
YET ANOTHER SWT SUMMARY

Einleitung

Anmerkung zur Zusammenfassung

Dies ist der Versuch einer Zusammenfassung der Vorlesung SOFTWARETECHNIK aus dem Wintersemester 2005/06 an der Universität Karlsruhe (TH). Sie erhebt weder Anspruch auf Vollständigkeit, noch auf Korrektheit. Dieses Dokument hält sich sehr stark an den Folien von Prof. Dr. Walter F. Tichy. Bei den Entwurfsmustern hab ich die UML-Diagramme der [wikipedia](#)-Enzyklopädie entnommen.

Bei Fehlern würde ich mich über eine eMail an adam.urban@gmail.com freuen.

Links

Meine Homepage:

adam.urban.de.vu

Offizielle Softwaretechnik Seite:

<http://www.ipd.uka.de/Tichy/teaching.php?id=111>

für die Community

*”Gebildet ist, wer weiß,
wo er findet, was er nicht weiß.”*

G. Simmel

Inhaltsverzeichnis

1 Die Planungsphase	8
1.1 Planen des Produktes	8
1.2 Lastenheft	8
1.2.1 Gliederungsschema	9
1.3 Anforderungsermittlung	9
2 Die Definitionsphase	10
2.1 Aufbau eines Pflichtenheftes	10
2.1.1 Gliederungsschema	10
2.2 Objektorientierte Softwareentwicklung	10
2.3 UML Diagramme	12
3 Die Entwurfsphase	13
3.1 Ziele und Aufgaben des Entwurfs	13
3.2 Modularer Entwurf	13
3.2.1 Anforderungen an das Modulkonzept	13
3.2.2 Das Modul	14
3.2.3 Allgemeine Methoden der Zerlegung	14
3.3 Entwurfsmuster	15
3.3.1 Beobachter	15
3.3.2 Kompositum	16
3.3.3 Strategie	17
3.4 Entkoppelungs-Muster	18
3.4.1 Datenablage (Repository)	18
3.4.2 Iterator	18
3.4.3 Schichtenarchitektur	19
3.4.4 Vermittler (Mediator)	19
3.4.5 Brücke (Bridge)	20
3.4.6 Adapter	21
3.4.7 Stellvertreter (Proxy)	22
3.4.8 Fließband (Pipes and Filters)	22
3.4.9 Ereigniskanal (Event Channel)	23
3.4.10 Rahmenarchitektur (Framework)	23
3.5 Varianten-Muster	23
3.5.1 Oberklasse	23
3.5.2 Besucher (Visitor)	24
3.5.3 Schablonenmethode	24
3.5.4 Fabrikmethode	25
3.5.5 Erbauer	25
3.5.6 Abstrakte Fabrik	26
3.6 Zustandshandhabungs-Muster	26
3.6.1 Memento	26
3.6.2 Prototyp	27
3.6.3 Fliegengewicht (Flyweight)	27
3.7 Steuerungs-Muster	28
3.7.1 Tafel (Blackboard)	28
3.7.2 Befehl (Command)	28
3.7.3 Zuständigkeitskette (Chain of Responsibility)	28
3.7.4 Master/Slave	29
3.7.5 Prozess-Steuerung	29
3.8 Virtuelle Maschinen	29
3.8.1 Interpretierer	29
3.8.2 Regelbasierter Interpretierer	29
3.9 Bequemlichkeits-Muster	30

3.9.1	Bequemlichkeits-Methode	30
3.9.2	Bequemlichkeits-Klasse	30
3.9.3	Fassade	30
3.9.4	Null-Objekt	30
4	Die Implementierungsphase	31
4.1	Einführung und Überblick	31
4.2	Programmoptimierung	31
4.2.1	Laufzeitreduktion	31
4.2.2	Speicherplatzreduktion	32
4.3	Programmier-Richtlinien	33
5	Testen & Prüfen	34
5.1	Modul- / Softwaretestverfahren	34
5.2	Klassifikation testender Verfahren	34
5.2.1	Kontrollflussorientierte Testverfahren	35
5.2.2	Funktionale Tests	37
5.2.3	Leistungstests	37
5.3	Software-Inspektionen	37
5.4	Testwerkzeuge	38
6	Die Abnahme- & Einführungsphase	39
6.1	Die Abnahmephase	39
6.2	Die Einführungsphase	39
7	Die Wartungs- & Pflegephase	41
7.1	Aufgaben und ihr Aufwand	41
7.1.1	4 Kategorien der Wartungs- & Pflegephase	41
7.2	Aufwandsschätzung	42
7.2.1	Funktionspunkte	42
8	Prozessmodelle	43
8.1	Agiles Vorgehensmodell Extreme Programming XP	43
8.1.1	XP-Praktiken	43
8.1.2	Kritik an XP	44
8.1.3	Zusammenfassung	45
9	Konfigurationsverwaltung	46
9.1	Grundzüge der Konfigurationsverwaltung (KV)	46
9.2	Revision Control System (RCS)	48
9.3	Concurrent Version System (CVS)	48
10	Einführung in XML 1.0	49
10.1	Allgemeines	49
10.2	Einführung	49

1 Die Planungsphase

1.1 Planen des Produktes

- Auswählen des Produktes
 - Trendstudien
 - Marktanalysen
 - Forschungsergebnisse
 - Kundenanfragen
 - Vorentwicklung
- Voruntersuchung des Produkts
 - u.U. gezielte Ist-Aufnahme, wenn bereits Vorgängerprodukt vorhanden; anschliessend Ist-Analyse
 - Festlegen der Hauptanforderungen (Hauptfunktionen, Hauptdaten, Hauptleistungen, wichtigste Aspekte der Benutzerschnittstelle, wichtigste Qualitätsmerkmale)
- Durchführbarkeitsuntersuchung
 - Prüfen der fachlichen Durchführbarkeit (softwaretechnische Realisierbarkeit, Verfügbarkeit Entwicklungs- und Zielmaschinen)
 - Prüfen alternativer Lösungsvorschläge
 - Prüfen der personellen Durchführbarkeit (Verfügbarkeit qualifizierter Fachkräfte für die Entwicklung)
 - Prüfen der Risiken
- Prüfen der ökonomischen Durchführbarkeit (Aufwands- und Termschätzung, Wirtschaftlichkeitsrechnung)
- Rechtliche Gesichtspunkte (Datenschutz, Zertifizierung, relevante Standards)

Die Ergebnisse dieser Tätigkeit: **Durchführbarkeitsstudie** (Lastenheft (grobes Pflichtenheft), Projektkalkulation, Projektplan)

1.2 Lastenheft

Das Lastenheft beschreibt die Eigenschaften, die das Produkt aus der Sicht des Kunden erfüllen soll.

Funktionale Eigenschaften: Die Dienste, die das Produkt zur Verfügung stellt; z.B. die Reaktion des Produkts auf bestimmte Eingaben oder Situationen; manchmal auch, was das Produkt nicht tun sollte

Nicht funktionale Eigenschaften: Einschränkungen der Dienste, Einsatzgebiete, Interaktion mit anderen Systemen, Antwortzeitverhalten, Speicherbedarf, Zuverlässigkeit, Standards, rechtliche Gesichtspunkte, etc.

Das **Lastenheft** enthält die Hauptanforderungen an das Produkt, formuliert mit natürlicher Sprache, evtl. Diagramme. Das **Pflichtenheft** (in der Definitionsphase) ist ausführlicher und genauer. Es spezifiziert die Anforderungen in eindeutiger Weise, so dass sie implementiert werden können. Das Lastenheft dient der Kommunikation mit dem Kunden und der Projektplanung; das Pflichtenheft ist die genaue Vorschrift für Entwickler.

1.2.1 Gliederungsschema

- Zielbestimmung
- Produkteinsatz
- Produktfunktionen
- Produktdaten
- Produktleistungen
- Qualitätsanforderungen
- Ergänzungen
- Glossar (Begriffslexikon zur Beschreibung des Produkts)

1.3 Anforderungsermittlung

Einsicht: Man darf sich nicht auf intuitiven Eindruck darüber verlassen, was gebaut werden sollte, sondern sollte die Anforderungen systematisch ermitteln. Prinzipien:

- Erhebung der Anforderungen bei allen Gruppen und Beteiligten
- Beschreibung in einer Form, die die Beteiligten verstehen
- Validierung anhand der verschriftlichten Form
- Spezifikation: Übertragung in zur Weiterverarbeitung günstige Form
- Trennung von Belangen: Anforderung möglichst wenig koppeln
- Analyse auf Vollständigkeit: Lücken aufdecken
- Analyse auf Konsistenz: Widersprüche aufdecken und lösen
- Mediation: Widersprüche, die auf Interessengegensätze beruhen, einer Lösung zuführen
- Verwaltung: Übermäßige Anforderungsänderungen eindämmen, Anforderungsdokument immer aktuell halten

Anforderung: Eine Bedingung oder Leistungsfähigkeit, die von einem System oder einer Systemkomponente getroffen oder ausgeführt werden muss um einem Vertrag, Standard, Spezifikation, oder einem anderen formal erhobenen Dokument zu genügen.

Alle Anforderungen formen die Basis für nachfolgende Entwicklungen am System oder an der Systemkomponente.

Typen von Anforderungen:

- Funktionale Anforderungen: Beschreiben das Zusammenspiel zwischen dem System und seiner Umgebung unabhängig von der Implementierung
- Nicht funktionale Anforderungen: Sichtbare Aspekte des Systems, die nicht direkt mit funktionalem Verhalten in Bezug stehen (Performance, Qualitätsmerkmale)
- Sicherheitsanforderungen: Eine Art nicht funktionale Anforderung: Verhalten, das das System nie annehmen sollte
- Nebenbedingungen ("Pseudo Anforderungen"): Bestimmt vom Client oder der Umgebung, in der sich das System befindet.

System: Ein Teil der Realität, der überwacht wird mit seiner Umgebung zu interagieren.

- abgetrennt von seiner Umgebung durch eine Abgrenzung
- erhält Eingaben von seiner Umgebung & sendet Ausgaben an seine Umgebung
- üblicherweise hat es Untersysteme

Ein geschlossenes System interagiert nicht mit seiner Umgebung.

2 Die Definitionsphase

2.1 Aufbau eines Pflichtenheftes

2.1.1 Gliederungsschema

1. Zielbestimmung: Musskriterien, Wunschkriterien, Abgrenzungskriterien
2. Produkteinsatz: Anwendungsbereiche, Zielgruppen, Betriebsbedingungen
3. Produkt-Umgebung: Software, Hardware, Orgware, Produkt-Schnittstellen
4. Produktfunktionen: Funktion 1, Funktion 2, usw.
5. Produktdaten: Daten 1, Daten 2, usw.
6. Produktleistungen (Antwortzeit, Durchsatz, Genauigkeit)
7. Benutzungsschnittstelle
8. Qualitätsanforderungen
9. Nichtfunktionale Anforderungen (Gesetze, Normen, Sicherheit, Plattformabhängigkeiten)
10. Globale Testszenarien / Testfälle: Testfall 1, Testfall 2, usw.
11. Entwicklungs-Umgebung: Software, Hardware, Orgware, Entwicklungs-Schnittstellen
12. Ergänzungen
13. Anhang, Glossar oder Begriffslexikon

2.2 Objektorientierte Softwareentwicklung

Objekt Ein individuell erkennbares, von anderen Objekten eindeutig unterscheidbares Element der Realität (auch **Entität**).

Klasse Die Menge aller Objekte eines bestimmten Typs.

Instanz Ein konkretes Element aus dieser Menge (oder auch Ausprägung).

Attribut Gemein vorhandene Eigenschaft aller Instanzen einer Klasse, die

- für jede einzelne Instanz unabhängig von den anderen angegeben werden kann und
- einen klar definierten Wert aus einer bestimmten, für alle gleichen Domäne hat.

Notation: Attributname: Typ [= Wert]

Objektidentität Die Existenz eines Objektes ist unabhängig von seinen Attributwerten. Zwei Objekte sind auch dann unterscheidbar, wenn sie die gleichen Attributwerte besitzen.

Gleichheit x -ter Stufe

- Gleichheit 0. Stufe: es handelt sich um ein und das selbe Objekt (identisch)
- Gleichheit 1. Stufe: es handelt sich um das selbe Objekt oder zwei verschiedene Objekte, die aber in allen Attributen identische Werte besitzen (Gleichheit 0. Stufe oder paarweise Gleichheit 0. Stufe in allen Attributen)
- Gleichheit 2. Stufe: es handelt sich um das selbe Objekt oder es handelt sich um zwei verschiedene Objekte, die aber in allen Attributen gleiche oder identische Werte besitzen (Gleichheit 1. Stufe oder paarweise Gleichheit 0. oder 1. Stufe in allen Attributen)
- Gleichheit 3. Stufe: Gleichheit 2. Stufe oder paarweise Gleichheit 0., 1. oder 2. Stufe in allen Attributen
- etc.

Zustand Solange sich ein Objekt in einem Zustand befindet reagiert es im gleichen Kontext immer gleich auf seine Umwelt. Ändert sich der Zustand, reagiert das Objekt in mindestens einem Kontext anders als zuvor. (Außensicht)

Kapselungsprinzip Der Zustand ist zwar nach außen sichtbar, er wird aber im Inneren des Objektes verwaltet (also bedarfsgemäß geändert).

