modularisierung:* Aufteilen eines monolithischen Systems in separate Teile, die unabhängig voneinander gepflegt werden können
- *Performance:* Schrittweise Verfeinerung einer lauffähigen Version hin zur Leistung
- *Verbesserung der Architektur:* Erleichtert die Wartbar- und Portierbarkeit
- *Ausnutzen neuer Technologien:* z.B. neuer Sprachkonstrukte, neuer Klassenbibliotheken, etc.

Techniken:

- *Neustrukturierung* der Systemrepräsentation, die einen besseren Überblick gibt, die Semantik und das Verhalten des Originals aber beibehält.
- *Data reengineering:* Reorganisation der Datenstrukturen (Duplikate, Umstrukturierung).
- *Refactoring:* → Restrukturierung innerhalb eines objektorientierten Kontextes → Umbenennen/Verschieben von Methoden/Klassen etc.

Ein Beispiel einer Analyse und anschließender Neustrukturierung eines Systems um eine neue Anforderung einzufügen, wir in den Folien gegeben.

Refactoring-Regeln

1.) *Soll eine neue Funktion in ein bestehendes System eingebracht werden und der vorhandene Code sowie das Design erlauben nicht, daß dies einfach möglich ist, dann verändere den Code und das Design so, daß es einfach wird, die neue Funktion einzubringen. Realisiere erst danach die neue Funktion*
2.) *Bevor der Code und der Entwurf eines bestehenden Systems verändert werden, müssen entsprechende Testfälle definiert sein. Mit diesen muß geprüft werden, ob die Funktionalität nach den Änderungen unverändert ist.*
3.) *Im Zuge der Codeverbesserung sollten überall dorrt neue Bezeichner eingeführt werden, wenn die vorhandenen Bezeichner nicht präzise genug das ausdrücken, was mit ihnen gemeint ist. (guter Code ist deshalb lesbar, weil Bezeichner viel über ihre Semantik aussagen und Codefragmente selbsterklärend sein sollen).*

Die durchgeführten Refactoring-Schritte sollten klein und überschaubar sein. „Nicht überall gleichzeitig anfangen."

Das Refactoring wirkt sich meistens auf die Effizienz des Systems aus (Effizienz ist eine wichtige Eigenschaft!). Entwickle das System zuerst in guter Design-Qualität und messe dann, wo Effizienz-Probleme auftreten. Optimiere das System danach, bis die Effizienz ausreichend ist.

Ansätze, um effiziente Systeme zu erstellen

- *Zeit-Budgetierung:* Definiert von Zeitschranken für jede Komponente bei der Entwicklung (Anwendung in Real-Time-Systemen).
- *Kontinuierliche Effizienzbetrachtung:* Jeder Entwickler versucht seinen Bereich zu optimieren.
- *Messen und Optimieren:* Messen der Hot Spots bzgl. Effizienz → Refactoring mit dem Ziel, die Effizienz zu verbessern.

Symptome schlechter Design-Qualität

- Sehr lange Klassen → Aufteilen der Klassen, Unterklassen bilden
- Duplizierter Code → Extrahieren einer eigenen Methode, Einführen von Template-Methoden.
- CASE – Anweisung → Einsatz von Polymorphismus (Vererbungshierarchie), Typ-Merkmale werden durch das Muster Zustand oder Strategie ersetzt.
- Aufwendige Kommentierung → Prozedurale Abstraktion, verwenden von Zusicherungen.

10. Unified Process

Bevor ein Projekt gestartet werden kann, müssen die Verantwortlichkeiten und Zuständigkeiten geklärt sein.

- Welche Mitarbeiter mit welchen Qualifikationen werden benötigt (Rollen)
- Nach welchem Prozeßmodell wird vorgegangen (Vorgehensweise)
- Sind bereits Vorgehensmodelle vorhanden, die verändert oder angepaßt werden können?
- Welche Sprachen, Methoden und Werkzeuge sollen eingesetzt werden?

Das Unified Process Modell ist im einzelnen auf den Folien nachzulesen.

Diskussion

- Sind alle Rollen wirklich nötig? – Abhängigkeit vom Umfang des Projekts
- Gut: dynamischer und anpaßbarer Ansatz
- Contra: An manchen Stellen wird der Prozeß übernommen, an anderen wird er als Vorbild verwendet, um essentielle Ansätze zu extrahieren.

11. Extreme Programming

Die Extreme Programmierung sieht den industriellen Fertigungsprozeß für die Softwareerstellung als ungeeignetes Mittel. Die Softwareerstellung soll als Dialog zwischen Kunden und Entwicklung verstanden werden.
→ Zurückbesinnung auf das Wesentliche.
Frage: Wie würde programmiert, wenn genug Zeit vorhanden wäre?

XP beruht auf vier untereinander in Beziehung stehenden Werten:

- Kommunikation
- Feedback
- Einfachheit
- Mut

Das Wasserfallmodell wird als ungeeignet angesehen, ein zyklisches Schema wird bevorzugt. Dieses ist nicht viel teurer, dafür effektiver (gilt zumindest bei mittleren Projekten).
→ Denke hier an DIA-Cycle von DIS I & II!

Prinzipien:

- Programmieren in Paaren
- Komponententests
- Fortlaufende Integration (nur bei Entwicklung in kleinen Systemen möglich)
- Einfaches Design
- Refaktorisieren
- Gemeinsame Verantwortlichkeit
- Metapher-basierter Entwurf

→ XP benötigt Disziplin und Prinzipientreue, damit trotz der gelockerten Vorgaben kein Chaos entsteht.

Aus ökonomischer Sicht wird beim XP Geld langsamer ausgegeben und es werden schneller Einnahmen erzielt. Weiterhin verlängert sich die produktive Lebensdauer des Projekts.

Voraussetzungen für das Extreme Programming:

- Entwickler arbeiten nicht räumlich verteilt
- Direkte Kommunikation ist möglich
- Teamgröße 10-15 Personen
- Möglichkeit, mit der eingesetzten Programmiersprachen lesbaren Code zu schreiben
- Gute Verfügbarkeit der Kunden (räumliche Nähe)
- Alle Beteiligten müssen XP akzeptieren

Nutzen:

- wechselnde Anforderungen werden berücksichtigt
- Relativ schnelle Erstellung einer nützlichen Version
- Auch durchschnittliche Entwickler verbessern dadurch ihre Produktivität

Kritik

- Fehlen von zentralen Dokumenten
- Traceability
- Fehlende Studien