Methodensignatur Besteht aus **Methodenname**, **Rückgabotyp** und **Parameterliste**
Die Attribute und Methoden eines Objektes lassen sich vor dem Zugriff durch andere Objekte schützen:

- private (-): nur Instanzen der selben Klasse
- protected (#): Instanzen der selben Klasse und aller abgeleiteten Klassen, sowie Instanzen aus dem gleichen Paket
- public (+): jede Instanz

Assoziation eine Menge von Tupel (k_1, \dots, k_n) mit $k_1 \in \text{Klasse}_1, \dots, k_n \in \text{Klasse}_n$, wobei $n \geq 2$. Eine Instanz einer Assoziation heisst **Verknüpfung**.

Substitutionsprinzip Jede Instanz der Unterklasse hat die gleichen Eigenschaften, die eine Instanz der Oberklasse hätte.

Signaturvererbung Eine in der Oberklasse definierte und (evtl.) implementierte Methode überträgt nur ihre Signatur auf die Unterklasse.

Implementierungsvererbung Eine in der Oberklasse definierte und implementierte Methode überträgt ihre Signatur und ihre Implementierung auf die Unterklasse.

Abstrakte Methode Schnittstellendefinition ohne Implementierung.

Schnittstelle Definition einer Menge abstrakter Methoden, die Klassen, die es implementieren, anbieten müssen.

Varianz Modifikation der Typen der Parameter einer überschriebenen Methode

Kovarianz Verwendung einer Spezialisierung des Parametertyps in der überschreibenden Methode

Kontravarianz Verwendung einer Verallgemeinerung des Parametertyps in der überschreibenden Methode

Invarianz keine Modifikation des Typs

Forderung: Transaktionalität, **ACID**-Prinzip:

- Atomicity: Unteilbarkeit der Änderung
- Consistency: Änderungsanfragen hinterlassen einen konsistenten Zustand (Zielzustand oder im Problemfall auch den Ausgangszustand)
- Isolation: Änderungen in nebenläufigen Programmen laufen ohne Beeinflussung durch andere Programmfäden ab
- Durability: Änderung nach Abschluss für alle dauerhaft zu sehen

2.3 UML Diagramme

Anwendungsfalldiagramm, use case Zur Anforderungsspezifikation; Modellieren typischer Interaktionen des Benutzers mit dem System; ermöglicht Kontrolle, ob das System das vom Auftraggeber gewünschte leistet (Design und Implementierung)

Aktivitätsdiagramm Beschreibt eine Operation, einen Anwendungsfall, das Zusammenspiel mehrerer Anwendungsfälle, einen Geschäftsprozess. Vorteil: Beschreibung von Arbeitsabläufen unabhängig von einem konkreten Objektentwurf/Implementierung. Nebenläufigkeit und Abhängigkeiten nebenläufiger Tätigkeiten sehr leicht auszudrücken.

Interaktionsdiagramme Zeigen die für einen bestimmten Zweck notwendigen Interaktionen zwischen Objekten. Zwei Typen: Sequenzdiagramm, Kollaborationsdiagramm

Zustandsdiagramm Beschreibt mögliche Zustände eines Objektes sowie mögliche Zustandsübergänge.

3 Die Entwurfsphase

3.1 Ziele und Aufgaben des Entwurfs

Ziel Für das zu entwerfende Produkt eine Softwarearchitektur erstellen, die die funktionalen und nichtfunktionalen Produktanforderungen sowie allgemeine und produktspezifische Qualitätsanforderungen erfüllt und die Schnittstellen zur Umgebung versorgt.

Softwarearchitektur Beschreibt die Struktur des Softwaresystems durch Systemkomponenten und ihre Beziehungen untereinander.

Strukturierungsformen

- Schichten mit linearer Ordnung
- Schichten mit strikter Ordnung
- Schichten mit baumartiger Ordnung

Schichtenarchitektur + Übersichtliche Strukturierung in Abstraktionsebenen oder virtuellen Maschinen

- + Keine zu starke Einschränkung des Entwerfers, da er neben einer strengen Hierarchie noch eine liberale Strukturierungsmöglichkeit innerhalb der Schichten besitzt
- + Es werden die Wiederverwendbarkeit, die Änderbarkeit, die Wartbarkeit, die Portabilität und die Testbarkeit unterstützt
- Effizienzverlust, da alle Daten über verschiedene Schichten transportiert werden müssen
- Eindeutig voneinander abgrenzbare Abstraktionsschichten lassen sich nicht immer definieren
- Innerhalb einer Schicht kann Chaos herrschen

3.2 Modularer Entwurf

Die Hauptaufgabe des Entwurfs ist es, aus dem Pflichtenheft die Systemarchitektur zu entwickeln.

Modulführer (Grobentwurf, module guide, software architecture): Beschreibung der Funktion jedes Moduls und der Gliederung in Subsysteme; benutzt Entwurfsmuster, z.B. Schichtenarchitektur oder Fließbandarchitektur.

Modulschnittstellen Genaue Beschreibung der von jedem Modul zur Verfügung gestellten Komponenten (Typen, Variablen, Unterprogramme etc.), informell oder formal. Für Module mit Ein-/Ausgabe auch genaue Beschreibung der entsprechenden Formate.

Benutzrelation Beschreibt, wie sich Module und Subsysteme untereinander benutzen (azyklischer, gerichteter Graph).

Feinentwurf Beschreibung der modul-internen Datenstrukturen und Algorithmen; bei Implementierung in Assembler auch vollständige Programmierung der Module in Pseudocode. Der Pseudocode ist in einer höheren, meist hypothetischen Programmiersprache abgefasst und wird in der Implementierungsphase von Hand in Assembler umgesetzt.

Externer Entwurf Grobentwurf und Modulschnittstellen

Interner Entwurf Benutzrelation und Feinentwurf

3.2.1 Anforderungen an das Modulkonzept

Module sollen unabhängig voneinander bearbeitet und benutzt werden können.

1. Ein Modul sollte ohne Kenntnis der späteren Nutzung entworfen, implementiert, getestet und überarbeitet werden können;

2. Die Implementierung eines Moduls sollte möglich sein, ohne etwas über Implementierungsdetails anderer Module wissen zu müssen und ohne das Verhalten anderer Module zu beeinflussen;
3. Ein Modul sollte ohne Kenntnis seines inneren Aufbaus benutzt werden können.

3.2.2 Das Modul

Modul Ein Modul ist eine Menge von Programmkomponenten, die nach dem Geheimnisprinzip gemeinsam entworfen und geändert werden.

Geheimnisprinzip Jedes Modul verbirgt eine wichtige Entwurfsentscheidung hinter einer wohldefinierten Schnittstelle, die sich bei einer Änderung der Entscheidung nicht mitändert.

Kandidaten für Vererbung Implementierung von Datenstrukturen und deren Operationen, maschinennahe Details (z.B. Gerätetreiber, Steuerung von E/A, ...), betriebssystemnahe Details (Ein-/Ausgabeschnittstellen, Dateiformate, ...), Grundsoftware (wie Datenbanken), Ein-/Ausgabeformate, Benutzungsschnittstellen, ...

3.2.3 Allgemeine Methoden der Zerlegung

Unsere Fähigkeiten, umfangreiche und logisch komplexe Zusammenhänge einer Problemstellung zu erfassen, sind begrenzt. Daher ist eine Zerlegung der Aufgaben- und Problemstellung erforderlich, so dass die Bestandteile nacheinander bearbeitet werden können.

Übliche Vorgehensweisen:

- **Schrittweise Verfeinerung (top-down):** Bei der schrittweisen Verfeinerung gliedert man die Problem- bzw. Aufgabenstellung in Teilprobleme auf. Man verfeinert die Teilproblemstellung solange, bis man auf dem Niveau einer abstrakten Maschine angekommen ist, die zur Lösung der Blätter der Problemstellungshierarchie eingesetzt werden kann.

1. Verfeinerung anhand der Operationen von abstrakten Maschinen.
2. Verfeinerung anhand der Datenstrukturen

Probleme: Duplizieren von Teilproblemen; Effizienz; Sackgassen bei der Zerlegung: Gefahr, durch falsche Zerlegung die Problemstellung zu verfehlen.

- **Schrittweiser Aufbau (bottom-up):** Beim schrittweisen Aufbau werden die Operationen der vorhandenen abstrakten Maschine solange zusammengesetzt, bis sich eine Lösung für die gesamte Problemstellung ergibt.

1. Aufbau anhand von abstrakten Maschinen: Die Operationen der unterliegenden Maschinen werden zusammengesetzt, um komplexere Operationen einer höherliegenden Maschine zu bilden.
2. Die Daten der vorhandenen Maschine werden kombiniert.

- **Entwurf von der Mitte her (middle-out, yo-yo)**

Benutzrelation Programmkomponente A benutzt Programmkomponente B genau dann, wenn A für den korrekten Ablauf die Verfügbarkeit einer korrekten Implementierung von B erfordert. Wenn die Benutzrelation zyklensfrei ist, heisst sie **Benutzthierarchie**.

abstrakte Maschine Eine abstrakte Maschine oder virtuelle Maschine ist eine Menge von Softwarebefehlen und -objekten, die auf einer darunterliegenden (abstrakten oder realen) Maschine aufbauen und diese ganz oder teilweise verdecken können.

Produktfamilie Eine Produktfamilie oder Software-Produktlinie ist eine Menge von Programmen, die soviel gemeinsam haben, dass es sich lohnt, die Gemeinsamkeiten vor den Unterschieden zu betrachten.

3.3 Entwurfsmuster

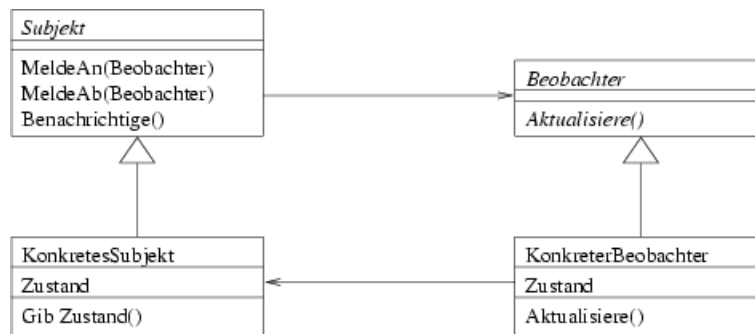
- Muster verbessern Kommunikation im Team: Entwurfsmuster bilden nützliche Terminologie, d.h. bieten Begriffe und Kurzformeln für die Diskussion zwischen Entwicklern und komplexe Konzepte
- Muster erfassen wesentliche Konzepte und bringen sie in eine verständliche Form: Muster helfen Entwürfe zu verstehen; dokumentieren Entwürfe kurz und knapp; verhindern unerwünschte Architektur-Drift; verdeutlichen Entwurfswissen
- Muster dokumentieren und fördern den Stand der Kunst: Muster helfen weniger erfahrenen Entwerfern; Muster vermeiden die Neuerfindung des Rades; ein Muster ist keine feste Regel, der man blind folgt, sondern ein Vorschlag und ein Satz von Alternativen zur Lösung eines Problems. Anpassung erforderlich
- Muster können Code-Qualität und -struktur verbessern: Muster fördern gute Entwürfe und guten Code durch Angabe konstruktiver Beispiele

3.3.1 Beobachter

Zweck Definiert eine 1-zu- n Abhängigkeit zwischen Objekten, so dass die Änderung eines Zustandes eines Objektes dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

Motivation Teilt man ein System in eine Menge von interagierenden Klassen auf, so muss die Konsistenz zwischen den miteinander in Beziehung stehenden Objekten aufrechterhalten werden. Eine enge Kopplung dieser Klassen ist nicht empfehlenswert, weil dies die individuelle Wiederverwendbarkeit einschränkt. Im MVC-Beispiel wissen die Tabellendarstellung und die Säulendarstellung nichts voneinander. Damit können sie unabhängig voneinander wiederverwendet werden. Trotzdem verhalten sich beide Objekte so, als ob sie einander kennen würden.

Struktur



Anwendbarkeit

- Wenn die Änderung eines Objekts die Änderung anderer Objekte verlangt und man nicht weiß, wie viele und welche Objekte geändert werden müssen.
- Wenn ein Objekt andere Objekte benachrichtigen muss, ohne Annahmen über diese Objekte zu treffen.
- Wenn eine Abstraktion zwei Aspekte besitzt, wobei einer von dem anderen abhängt. Die Kapselung dieser Aspekte in separaten Objekten unabhängige Wiederverwendbarkeit.

Konsequenzen

- Subjekte und Beobachter können unabhängig voneinander wiederverwendet werden.
- Beobachter können neu hinzugefügt oder entfernt werden, ohne das Subjekt oder andere Beobachter zu ändern.

- Die abstrakte Kopplung zwischen Subjekt und Beobachter wird durch die Benachrichtigung erreicht. Subjekt und Beobachter gehören verschiedenen Schichten der Benutz-Hierarchie an, ohne dabei Zyklen zu erzeugen. (Subjekt benutzt nicht die Beobachter, aber umgekehrt.)
- Automatischer Rundruf von Änderungen.
- Beobachter entscheiden selbst, ob sie die Benachrichtigung ignorieren oder nicht.
- Der Aufwand der Aktualisierung kann versteckt sein.
Eine einfache Benachrichtigung kann zu einer Kaskade von Aktualisierungen bei den Beobachtern führen.
Die Botschaft enthält keinen Hinweis was geändert wurde. Ein erweitertes Protokoll kann verwendet werden, um den Beobachtern die konkrete Änderung mitzuteilen.

Implementierung

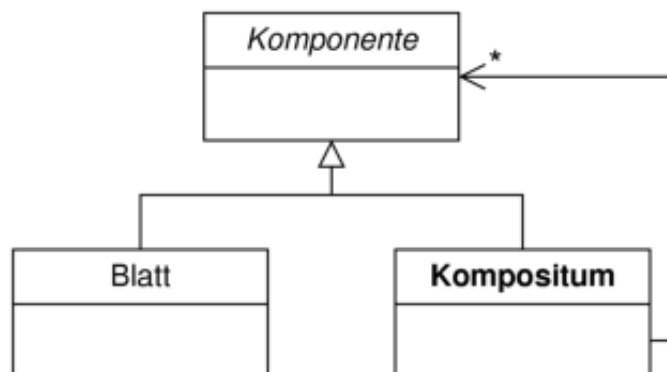
1. Wenn mehr als ein Subjekt beobachtet wird: Verwende Subjekt als Parameter von `aktualisiere(s : Subjekt)`
2. Auflösen der Aktualisierung:
 - `setzeZustand()` ruft `benachrichtige()`,
 - oder Klienten rufen `benachrichtige()` explizit.
3. Löschen eines Beobachters: Als erstes beim entsprechenden Subjekt abmelden.
4. Subjekte müssen vor der Benachrichtigung konsistent sein. Wenn eine Subjekt-Unterklasse geerbte Operationen aufruft, kann versehentlich die Oberklasse eine Benachrichtigung auslösen bevor das Unterklassenobjekt komplett konsistent ist. Alternative: Schablonenmethode für die Aktualisierung verwenden.
5. Aktualisierung: Pull- und Push-Modell
 - Pull-Modell: Beobachter holt alle Daten direkt vom Subjekt (kann ineffizient sein)
 - Push-Modell: Subjekt schickt Änderungsdaten an die Beobachter in `aktualisiere()` (kann Wiederverwendbarkeit reduzieren)
6. ÄnderungsManager zwischen Subjekt und Beobachtern:
 - bildet Subjekt auf seine Beobachter ab und bietet eine Schnittstelle zur Verwendung dieser Abbildung
 - aktualisiert bei Anforderung durch das Subjekt alle abhängigen Beobachter
 - vermeidet mehrfache Aktualisierungen
 - kapselt komplexe Aktualisierungssemantik

3.3.2 Kompositum

Zweck Füge Objekte zu Baumstrukturen zusammen, um Bestands-Hierarchien zu repräsentieren. Das Muster ermöglicht es Klienten, sowohl einzelne Objekte als auch Aggregate einheitlich zu behandeln.

Motivation Bestands-Hierarchien treten überall dort auf, wo komplexe Objekte modelliert werden, wie beispielsweise Datei-Systeme, graphische Anwendungen, Textverarbeitung, CAD, CIM, ... Bei diesen Anwendungen werden einfache Objekte zu Gruppen zusammengefasst, welche wiederum zu größeren Gruppen zusammengefügt werden können. Häufig soll dabei die Behandlung von Objekten und Aggregaten durch das Programm einheitlich sein. Das Kompositum isoliert die gemeinsamen Eigenschaften von Objekt und Aggregat und bildet daraus eine Oberklasse.

Struktur



Anwendbarkeit

- Wenn Bestands-Hierarchien von Objekten repräsentiert werden sollen.
- Wenn die Klienten in der Lage sein sollen, die Unterschiede zwischen zusammengesetzten und einzelnen Objekten zu ignorieren.

Implementierung

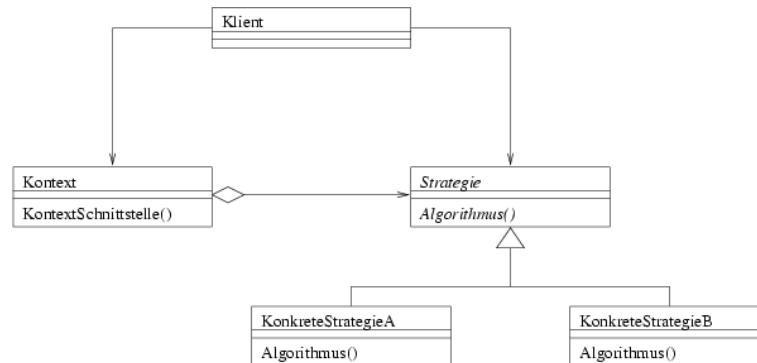
1. Es ist häufig nützlich eine Eltern-Referenz in jeder Komponente zu führen (erleichtert Traversierung). Diese Referenz kann von den `fügeHinzu` und `entferne` Methoden des Kompositums gepflegt werden.
Das Teilen von Komponenten kann zu mehreren Eltern und damit zu Zweideutigkeiten führen.
2. Maximieren der Komponenten-Schnittstelle
Die Komponenten-Schnittstelle sollte so viele gemeinsame Methoden des Kompositums und der Blätter wie möglich definieren um Transparenz zu garantieren.
Wenn Methoden des Kompositums in der Komponente definiert werden, sollte `gibKindobjekt` bei Blättern nichts zurück- geben - das kann auch durch eine entsprechende Implementierung in der Komponente erreicht werden.
`fuegeHinzu` und `entferne` sollten bei Blättern fehlschlagen (und einen Fehler zurückgeben oder eine Ausnahme generieren).
3. Speichern der Kinder:
Felder, Listen oder Hash-Tabellen

3.3.3 Strategie

Zweck Definiere eine Familie von Algorithmen, kapsle sie und mache sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von nutzenden Klienten zu variieren.

Motivation Manchmal müssen Algorithmen, abhängig von der notwendigen Performance, der Anzahl oder des Typs der Daten, variiert werden.

Struktur



Anwendbarkeit

- Wenn sich viele verwandte Klassen nur in ihrem Verhalten unterscheiden. Strategieobjekte bieten die Möglichkeit, eine Klasse mit einer von mehreren möglichen Verhaltensweisen zu konfigurieren.
- Wenn unterschiedliche Varianten eines Algorithmus benötigt werden.
- Wenn ein Algorithmus Datenstrukturen verwendet, die Klienten nicht bekannt sein sollen.
- Wenn eine Klasse unterschiedliche Verhaltensweisen definiert und diese als mehrfache Bedingungsanweisungen in ihren Operationen erscheinen.
- Alternativ zur Ableitung der Klasse Strategie kann man auch die Klasse Kontext ableiten, um verschiedene Verhaltensmuster zu implementieren. Das Ergebnis sind viele Klassen, die sich nur im Verhalten unterscheiden, welches für jede Klasse fest ist. Das Strategie-Muster erlaubt demgegenüber auch eine dynamische Veränderung des Verhaltens.

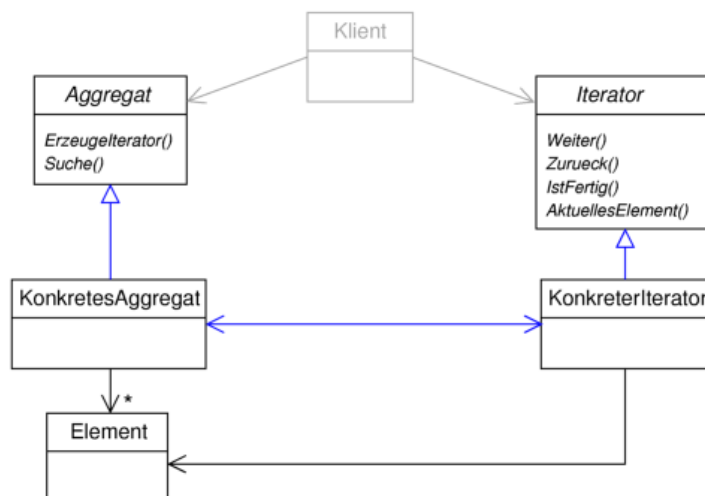
3.4 Entkoppelungs-Muster

3.4.1 Datenablage (Repository)

Zweck Eine Menge unabhängiger Komponenten kommunizieren über eine zentrale Ablage, in dem sie Elemente in dieser Datenstruktur ablegen oder aus ihr herausholen.

3.4.2 Iterator

Zweck Ermöglicht den sequentiellen Zugriff auf die Elemente eines zusammengesetzten Objekts, ohne seine zugrundeliegende Repräsentation offenzulegen.

Struktur**Anwendbarkeit**

- Um den Zugriff auf den Inhalt eines zusammengesetzten Objekts zu ermöglichen, ohne dabei seine interne Struktur offenzulegen.
- Um mehrfache gleichzeitige Traversierungen auf zusammengesetzten Objekten zu ermöglichen.
- Um eine einheitliche Schnittstelle zur Traversierung unterschiedlicher zusammengesetzter Strukturen anzubieten (d.h. um polymorphe Iteration zu ermöglichen).

3.4.3 Schichtenarchitektur

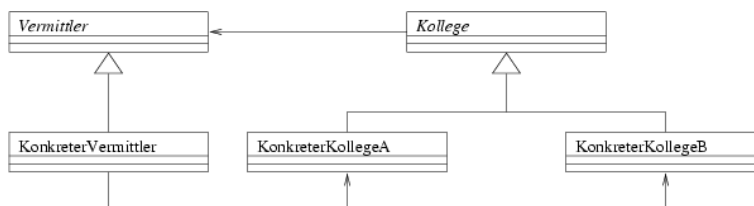
Zweck Gliedere ein System in eine hierarchisch geordnete Menge von Schichten. Eine Schicht besteht aus einer Menge von Software-Komponenten mit einer wohldefinierten Schnittstelle, nutzt die darunterliegenden Schichten als Klient und stellt seine Dienste an darüberliegende Schichten zur Verfügung.

Anwendbarkeit

- Unabhängige Entwicklung und Korrektur, Austausch von Schichten.
- Schrittweiser Aufbau und schrittweises Testen.
- Wiederverwendung von tieferen Schichten in anderen Konfigurationen.

3.4.4 Vermittler (Mediator)

Zweck Definiere ein Objekt, welches das Zusammenspiel einer Menge von Objekten in sich kapselt. Vermittler fördern lose Kopplung, indem sie Objekte davon abhalten, aufeinander explizit Bezug zu nehmen. Sie ermöglichen es, das Zusammenspiel der Objekte unabhängig zu variieren.

Struktur

Beispiele Abhängigkeiten zwischen Elementen einer Dialogbox; unterschiedliche Dialogboxen besitzen unterschiedliche Abhängigkeiten zwischen Elementen

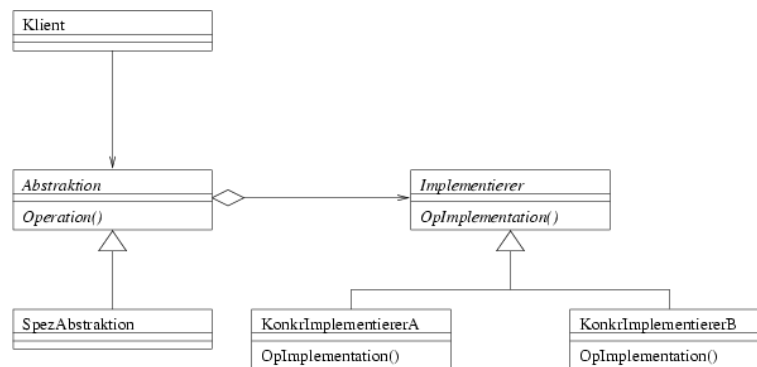
Anwendbarkeit

- Wenn eine Menge von Objekten vorliegt, die in wohldefinierter, aber komplexer Weise miteinander zusammenarbeiten. Die sich ergebenden Abhängigkeiten sind unstrukturiert und schwer zu verstehen.
- Wenn die Wiederverwertung eines Objektes schwierig ist, weil es sich auf viele andere Objekte bezieht und mit ihnen zusammenarbeitet.
- Wenn ein auf mehrere Klassen verteiltes Verhalten maßgeschneidert werden soll, ohne viele Unterklassen bilden zu müssen.

3.4.5 Brücke (Bridge)

Zweck Entkopple eine Abstraktion von ihrer Implementierung, so dass beide unabhängig variiert werden können.

Struktur



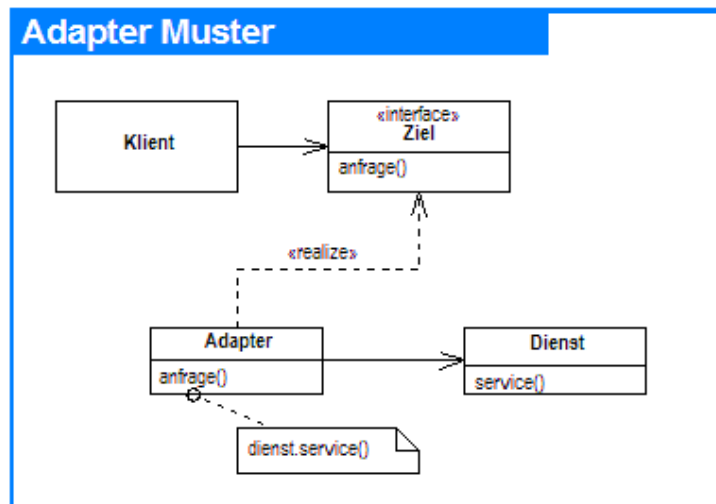
Anwendbarkeit

- Wenn eine dauerhafte Verbindung zwischen Abstraktion und Implementierung vermieden werden soll.
- Wenn sowohl Abstraktion als auch Implementierungen durch Unterklassenbildung erweiterbar sein soll.
- Wenn Änderungen in der Implementierung einer Abstraktion keine Auswirkung auf Klienten haben sollen.
- Wenn die Implementierung einer Abstraktion vollständig vom Klienten versteckt werden soll.
- Wenn eine starke Vergrößerung der Anzahl der Klassen vermieden werden soll.
- Wenn eine Implementierung von mehreren Objekten aus gemeinsam benutzt werden soll.

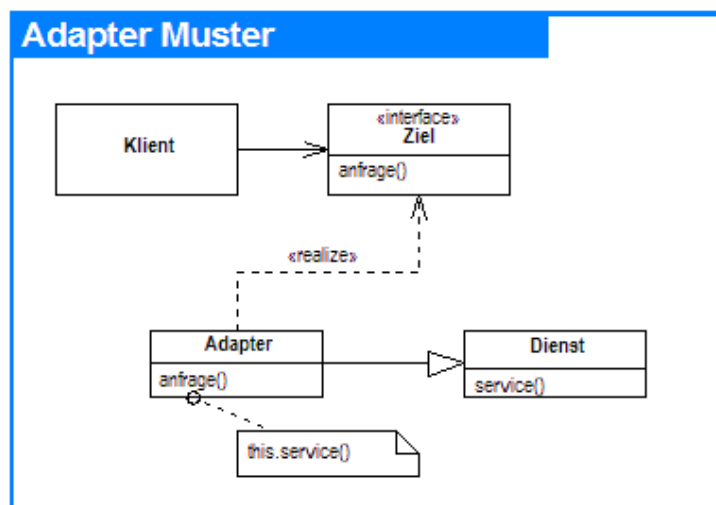
3.4.6 Adapter

Zweck Passe die Schnittstelle einer Klasse an eine andere von ihren Klienten erwartete Schnittstelle an. Das Adaptermuster läßt Klassen zusammenarbeiten, die wegen inkompatibler Schnittstellen ansonsten dazu nicht in der Lage wären.

Struktur



mit Mehrfach-Vererbung:



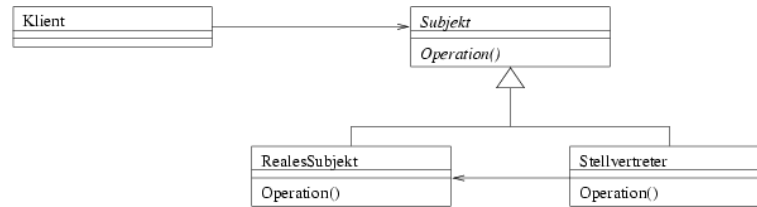
Anwendbarkeit

- Wenn eine existierende Klasse verwendet werden soll, deren Schnittstelle aber nicht der benötigten Schnittstelle entspricht.
- Wenn eine wieder verwendbare Klasse erstellt werden soll, die mit unabhängigen oder nicht vorhersehbaren Klassen zusammenarbeitet, d.h. Klassen die nicht notwendigerweise kompatible Schnittstellen besitzen.
- Wenn verschiedene existierende Unterklassen benutzt werden sollen, es aber unpraktisch ist, die Schnittstellen jeder einzelnen Unterklasse durch Ableiten anzupassen. Ein Objektadapter ist in der Lage, die Schnittstelle seiner Oberklasse anzupassen.

3.4.7 Stellvertreter (Proxy)

Zweck Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.

Struktur



Anwendbarkeit Das Stellvertretermuster ist anwendbar, sobald es den Bedarf nach einer anpassungsfähigeren und intelligenteren Referenz auf ein Objekt als einen einfachen Zeiger gibt. Es folgen einige verbreitete Situationen, in denen das Stellvertretermuster anwendbar ist:

1. Ein **protokollierender Stellvertreter** zählt Referenzen auf das eigentliche Objekt, so dass es automatisch freigegeben werden kann, wenn keine Referenzen mehr auf das Objekt existieren. Er kann auch andere Zugriffsinformationen protokollieren und leitet Zugriffe weiter.
2. Ein **puffernder Stellvertreter** lädt ein persistentes Objekt erst dann in den Speicher, wenn es das erste Mal dereferenziert wird. Er kann auch einen Puffer mit mehreren Objekten verwalten, die nach Bedarf zwischen Hintergrund- und Hauptspeicher bewegt werden.
3. Ein **Fernzugriffsvertreter** stellt einen lokalen Stellvertreter für ein Objekt in einem anderen Adressraum dar.
4. Ein **Platzhalter** erzeugt teure Objekte auf Verlangen (verzögertes Laden, verzögertes Erzeugen).
5. Eine **Schutzwand** kontrolliert den Zugriff auf das Originalobjekt. Schutzwände sind nützlich, wenn Objekte über verschiedene Zugriffsrechte verfügen sollen.
6. Ein **Dekorierer** fügt zusätzliche Zuständigkeiten zu einem bestehenden Objekt hinzu (möglicherweise kaskadiert).

3.4.8 Fließband (Pipes and Filters)

Zweck Bietet eine Struktur für Systeme, die Datenströme bearbeiten. Jeder Bearbeitungsschritt ist in einer Filter Komponente gekapselt. Daten werden über Kanäle von einem Filter zu einem anderen weitergegeben. Eine Neukombination von Filtern ermöglicht es, Familien von Systemen zu erstellen.

Beispiel Erzeugen eines Reimwörterbuchs (Wörter sind von hinten her sortiert, d.h. Wörter, die gleich enden, stehen hintereinander)

Anwendbarkeit

- Wenn ein System Datenströme bearbeiten oder transformieren muss und ein System bestehend aus nur einer Komponente unhandhabbar ist.
- Wenn in zukünftigen Entwicklungen einzelne Komponenten ersetzt oder Arbeitsschritte umgeordnet werden sollen.
- Wenn kleinere Komponenten einfacher in anderen Zusammenhängen wiederverwendet werden können.
- Wenn einzelne Komponenten parallel oder quasi-parallel ablaufen sollen.
- Nachteil: nicht geeignet für interaktive Systeme

3.4.9 Ereigniskanal (Event Channel)

Zweck Entkopple Teilnehmer an einem Gesamtsystem vollständig voneinander, so dass sie völlig eigenständig arbeiten können und über die Existenz oder Anzahl anderer Teilnehmer nichts wissen. Interaktionen erfolgen über Ereignisse.

3.4.10 Rahmenarchitektur (Framework)

Zweck Bietet ein (nahezu) vollständiges Programm, das durch Einfüllen geplanter "Lücken" erweitert werden kann. Es enthält die vollständige Anwendungslogik, meistens sogar ein komplettes Hauptprogramm. Von einigen der Klassen in dem Programm können Benutzer Unterklassen bilden und dabei Methoden überschreiben oder vordefinierte abstrakte Methoden implementieren. Das Rahmenprogramm sieht vor, dass die vom Benutzer gelieferten Erweiterungen richtig aufgerufen werden.

Anwendbarkeit

- Wenn eine Grundversion der Anwendung schon funktionsfähig sein soll.
- Wenn Erweiterungen möglich sein sollen, die sich konsistent verhalten (Anwendungslogik im Rahmenprogramm).
- Wenn komplexe Anwendungslogik nicht neu programmiert werden soll.

Die Muster Fabrikmethode, abstrakte Fabrik und Schablonenmethode werden häufig in Rahmenprogrammen benötigt.

3.5 Varianten-Muster

3.5.1 Oberklasse

Zweck Einheitliche Behandlung von Objekten, die unterschiedlichen Klassen angehören, aber gemeinsame Attribute oder Methoden besitzen.

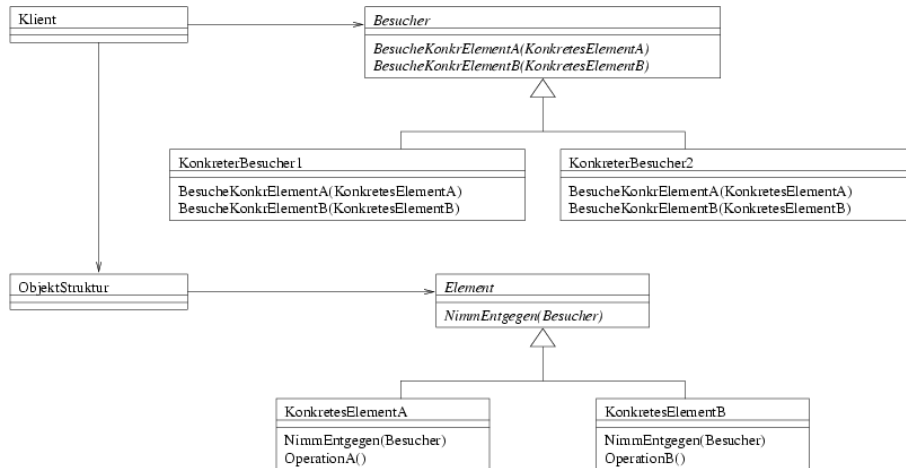
Anwendbarkeit

- Wenn Objekte verschiedener Klassen gemeinsame Attribute, Methoden oder Schnittstellen haben.
- Wenn Objekte verschiedener Klassen einheitlich in einem Programm behandelt werden sollen.
- Wenn es möglich sein soll, weitere Klassen hinzuzufügen, ohne den bestehenden Quelltext zu ändern.

3.5.2 Besucher (Visitor)

Zweck Kapsle eine auf den Elementen einer Objektstruktur auszuführende Operation als ein Objekt. Das Besuchermuster ermöglicht es, eine neue Operation zu definieren, ohne die Klassen der von ihr bearbeiteten Elemente zu verändern.

Struktur



Anwendbarkeit

- Wenn eine Objektstruktur viele Klassen von Objekten mit unterschiedlichen Schnittstellen enthält und Operationen auf diesen Objekten ausgeführt werden sollen, die von ihren konkreten Klassen abhängen.
- Wenn viele unterschiedliche und nicht miteinander verwandte Operationen auf den Objekten einer Objektstruktur ausgeführt werden müssen und diese Klassen nicht mit diesen Operationen "verschmutzt" werden sollen.
- Wenn sich die Klassen, die eine Objektstruktur definieren, praktisch nie ändern, aber häufig neue Operationen für die Struktur definiert werden.

3.5.3 Schablonenmethode

Zweck Definiere das Skelett eines Algorithmus in einer Operation und delegiere einzelne Schritte an Unterklassen. Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern.

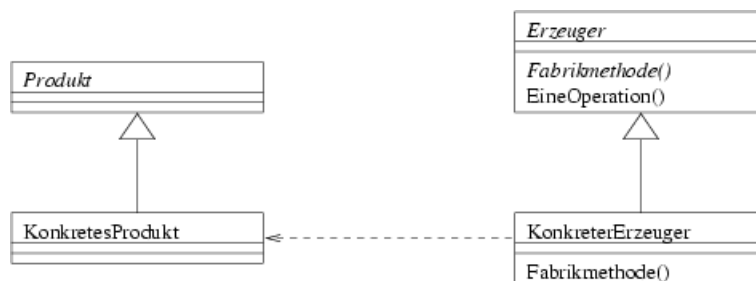
Anwendbarkeit

- Um die invarianten Teile eines Algorithmus genau einmal festzulegen und es dann Unterklassen zu überlassen, das variierende Verhalten zu implementieren.
- Wenn gemeinsames Verhalten aus Unterklassen herausfaktoriert und in einer allgemeinen Klasse platziert werden soll, um die Verdopplung von Code zu vermeiden.
- Um die Erweiterungen durch Unterklassen zu kontrollieren. Eine Schablonenmethode lässt sich so definieren, dass sie "Einschubmethoden" (hooks) an bestimmten Stellen aufruft und damit Erweiterungen nur an diesen Stellen zulässt.

3.5.4 Fabrikmethode

Zweck Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.

Struktur



Anwendbarkeit

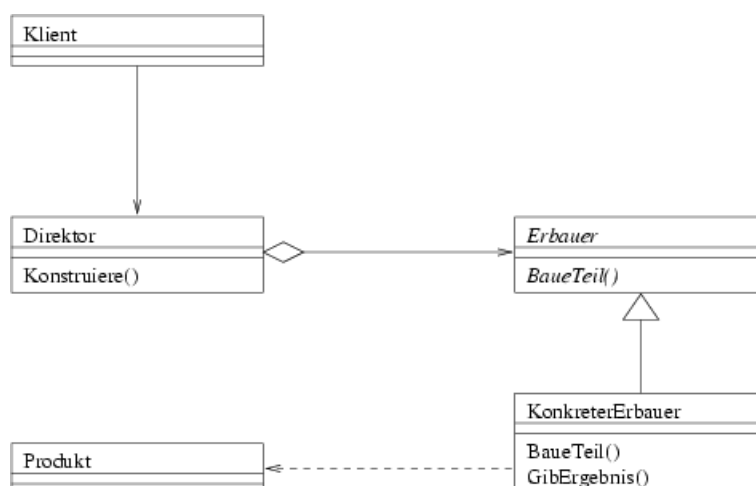
- Wenn eine Klasse die Klasse von Objekten, die sie erzeugen muss, nicht im voraus kennen kann.
- Wenn eine Klasse möchte, dass ihre Unterklasse die von ihr zu erzeugenden Objekte festlegen.
- Wenn Klassen Zuständigkeiten an eine von mehreren Hilfsunterklassen delegieren sollen und das Wissen, an welche Hilfsunterklasse die Zuständigkeit delegiert wird, lokalisiert werden soll.

Eine Fabrikmethode ist die Einschubmethode bei einer Schablonenmethode für Objekterzeugung.

3.5.5 Erbauer

Zweck Trenne die Konstruktion eines komplexen Objekts (bestehend aus mehreren Teilen) von seiner Repräsentation, so dass derselbe Konstruktionsprozess unterschiedliche Repräsentationen erzeugen kann.

Struktur



Anwendbarkeit

- Der Algorithmus zum Erzeugen eines komplexen Objekts soll unabhängig von den Teilen sein, aus denen das Objekt besteht und wie sie zusammengesetzt werden.

- Der Konstruktionsprozess muss verschiedene Repräsentationen des zu konstruierenden Objekts erlauben.

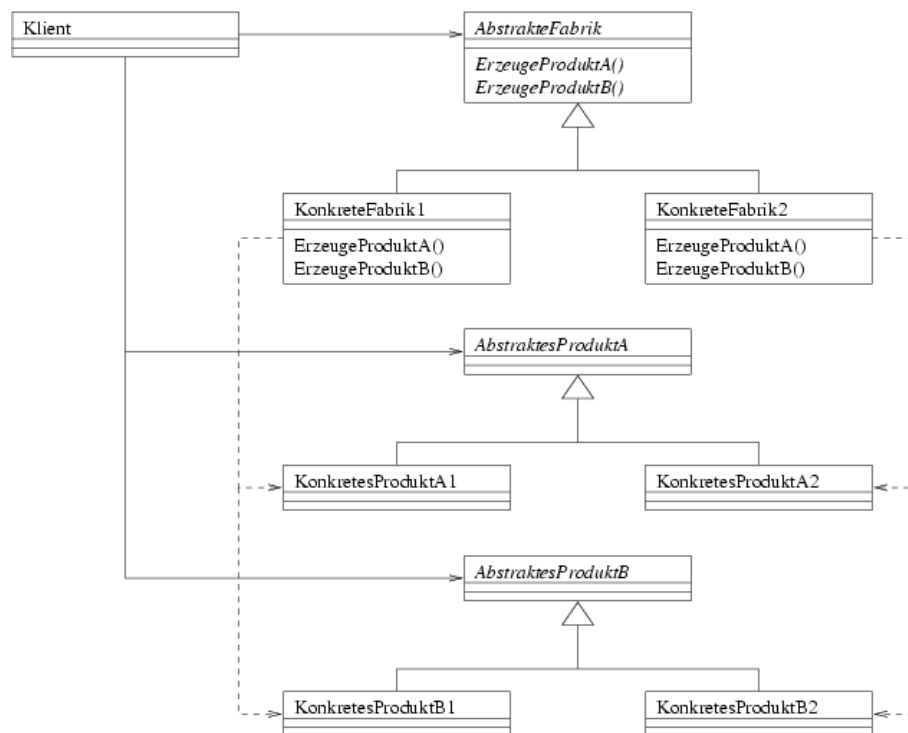
Interaktionen

- Der Klient erzeugt das Direktorobjekt und konfiguriert es mit dem gewünschten Erbauerobjekt.
- Der Direktor informiert den Erbauer, wenn ein Teil des Produkts gebaut werden soll.
- Der Erbauer bearbeitet die Anfragen des Direktors und fügt Teile zum Produkt hinzu.
- Der Klient erhält das Produkt vom Erbauer.

3.5.6 Abstrakte Fabrik

Zweck Bietet eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander abhängiger Objekte, ohne ihre konkreten Klassen zu benennen.

Struktur



Anwendbarkeit

- Wenn ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und repräsentiert werden.
- Wenn ein System mit einer von mehreren Produktfamilien konfiguriert werden soll.
- Wenn eine Familie von verwandten Produktobjekten zusammen verwendet werden sollen, und dies erzwungen werden muss.
- Bei einer Klassenbibliothek, die nur die Schnittstellen, nicht aber die Implementierung offenlegt.

3.6 Zustandshandhabungs-Muster

3.6.1 Memento

Zweck Erfasse und externalisiere den internen Zustand eines Objekts, ohne seine Kapselung zu verletzen, so dass das Objekt später in diesen Zustand zurückversetzt werden kann.

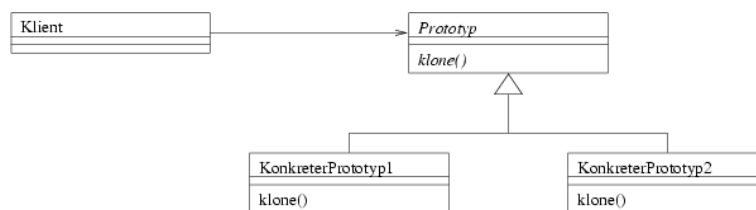
Anwendbarkeit

- Wenn eine Momentaufnahme (eines Teils) des Zustandes eines Objekts zwischengespeichert werden muss, so dass es zu einem späteren Zeitpunkt in diesen Zustand zurückversetzt werden kann, und
- wenn eine direkte Schnittstelle zu Ermitteln des Zustands die Implementierungsdetails offenlegen und die Kapselung des Objekts aufbrechen würde.

3.6.2 Prototyp

Zweck Bestimme die Arten zu erzeugender Objekte durch die Verwendung eines typischen Exemplars und erzeuge neue Objekte durch Kopieren dieses Prototyps.

Struktur



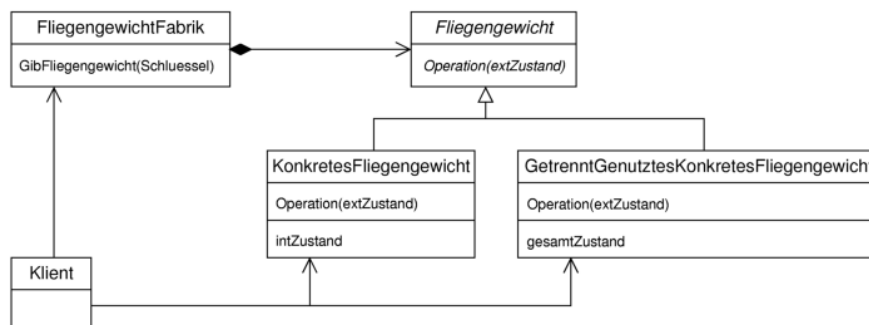
Anwendbarkeit Das Prototypmuster wird verwendet, wenn ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und repräsentiert werden, und

- wenn die Klassen zu erzeugender Objekte erst zur Laufzeit spezifiziert werden, z.B. durch dynamisches Laden, oder
- um eine Klassenhierarchie von Fabriken zu vermeiden, die parallel zur Klassenhierarchie der Produkte verläuft, oder
- wenn Exemplare einer Klasse nur wenige Zustandskombinationen haben können. Es ist möglicherweise bequemer, eine entsprechende Anzahl von Prototypen einzurichten und sie zu klonen statt die Objekte einer Klasse jedesmal von Hand mit dem richtigen Zustand zu erzeugen.

3.6.3 Fliegengewicht (Flyweight)

Zweck Nutze Objekte kleinster Granularität gemeinsam, um große Mengen von ihnen effizient speichern zu können.

Struktur



Anwendbarkeit

- Wenn die Anwendung eine große Menge von Objekten verwendet, und
- wenn Speicherkosten allein aufgrund der Anzahl von Objekten hoch sind, und

- wenn ein Großteil des Objektzustands in den Kontext verlagert und damit extrinsisch gemacht werden kann, und
- viele Gruppen von Objekten durch relativ wenige gemeinsam genutzte Objekte ersetzt werden, sobald einmal der extrinsische Zustand entfernt wurde, und
- die Anwendung nicht von der Identität der Objekte abhängt (sonst Identität trotz konzeptuellem Unterschied).

3.7 Steuerungs-Muster

3.7.1 Tafel (Blackboard)

Zweck Das Tafelmuster ist nützlich bei Problemen, für die keine deterministischen Lösungsstrategien bekannt sind. Mehrere spezialisierte Subsysteme vereinigen ihr Wissen auf der Tafel um eine eventuell partielle oder approximative Lösung zu finden.

Einfache Form Steuerung aktiviert alle Wissensquellen der Reihe nach.

Komplexere Form bestimmt eine Folge anwendbarer Wissensquellen (über `bedingung()`), wählt davon aus, und führt diese dann aus.

Anwendbarkeit

- Wenn mehrere Transformationen ("Wissensquellen") auf einer gemeinsamen Datenstruktur ("Tafel") operieren.
- Wenn das Auslösen der Transformationen vom Inhalt der Datenstruktur gesteuert wird.
- Wenn die Auswahl der anwendbaren Transformationen gesteuert werden soll.

3.7.2 Befehl (Command)

Zweck Kapselt einen Befehl als ein Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine Warteschlange zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen.

Anwendbarkeit

- Wenn Objekte mit einer auszuführenden Aktion parametrisiert werden sollen.
- Wenn Anfragen zu unterschiedlichen Zeiten spezifiziert, aufgereiht und ausgeführt werden sollen.
- Wenn ein Rückgängigmachen von Operationen (Undo) unterstützt werden soll.
- Wenn das Mitprotokollieren von Änderungen unterstützt werden soll (um System nach Absturz wiederherzustellen).
- Wenn ein System mittels komplexer Operationen strukturiert werden soll, die aus primitiven Operationen aufgebaut werden (Makrobefehle).

3.7.3 Zuständigkeitskette (Chain of Responsibility)

Zweck Vermeide die Kopplung des Auslösers einer Anfrage mit seinem Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Anfrage zu erledigen. Verkette die empfangenden Objekte und leite die Anfrage an der Kette entlang, bis ein Objekt sie erledigt.

Anwendbarkeit

- Wenn mehr als ein Objekt eine Anfrage bearbeiten können soll und dasjenige Objekt, das dies tut, nicht von vornherein bekannt ist. Dieses Objekt muss zur Laufzeit automatisch bestimmt werden.
- Wenn eine Anfrage an eines von mehreren Objekten gerichtet werden soll, ohne den Empfänger explizit anzugeben.
- Wenn eine Menge Objekte, welche eine Anfrage bearbeiten sollen, dynamisch festgelegt wird.

3.7.4 Master/Slave

Zweck Das Master/Slave-Muster unterstützt Fehlertoleranz und parallele Berechnung. Eine Master Komponente (Auftraggeber) verteilt die Arbeit an identische Slave-Komponenten (Arbeiter, Auftragnehmer) und berechnet das Endergebnis aus den Teilergebnissen, welche die Arbeiter liefern.

Anwendbarkeit

- Wenn es mehrere Aufgaben gibt, die unabhängig voneinander bearbeitet werden können.
- Wenn mehrere Prozessoren zur parallelen Verarbeitung zur Verfügung stehen.
- Wenn die Belastung der Arbeiter ausgeglichen werden soll.

3.7.5 Prozess-Steuerung

Zweck Regulierung eines physikalischen (kontinuierlichen) Prozesses.

- System ohne Rückkoppelung
- System mit Rückkoppelung (Regelung mit Rückführung, Regelung mit Störgrößenaufschaltung)

Vokabular Prozessvariablen: Eigenschaften des Prozesses, die gemessen werden können; folgende speziellen Arten werden häufig unterschieden.

Regelgröße: Prozessvariable, deren Wert das System kontrollieren möchte.

Eingangsgrößen: Prozessvariablen, die als Eingabewerte für den Prozess dienen.

Stellgröße: Prozessvariablen, deren Wert von der Steuerung verändert werden kann.

Sollwert: Der angestrebte Wert einer Regelgröße.

3.8 Virtuelle Maschinen

3.8.1 Interpretierer

Zweck Definiert für eine gegebene Sprache eine Repräsentation der Grammatik sowie einen Interpretierer, der die Repräsentation nutzt, um Sätze in der Sprache zu interpretieren.

Anwendbarkeit Wenn eine Sprache interpretiert werden muss und Ausdrücke der Sprache als abstrakte Syntaxbäume darstellbar sind. Das Interpretierermuster funktioniert am besten, wenn

- die Grammatik einfach ist. Bei komplexen Grammatiken wird die Klassenhierarchie zu groß und nicht mehr handhabbar. In diesem Falle stellen Werkzeuge wie Zerteilergeneratoren eine bessere Alternative dar.
- die Effizienz unproblematisch ist. Effiziente Interpretierer werden üblicherweise nicht durch Interpretation von Syntaxbäumen implementiert; sie transformieren die Bäume stattdessen in eine andere Form, z.B. Zwischencode.

3.8.2 Regelbasierter Interpretierer

Zweck Löse ein Problem durch Anwendung einer Menge von Regeln. Eine Regel besteht aus einem Bedingungsteil und einem Aktionsteil. Falls der Bedingungsteil, angewandt auf eine Menge von Datenelementen im Arbeitsspeicher, wahr ergibt, dann kann der Aktionsteil ausgeführt werden. Der Aktionsteil ändert, ersetzt oder löscht Datenelemente, die im Bedingungsteil ausgewählt wurden, oder fügt neue Datenelemente hinzu.

Anwendbarkeit

- Wenn eine Problemlösung am besten als eine Menge von Bedingungs-Aktions-Regeln formuliert werden kann, z.B. bei Diagnose- oder Konfigurierungsaufgaben.
- Wenn der Aktionsteil nur einfache Operationen an den Datenelementen erfordert (keine Schleifen, keine Rekursion, keine Prozeduraufrufe, um Arbeitsspeicher zu modifizieren).

3.9 Bequemlichkeits-Muster

3.9.1 Bequemlichkeits-Methode

Zweck Vereinfachen des Methodenaufrufs durch die Bereitstellung häufig genutzter Parameterkombinationen durch zusätzliche Methoden (Überladen).

Anwendbarkeit Wenn Methodenaufrufe häufig mit den gleichen Parametern auftreten.

3.9.2 Bequemlichkeits-Klasse

Zweck Vereinfache den Methodenaufruf durch Bereithaltung der Parameter in einer speziellen Klasse.

Anwendbarkeit Wenn Methoden häufig mit den gleichen Parametern aufgerufen werden, die sich nur selten ändern.

3.9.3 Fassade

Zweck Bietet eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Benutzung des Subsystems vereinfacht.

Anwendbarkeit

- Wenn eine einfache Schnittstelle zu einem komplexen Subsystem angeboten werden soll. Eine Fassade kann eine einfache voreingestellte Sicht auf das Subsystem bieten, die den meisten Klienten genügt.
- Wenn es viele Abhängigkeiten zwischen den Klienten und den Implementierungsklassen einer Abstraktion gibt. Die Einführung einer Fassade entkoppelt die Subsysteme von Klienten und anderen Subsystemen.
- Wenn Subsysteme in Schichten aufgeteilt werden sollen. Man verwendet eine Fassade, um einen Eintrittspunkt zu jeder Subsystemschicht zu definieren.

3.9.4 Null-Objekt

Zweck Stelle einen Stellvertreter zur Verfügung, der die gleiche Schnittstelle bietet, aber nichts tut. Das Null-Objekt kapselt die Implementierungs-Entscheidungen (wie genau es "nichts tut") und versteckt diese Details vor seinen Mitarbeitern.

Motivation Es wird verhindert, dass der Code mit Tests gegen Null-Werte verschmutzt wird.

Anwendbarkeit

- Wenn ein Objekt Mitarbeiter benötigt und einer oder mehrere von ihnen nichts tun sollen.
- Wenn Klienten sich nicht um den Unterschied zwischen einem echten Mitarbeiter und einem der nichts tut kümmern sollen.
- Wenn das "tue nichts"-Verhalten von verschiedenen Klienten wiederverwendet werden soll.

4 Die Implementierungsphase

4.1 Einführung und Überblick

Programmierung, Dokumentierung und Testen der Systemkomponenten aufgrund vorgegebener Spezifikationen der Systemkomponenten.

Voraussetzung

- Für jede Systemkomponente existiert eine Spezifikation.
- Die Softwarearchitektur ist so ausgelegt, dass die Implementierung umfangmäßig pro Funktion, Zugriffsoption bzw. Methode wenige Seiten nicht überschreitet.

Aktivitäten

- Konzeption von Datenstrukturen und Algorithmen
- Strukturierung des Programms durch geeignete Verfeinerungsebenen
- Dokumentation der Problemlösung und der Implementierungsentscheidungen
- Umsetzung der Konzepte in die Konstrukte der verwendete Programmiersprache
- Angaben zur Zeit- und Speicherkomplexität
- evtl. Programmieroptimierung
- Test oder Verifikation des Programms einschliesslich Testplanung und Testfallerstellung

Teilprodukte

- Quellprogramm einschliesslich integrierter Dokumentation
- Objektprogramm
- ausführbare Testfälle und Testprotokoll bzw. Verifikationsdokument

Alle Teilprodukte aller Systemkomponenten müssen später integriert und einem Systemstart unterzogen werden.

4.2 Programmoptimierung

Die Hauptaufgaben der Implementierungsphase sind die Umsetzung der Spezifikation in korrekte, ablauffähige Programme, die Dokumentation und das Testen. Programmoptimierung ist in den meisten Fällen zweitrangig. Arten der Optimierung: Laufzeitreduktion, Speicherplatzreduktion, Cache-Optimierungen

4.2.1 Laufzeitreduktion

Wenn eine kleine Beschleunigung erzielt werden soll, arbeite auf der vielversprechendsten Optimierungsebene! Wenn eine große Beschleunigung gewünscht ist, beachte alle Optimierungsebenen!

Problemstellung Vereinfache!

Vermeidung übermäßiger Komplexität (schleichende Funktionsanhäufung; Vergoldung; "Effekt des zweiten System")

Systemstruktur Benutze Überschlagsrechnungen, um die Leistung eines geplanten Systems abzuschätzen!

Beantwortung folgender Fragen: Genügt ein geplantes System den Effizienzanforderungen? Welche Systemstruktur ist die beste?

Algorithmen und Datenstrukturen

- Speicherung von Zwischenergebnissen statt Neuberechnung
- Vorverarbeitung von Daten

- Teile und Herrsche
- Dynamisches Programmieren

Feinoptimierung (Code Tuning)

- Ausnutzung eines häufig auftretenden Falles
- Vorausberechnung einer logischen Funktion
- Ausnutzung algebraischer Identitäten
- Erweiterung von Datenstrukturen um Wächterelemente
- Ausrollen von Schleifen
- Kombinieren von Schleifen über denselben Bereich (loop jamming)
- Entfernen invarianter Ausdrücke aus Schleifen
- Rekursionseliminierung

Transformation von Rechtsrekursion in Iteration:

```
int g(int x) {
    if (B(x)) {
        S(x); return g(E(x));
    } else {
        T(x); return p(x);
    }
}
```

Falls g nicht in B , S , T , E und p vorkommt:

```
int g(int x) {
    int x1 = x;
    while (B(x1)) {
        S(x1); x1 = E(x1);
    }
    T(x1); return p(x1);
}
```

Benutzte Systemsoftware

- Interpretierer \Rightarrow Übersetzer
- Übersetzungsoptimierung (Laufzeitsystem)
- Betriebssystem-Auswahl, -Spezialisierung
- Datenbanksystem-Auswahl, -Anpassung
- Neuimplementierung (Spezialisierung) von Bibliotheksroutinen

Hardware

- Spezialhardware (z.B. Grafikprozessoren, Chips für Sprachsynthese, Signalverarbeitung, etc.)
- Fließbandverarbeitung (Pipelining)
- Multiprozessoren, Rechnerbündel (Cluster)

4.2.2 Speicherplatzreduktion

Datenraum

- Neuberechnen statt Speichern
- Komprimierung von spärlichen Datenstrukturen
- Speichere große, identische Datenobjekte nur einmal

- Datenkompression
- Dynamische Speicherplatzvergabe
- Sätze variabler Länge

Programmraum

- Gemeinsame Nutzung von Speicherplatz
- Ersetze wiederholte Anweisungen durch Unterprogramme
- Minisprachen und Spezial-Interpreter für kompakte Darstellung
- Assembler-Codierung (nur letzter Ausweg - alles codieren!)

Cache-Optimierung Temporale und örtliche Lokalität ausnutzen

4.3 Programmier-Richtlinien

Wozu Programmier-Richtlinien? Konsistenter Stil erleichtert die Lesbarkeit; beschleunigt Einarbeitung bei Personalwechsel und Wiedereinarbeitung; Zeitersparnis bei Fehlerfindung, Erweiterung und Pflege des Programms.

Formatierung (Layout)

- Leerzeichen verbessern die Lesbarkeit
- Binäroperatoren mit je einem Zwischenraum umgeben
- Leerzeilen zur Trennung von Programmabschnitten (Klassen; Funktionen, Prozeduren, Methoden; Deklaration von Anweisungen)
- lange Zeilen umbrechen (typische Zeilenlänge: 80 Zeichen)
- lange Ausdrücke in Unterausdrücke aufteilen und deren Werte in Hilfsvariablen speichern

Namens-Konventionen

- Verbalisierung: Gedanken und Vorstellungen in Worten ausdrücken und damit ins Bewusstsein bringen
- Gute Verbalisierung: Aussagekräftige, mnemonische Namengebung; erklärende Kommentare unterstützen die Vorstellung
- Bezeichnerwahl: problemadäquate Charakterisierung von Konstanten, Variablen, Klassen, Methoden, prozeduren, etc.

Implementierungs-Dokumentation

- Prinzip der integrierten Dokumentation (Implementierungs-Dokumentation ist integraler Bestandteil jedes Programms)
 - + reduziert den Aufwand zur Dokumentenerstellung
 - + stellt sicher, dass keine Informationen verlorengehen
 - + garantiert die rechtzeitige Verfügbarkeit der Dokumentation
 - + erfordert die entwicklungsbegleitende Dokumentation
- Ziele der Implementierungs-Dokumentation: Angabe von Verwaltungsinformationen; Erleichterung der (Wieder-) Einarbeitung; Erleichterung der Wartung; Beschreibung der Entwicklungsentscheidungen; Feshalten der Verfeinerungsebenen

Folgende Verwaltungsinformationen sind am Anfang eines Programms aufzuführen:

1. Name der Programms
2. Name des bzw. der Programmautoren
3. Versionsnummer, Status, Datum
4. Aufgabe des Programms (Kurzbeschreibung)
5. Zeit- und Speicherkomplexität in O -Notation

5 Testen & Prüfen

5.1 Modul- / Softwaretestverfahren

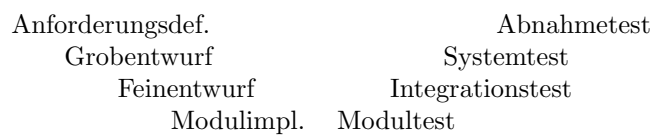
Softwaretest Ein Softwaretest führt eine einzelne Software-Komponente oder eine Konfiguration von Softwarekomponenten unter bekannten Bedingungen (Eingaben und Ausführungsumgebungen) aus und überprüft ihr Verhalten (Ausgaben und Reaktionen).

Testling Die zu überprüfende Software-Komponente oder Konfiguration wird Testling, Prüfling oder Testobjekt genannt.

Testfall Ein Testfall besteht aus einem Satz von Daten für die Ausführung eines Teils oder des ganzen Testlings.

Testrahmen Ein Testtreiber oder Testrahmen versorgen Testlinge mit Testfällen und stoßen die Ausführung der Testlinge an (interaktiv oder selbsttätig).

V-Modell



Modultest Der Modultest überprüft die Funktion eines Einzelmoduls durch Beobachtung der Verarbeitung von Testdaten.

Integrationstest Der Integrationstest überprüft schrittweise das fehlerfreie Zusammenwirken von bereits einzeln getesteten Systemkomponenten.

Systemtest Der Systemtest ist der abschliessende Test der Auftragnehmers in realer (bzw. realistischer) Umgebung ohne Kunden.

Regressionstest Ein Regressionstest ist die Wiederholung eines bereits vollständig durchgeführten Systemtests, z.B. aufgrund von Pflege, Änderung und Korrektur des betrachteten Systems.

Zweck: sicherstellen, dass das System nicht in einen schlechteren Zustand als vorher zurückgefallen ist.

Zur Vereinfachung der Testauswertung werden die Ergebnisse des Regressionstests mit den Ergebnissen des vorausgegangenen Tests verglichen.

Abnahmetest Der Abnahmetest ist der abschliessende Test der in realer Umgebung unter Beobachtung, Mitwirkung und/oder Federführung des Kunden beim Kunden.

5.2 Klassifikation testender Verfahren

Dynamische Verfahren Das übersetzte, ausführbare Programm wird mit bestimmten Testfällen versehen und ausgeführt. Das Programm wird in der realen (realistischen) Umgebung getestet. Stichprobenverfahren, Korrektheit des Programms wird nicht bewiesen.

- Strukturtest (**White Box Testing**: Bestimmen der Werte mit Kenntnis von Kontroll- und/oder Datenfluss.)
Kontrollflussorientierte Tests; Datenflussorientierte Tests
- Funktionale Tests (**Black Box Testing**: Bestimmen der Werte ohne Kenntnis von Kontroll- und/oder Datenfluss aus der Spezifikation heraus.)
- Leistungstest

Statische Verfahren Das Programm (die Komponente) wird nicht ausgeführt, sondern der Quellcode analysiert.

- Manuelle Prüfmethode (Inspektion, Reviews, Durchsichten (Walkthroughs))
- Prüfprogramme (statische Analyse von Programmen)

5.2.1 Kontrollflussorientierte Testverfahren

Zwischensprache bestehend aus:

- beliebigen Befehlen, außer solchen, die die Ausführungsreihenfolge beeinflussen (bedingte Anweisungen, Sprünge, Schleifen, etc.)
- bedingten Sprungbefehlen zu beliebigen aber festen Stellen der Befehlsfolge
- unbedingten Sprungbefehlen zu beliebigen aber festen Stellen der Befehlsfolge (zur Vereinfachung)
- einer beliebigen Anzahl von Variablen

strukturerhaltende Transformation Wir sprechen von einer strukturerhaltenden Transformation einer Quellsprache in die Zwischensprache, wenn

- (ausschließlich) die Befehle, die die Ausführungsreihenfolge beeinflussen, durch Befehlsfolgen der Zwischensprache ersetzt werden, wobei die Ausführungsreihenfolge der anderen Befehle bei gleicher Parametrisierung gleich bleibt mit der in der Quellsprache.
- alle anderen Befehle unverändert übernommen werden.
- Transformationen, bei denen Anweisungsfolgen oder bedingte Sprünge repliziert werden, vermieden werden (kein Ausrollen von Schleifen, keine Optimierungen)

Grundblock Ein Grundblock bezeichnet eine maximal lange Folge fortlaufender Anweisungen der Zwischensprache in die der Kontrollfluss nur am Anfang eintritt und die ausser am Ende keine Sprungbefehle enthält.

Kontrollflussgraph finden

1. Transformiere in Zwischensprache
2. Fasse alle Folgen, die mit einem Sprung enden, zu je einem Grundblock zusammen
3. Prüfe, ob Eintritt nur am Anfang
4. Teile ggf. auf

Zweig Eine Kante $e \in E$ in einem Kontrollflussgraph (KFG) G wird Zweig genannt. Zweige sind grundsätzlich gerichtet.

vollständiger Pfad Pfade im KFG, die mit dem Startknoten n_{start} anfangen und beim Stoppknoten n_{stopp} aufhören, heissen vollständige Pfade.

Anweisungsüberdeckung Die Teststrategie Anweisungsüberdeckung $C_{Anweisung}$ verlangt die Ausführung aller Grundblöcke des Programm P .

$$C_{Anweisung} = \frac{\text{Anzahl durchlaufener Anweisungen}}{\text{Anzahl aller Anweisungen}}$$

Nichtausreichendes Testkriterium; nicht ausführbare Programmteile können gefunden werden.

Zweigüberdeckung Die Zweigüberdeckung C_{Zweig} verlangt das Traversieren aller Zweige im KFG.

$$C_{Zweig} = \frac{\text{Anzahl traversierter Zweige}}{\text{Anzahl aller Zweige}}$$

Zweige, die nicht ausgeführt werden, können entdeckt werden.

Weder Kombination von Zweigen noch komplexe Bedingungen berücksichtigt; Schleifen nicht ausreichend getestet; fehlende Zweige nicht testbar.

Pfadüberdeckung Die Pfadüberdeckung fordert die Ausführung aller unterschiedlichen Pfade im Programm.

- Pfadanzahl wächst bei Schleifen dramatisch an

- Manche Pfade können nicht ausführbar sein, durch sich gegenseitig ausschliessende Bedingungen
- Mächtigste KFO Teststrategie
- Nicht praktikabel

Boundary-Interior Pfadtest Praktikabler Mittelweg für Schleifen zwischen Zweigüberdeckung und Pfadüberdeckung. Ansatz: Bilde Äquivalenzklassen über alle möglichen Pfade und wähle aus jeder Klasse einen Repräsentanten. Annahme: Getestet werden müssen

- unterschiedliche Pfade durch die oberste Ebene des Programms (kein Betreten der Schleifen)
- unterschiedliche Pfade durch Schleifen
- unterschiedliche "Grenzttests" bei Schleifen

Schleifenquerer Einen Teilpfad, der genau ein Mal einen Schleifenkörper und die zugehörige Bedingung einschließt, nennen wir Schleifenquerer.

BI-äquivalent Zwei Pfade, die sich ausschliesslich durch Anzahl oder Subpfade der Schleifenquerer unterscheiden, nennen wir BI-äquivalent.

Grenzttest Ein Grenzttest einer Schleife führt zum Betreten aber nicht zum Wiederholen des Schleifenkörpers (ein Grenzttest führt also einen einzelnen Schleifenquerer aus).

Interieurtest Ein Interieurtest einer Schleife führt zum Betreten und mindestens einmaligen Wiederholen des Schleifenkörpers (ein Interieurtest führt also mindestens zwei Schleifenquerer hintereinander aus).

Äquivalenzklassenbildung Betrachte die Menge aller BI-äquivalenter Pfade bzgl. einer geg. Schleife. Teile diese Mengen sukzessive wie folgt auf:

1. Nach Grenzttest-Pfaden und Interieurtest-Pfaden
2. Nach Pfaden, die die Schleifen auf verschiedenen Wegen verlassen (z.B. durch Ausnahmen)
3. Nach Grenzttest-Pfaden mit verschiedenen Schleifenquerern
4. Nach Interieurtest-Pfaden, die im zweiten Durchlauf verschiedene Schleifenquerer verwenden

Wähle mindestens einen Pfad aus allen so erzeugten Klassen zum Test.

einfache Bedingungsüberdeckung Die einfache Bedingungsüberdeckung fordert, dass jede atomare Bedingung einmal mit W und einmal mit F belegt wird.

mehrfache Bedingungsüberdeckung Die mehrfache Bedingungsüberdeckung fordert, dass die atomaren Bedingungen mit allen möglichen Kombinationen der Wahrheitswerte W und F belegt werden.

Minimal-Mehrfach BÜ Die minimal-mehrfache Bedingungsüberdeckung fordert, dass jede Bedingung, ob atomar oder zusammengesetzt, jeweils zu W und F evaluieren muss.

Zusammenfassung

- Anweisungsüberdeckung nur angebracht, wenn keine Schleifen oder Bedingungen ausgeführt werden, ansonsten nicht ausreichend.
- Zweigüberdeckung angebracht, wenn keine Schleifen und nur atomare Bedingungen vorhanden, sonst Kombination aus boundary-interior Pfadtest und minimal-mehrfacher BÜ.
- Pfadüberdeckung aufwendigstes und schon für kleine Programme nicht zu realisierendes Kriterium. Einfache BÜ nicht ausreichendes Kriterium, schwächer als Anweisungsüberdeckung.
- Mehrfach BÜ sehr aufwendig, deckt keine Fehler in Bedingungsstruktur auf.
- Minimal-Mehrfache BÜ gute Erweiterung des Konzepts der Zweigüberdeckung.

5.2.2 Funktionale Tests

Ziel Testen der spezifizierten Funktionalität

- Testfälle aus der Spezifikation ableiten
- Interne Struktur des Testlings nicht berücksichtigt, weil für den Tester unbekannt
- Vorteile: Testfälle unabhängig von der Implementierung erstellbar; vermeidet "Kurz-sichtigkeit" bei der Auswahl
- Nachteil: mögliche kritische Pfade nicht bekannt & nicht getestet

Funktionale Äquivalenzklassenbildung Bildung der Äquivalenzklassen (Ansatzidee)

- Partitionierung ausgehend von einer großen ÄK
- Aufteilung entlang Definitionsbereich
- Aufteilung entlang anzunehmenden (!) Verarbeitungsmethodengrenzen

Auswahl der Repräsentanten

Sei m die Anzahl der ÄK der Eingabeparameter und n die Anzahl der ÄK der Ausgabeparameter. Dann entstehen $\leq m \cdot n$ verschiedene ÄK aus denen jeweils 1 beliebiger Repräsentant zum Testen verwendet wird.

Grenzwertanalyse Weiterentwicklung der funktionalen Äquivalenzklassenbildung

- Off-by-one: Knapp daneben ist auch vorbei
- Testfälle, die die Grenzen der Äquivalenzklassen und deren unmittelbare Umgebung abdecken sind besonders effektiv.

⇒ Verwende nicht irgendein Element aus der ÄK, sondern solche auf und um den Rand (Annäherung von beiden Seiten)

Zufallstest Testen der Funktionen mit zufälligen Testfällen.

Test von Zustandsautomaten Hat eine Komponente einen internen Zustand, können Testfälle aus den Zustandsübergängen abgeleitet werden. Ziel: mindestens einmaliges Durchlaufen aller Übergänge.

5.2.3 Leistungstests

Lasttest Testet das System/die Komponente auf Zuverlässigkeit und das Einhalten der Spezifikation im erlaubten Grenzbereich.

Stresstest Testet das Verhalten des Systems beim Überschreiten der definierten Grenzen.

5.3 Software-Inspektionen

Inspektion Die Inspektion ist eine formale Qualitätssicherungstechnik, bei der Anforderungen, Entwurf oder Code eingehend durch eine vom Autor verschiedene Person oder eine Gruppe von Personen begutachtet werden. Zweck ist das Finden von Fehlern, Verstößen gegen Entwicklungsstandards und anderen Problemen.

Phasen einer Inspektion

1. **Vorbereitung:** Teilnehmer und ihre Rollen festlegen; Dokumente und Formulare vorbereiten; Lesetechniken festlegen; zeitlichen Ablauf planen
2. **Individuelle Fehlersuche:** jeder Inspektor prüft für sich das Dokument, schreibt alle Problempunkte (und Stelle im Dokument) auf

3. **Gruppensitzung:** Individuell gefundene Problempunkte sammeln; jeden einzelnen Problempunkt besprechen; Fragen zum Dokument klären
4. **Nachbereitung:** Liste mit allen Problempunkten wird an den Editor des Dokuments weitergeleitet; Editor leitet Änderung des Dokuments ein; alle Problempunkte werden bearbeitet (Bearbeitung wird überprüft); Restdefekte schätzen; Dokument wird freigegeben wenn $\#$ geschätzte Defekte $< N$
5. **Prozessverbesserung:** Prüflisten und Szenarien anpassen; Standards für Dokumente erarbeiten; Defekt-Klassifikationsschema anpassen; Formulare verbessern; Planung und Durchführung verbessern

5.4 Testwerkzeuge

Zusicherungen Boolesche Funktionen: Vor- und Nachbedingungen (z.B. dass ein übergebener Parameter positiv sein muss oder eine Referenz nicht unbestimmt sein darf); werden zur Laufzeit ausgeführt.

Im Fehlerfall melden sie sich mit einer Ausnahme oder Fehlermeldung. Zusicherungen können zu- und abgeschaltet werden.

- Konvention für **öffentliche** Methoden: Überprüfung der Eingabeparameter nicht mit Zusicherung, sondern mit `IllegalArgumentException`; Folge: Klientenprogramme können darauf reagieren.
- Konvention für **private** Methoden: Eingabeparameter, Nachbedingungen und Invarianten aller privaten Methoden mit Zusicherungen überprüfen; Grund: Verletzung ist unerwarteter Fehler; Aber: Falsche Parameter bei öffentlichen Methoden sind nicht unerwartet.

Mit Zusicherungen können Missverständnisse und Fehlinterpretationen der Programmierer rasch aufgedeckt werden. In Produktionsläufen werden Zusicherungen aus Leistungsgründen abgeschaltet. Bei Auftreten eines Fehlers oder beim Testen von Programmänderungen werden die Zusicherungen wieder zugeschaltet.

Zusicherungen sind keine Testfälle, es werden lediglich bestimmte Bedingungen im Programmablauf überprüft. Sofern sie zugeschaltet sind, werden sie beim Ablauf von Testfällen mit ausgeführt.

Automatisch ablaufende Testfälle

- Testfall: Ausführen eines Programms unter bekannten Bedingungen; Eingabedaten geben; Ergebnis bekannt; Ziel: Aufdecken von Fehlern
- Automatisch ablaufende Testfälle: Überprüfen Ergebnis ihrer Ausführung selbst; Rückgabe: boolescher Wert (erfolgreich oder fehlgeschlagen); Optional: Auslösen einer Ausnahme bei Fehlschlag
- Warum Rahmenarchitektur? Vereinfacht das Schreiben von Testfällen; Zusammenfassung von Testfällen zu Testsuiten; automatisches Ablaufen ganzer Testsuiten und nur Melden der Versager; wirkt standardisierend
- Regressionstest: Wiederholtes Ausführen einer Testsuit nach Programmänderungen; Voraussetzung: Automatische Ausführung, da in der Praxis oft Tausende von Testfällen

Zusammenfassung

- Automatisch ausführbare Testfälle sind besonders für den Regressionstest wichtig.
- Regressionstests helfen verhindern, dass alte Fehler wieder auftauchen.
- JUnit und ähnliche Regressionstest-Werkzeuge sind in der Praxis extrem wichtig.
- Sie erfordern, dass Testfälle sich selbst überprüfen.
- Können mit Zusicherungen kombiniert werden.

6 Die Abnahme- & Einführungsphase

Das fertiggestellte Gesamtprodukt wird abgenommen und beim Anwender eingeführt, d.h. in Betrieb genommen. Ab diesem Zeitpunkt unterliegt das Produkt dann der Wartung & Pflege.

6.1 Die Abnahmephase

Tätigkeiten

- Übergabe des Gesamtprodukts einschl. der gesamten Dokumentation an den Auftraggeber.
- Mit der Übernahme verbunden ist im allg. ein Abnahmetest.
- Innerhalb einer Abnahme-Testserie ist es auch sinnvoll, Belastungs- oder Stresstests durchzuführen.
- Das Ergebnis der Abnahmephase ist ein Abnahmeprotokoll.

Abnahme

- Nach erfolgreichen Tests des Produkts durch den Auftraggeber.
- Die formale Abnahme ist die (schriftliche) Erklärung der Annahme (im juristischen Sinne) eines Produkts durch den Auftraggeber.

Externer Auftraggeber

- Abnahmetest hängt auch davon ab, ob der Auftraggeber das Produkt nur nutzt, aber nicht wartet und pflegt oder nutzt und selbst wartet und pflegt.
- Welche Alternative der Auftraggeber wählt, sollte bereits bei der Auftragsvergabe bekannt sein. Die für den Auftraggeber relevanten Qualitätsmerkmale hängen von der gewählten Alternative ab.

Produktnutzung Die Qualitätsmerkmale **Nutzbarkeit**, **Integrität**, **Effizienz**, **Korrektheit** und **Zuverlässigkeit** sind wesentlich.

Abnahmetest

- Erfüllung der Qualitätsmerkmale prüfen
- Macht Auftraggeber Wartung & Pflege selbst, dann benötigt er: die gesamte Entwurfs- & Implementierungsdokumentation; eine sorgfältige Einführung in die Architektur; die gesamten Testeinrichtungen und Testfälle

6.2 Die Einführungsphase

Tätigkeiten

- Installation des Produkts: Einrichtung des Produkts in dessen Zielumgebung zum Zwecke des Betriebs
- Schulung der Benutzer und des Betriebspersonals: Nach der Installation des Produkts sind die Benutzer in die Handhabung des Produkts einzuweisen
- Inbetriebnahme des Produkts: Übergang zwischen Installation und Betrieb

Einführungsprotokoll Alle Vorkommnisse, die in der Einführungsphase auftreten, werden festgehalten

Einführung muss sorgfältig geplant werden

Umfangreiche Produkteinführungen wie Innovationseinführungen zu behandeln

Umstellung Bei Ersatz eines existierenden Systems: Zeitplanung; Wichtige Aufgaben: Umstellung der Datenbestände, ...

Das größte Problem ergibt sich bei der Übertragung "lebender" Datenbestände

Inbetriebnahme 3 Arten:

- **Direkte Umstellung:** Es wird unmittelbar von dem alten auf das neue System übergegangen.
- **Parallellauf:** Die Bewegungsdaten werden sowohl im alten als auch im neuen System verarbeitet, so dass die Ergebnisse verglichen werden können.
- **Versuchslauf:** Neues System arbeitet mit Daten aus vergangenen Perioden *oder* Einführung des neuen Systems in einzelnen Stufen (versch. Funktionsbereiche werden sukzessiv übernommen)

7 Die Wartungs- & Pflegephase

Die Wartung und Pflege beginnt mit der erfolgreichen Abnahme und Einführung eines Software-Produkts.

Alterung von Software

- Software, bei der nicht ständig Fehler behoben und Anpassungen sowohl an die Umwelt als auch an neue Anforderungen vorgenommen werden, altert und ist irgendwann veraltet.
- Sie kann dann nicht mehr für den ursprünglich vorgesehenen Zweck eingesetzt werden.
- "Software veraltet in dem Maße, wie sie mit der Wirklichkeit nicht Schritt hält."

Wartung Lokalisierung und Behebung von Fehlerursachen von in Betrieb befindlichen Software-Produkten, wenn die Fehlerwirkung bekannt ist. Ist ereignisgesteuert, daher schwer planbar.

Pflege Lokalisierung und Durchführung von Änderungen und Erweiterungen von in Betrieb befindlichen Software-Produkten, wenn die Art der gewünschten Änderungen/Erweiterungen festliegt. Ist planbar.

Faustregel Der Aufwand für die Wartung & Pflege ist typischerweise um einen Faktor von 2 bis 4 größer als der Entwicklungsaufwand für ein umfangreiches Produkt.

7.1 Aufgaben und ihr Aufwand

7.1.1 4 Kategorien der Wartungs- & Pflegephase

korrektive Tätigkeiten

(1) Stabilisierung / Korrektur:

- Alle Tätigkeiten, die dazu dienen, Fehler zu beheben
- Es kann sich dabei um Fehler handeln, die bereits bei der Entwicklung in das Produkt gelangt sind, oder um Fehler, die bei der Wartung neu entstehen.
- Software-Produkte werden mit durchschnittlich 0,75% Fehlern pro 100 Anweisungen freigegeben.
- Besonders schnell vermehren sich Wartungsfehler, die so genannten Second Level Defects.

(2) Optimierung / Leistungsverbesserung:

- Optimierung erfolgt selten vor der ersten Freigabe: Sobald ein Produkt funktionsfähig ist, wird es freigegeben.
- Optimierung bleibt der Wartung vorbehalten: Alle Aktivitäten um Leistung zu verbessern (Feinoptimierung, Reduzierung des Speicherbedarfs, Restrukturierungen)

Progressive Tätigkeiten

(3) Anpassung / Änderung: Anpassungen werden durch Wandlungen in der Umwelt erzwungen.

- Änderung in der technischen Umgebung
- Änderung in den Benutzungsoberflächen
- Änderung in den Funktionen

(4) Erweiterungen:

- Führen zu einer funktionalen Ergänzung des Produkts
- Funktionen, die bei der Erstentwicklung vorgesehen oder geplant, aber nicht implementiert wurden, werden eingebaut
- Oder es ergeben sich neue Funktionen aus den Erfordernissen des Betriebs der Software

7.2 Aufwandsschätzung

Die Aufwandsschätzung ist eine der schwierigsten und ungenauesten Tätigkeiten in der Softwaretechnik, doch sie ist entscheidend für den wirtschaftlichen Erfolg eines Softwarehauses.

$$\text{Gewinn (Verlust)} = \text{Deckungsbeitrag} \cdot \text{geschätzte Menge} - \text{einn. Entwicklungskosten}$$

Einflussfaktoren

- **Lines of Code (LOC):** Anzahl der Programmzeilen
- Ergebnis: geschätzter Aufwand in Mitarbeiterjahren (MJ) oder Mitarbeitermonaten (MM)
- 1 MJ = 9 MM oder 10 MM
- wichtige Einflussfaktoren: Quantität, Qualität, Entwicklungsdauer, Kosten

Basismethoden

- **Analogiemethode:** Vergleich der zu schätzenden Entwicklung mit bereits abgeschlossenen Produkt-Entwicklungen anhand von Ähnlichkeitskriterien.
- **Relationsmethode:** Das zu schätzende Produkt wird direkt mit ähnlichen Entwicklungen verglichen.
- **Multiplikatormethode:** Das zu entwickelnde System wird soweit in Teilprodukte zerlegt, bis jedem Teilprodukt ein bereits feststehender Aufwand zugeordnet werden kann (z.B. in LOC).
- **parametrische Schätzgleichungen:** Durch Korrelationsanalysen wird ermittelt, welche Faktoren welchen wertmäßigen Einfluss auf den Gesamtaufwand haben.
- **Phasenaufteilung:** Aus abgeschlossenen Entwicklungen wird ermittelt, wie der Aufwand sich auf die einzelnen Entwicklungsphasen verteilt hat.
- **Gewichtungsmethode:** Zunächst werden Faktoren festgelegt, die für die Schätzung relevant sind (subjektiv oder objektiv). Den Faktorausprägungen sind Werte zugeordnet. Die Werte aller Faktoren werden nach einer vorgegebenen mathematischen Formel verknüpft und ergeben dann des Gesamtaufwand (Bsp.: Funktionspunkte)

7.2.1 Funktionspunkte

Bei der Funktionspunktmethode werden Eingaben, Ausgaben, Anfragen und Datenbestände einer Anwendung in ihrer Komplexität bepunktet. Die Gesamtzahl dieser Funktionspunkte wird dann mit mehreren Einflussfaktoren, die den Aufwand erhöhen oder erniedrigen, skaliert. Abschliessend werden die Funktionspunkte aufgrund historischer Daten in Mitarbeitermonate umgerechnet.

Vorgehensweise

1. Klassifizierung: einfach, mittel, komplex
2. Eintrag in Berechnungsformular
3. Bewertung der Einflussfaktoren
4. Berechnung der bewerteten Funktionspunkte
5. Ablesen des Aufwands in MM
6. Aktualisierung der empirischen Daten

8 Prozessmodelle

Programmieren durch Probieren

Wasserfallmodell

Planung	<i>Lastenheft, Projektplan, Kalkulation</i>
Definition	<i>Pflichtenheft, GUI-Beschr., evtl. Benutzerhandbuch</i>
Entwurf	<i>Entwurf dok., Modulführer</i>
Implementierung	<i>Komponenten-Doku, Testeinrichtung</i>
Testen	<i>fertiges System</i>
Einsatz & Wartung	

V-Modell XT Entwicklungsstandard für IT-Systeme der öffentlichen Hand in Deutschland. Aktivitäten, Produkte und Verantwortlichkeiten werden festgelegt, jedoch keine Reihenfolge/Phasengrenzen

Prototypmodell

- Geeignet für Systeme, für die keine vollständige Spezifikation ohne explorative Entwicklung oder Experimentation erstellt werden kann.
- Der Prototyp (eingeschränkt funktionsfähiges System) kann Arbeitsmoral und Vertrauen zwischen Anbieter und Kunden stärken.
- Wichtig: Prototyp wegwerfen!

Iteratives Modell Idee: Zumindest Teile der Funktionalität lassen sich klar definieren und realisieren. Daher: Funktionalität wird Schritt für Schritt erstellt und dem Produkt "hinzugefügt". Gleiche Vorteile und Einsatzgebiete wie Prototypmodell.

Synchronisiere und Stabilisiere Ansatz: Organisiere Programmierer eines Projekts in "kleinen Hacker Teams" (Freiheit für eigene Ideen/Entwürfe). Aber: Synchronisiere regelmäßig (nächtlich) und Stabilisiere regelmäßig (Milestones, 3 Monate)
Drei Phase: Planungsphase, Entwicklungsphase, Stabilisierungsphase

8.1 Agiles Vorgehensmodell Extreme Programming XP

Der **Kunde** trifft die geschäftsrelevanten Entscheidungen und das **Entwicklungsteam** trifft die technischen Entscheidungen. Der **Kunde** kann seine Entscheidung jederzeit korrigieren.

XP-Prinzipien

- Schnelle Rückkopplung
- So einfach wie möglich
- Inkrementelle Änderung
- Hohe Qualität

Kostenkurve Bisherige Annahme: exponentielle Kostenkurve, neue Annahme von XP: asymptotische Kostenkurve

8.1.1 XP-Praktiken

Planungsspiel

- Planungsspiel ersetzt bisherige Anforderungsanalyse
- Planungszeitraum: 1-4 Monate
- Ungefährer Projektplan wird vom Kunden und vom Entwicklungsteam gemeinsam erstellt (Kunde äußert seine Wünsche, Team schätzt Kosten)
- Kunde priorisiert seine Wünsche

- Gesamtes Team stellt Zeitplan für die wichtigsten Kundenwünsche auf

Iterationsmodell

- Findet mehrmals innerhalb einer Freigabe statt
- Planungshorizont: 1-4 Wochen
- Alle Aufgaben werden auf Aufgaben-Karten notiert (Aufteilung von Stories in kleinere Teile, Identifizierung zusätzlicher Aufgaben)
- Teammitglied akzeptiert Aufgabe und schätzt Aufwand für diese Aufgabe
- Ausgleich zwischen Teammitgliedern
- Rückkopplung zum Kunden

Testen

- Programmierer schreiben automatische Modul-Tests, diese laufen immer zu 100%
- Kunde spezifiziert Funktionstests, die Team implementiert
- Testausführung automatisch!
- **Testgetriebene Entwicklung (TGE):**
 - Entwurf von Testfällen und erst danach Implementierung: Test-Zuerst
 - Entwicklung wird von den Tests geleitet, deshalb TGE
 - dafür keine Spezifikation, keine Entwurfsdokumentation
 - erfolgreiche Tests erhöhen Vertrauen von Team und Kunde in das Produkt

Refaktorisieren

- Umstrukturieren des Programmtextes ohne das nach Außen sichtbare Verhalten zu ändern, wenn neue Funktionen es erfordern.
- Ziel: Stets ein möglichst einfacher Entwurf, der für die bisher implementierten Anforderungen ausreicht (kein änderungsfreundlicher, vorausschauender Entwurf nötig)
- Ermöglicht durch: automatische Tests, kollektiver Code-Besitz

Paar-Programmierung

- Jede Programmierstätigkeit wird im Paar ausgeführt.
- Arbeit an einer Tastatur, einer Maus und einem Bildschirm.
- Rollenverteilung: Einer denkt an Implementierung des Algorithmus, der andere denkt strategisch und führt ständige Durchsicht durch.
- Führt zu nachweisbar höherer Qualität des Programmtextes.
- Strittig: Wirklicher Vorteil, Ressourcenauslastung

8.1.2 Kritik an XP

Kritik

- Asymptotische Kostenkurve beruht auf der Erfahrung einzelner.
- Fehlende Produktdokumentation.
- Nicht reproduzierbarer ad-hoc Prozess.
- Ressourcenauslastung bei Paar-Programmierung.

Probleme bei Einführung

- Kunde muss mit "ins Boot" geholt werden
- Softwareingenieure sind auf Vorausschau getrimmt. Folge kann sein: zu komplexer oder umfangreicher Entwurf
- TGE erfordert umdenken und umlernen
- Gleiche Wellenlänge bei Paar-Programmierung
- Respekt vor fremdem Programmtext

8.1.3 Zusammenfassung

XP ist Software-Prozess: geeignet für vage und sich schnell ändernde Anforderungen; für kleines Entwicklerteam; ohne zeitraubenden Verwaltungsaufwand.
Softwareentwicklung mit leichtem Gepäck.

9 Konfigurationsverwaltung

9.1 Grundzüge der Konfigurationsverwaltung (KV)

Argumente für KV

- verschiedene Dateisysteme
- unterschiedliche Rechner
- diverse Plattformen
- Komponenten von Drittherstellern
- mehrere Teams/Firmen
- geographisch verteilte Entwicklerteams

Konfigurationsmanagement (ISO 9001) KV stellt einen Mechanismus zur Identifizierung, Lenkung und Rückverfolgung der Versionen jedes Software-Elements dar.

Software-Element (SE) Ein Software-Element ist jeder indentifizierbare Bestandteil des entstehenden Produktes oder der entstehenden Produktlinie.

- besitzen systemweit eindeutigen Bezeichner
- Änderung am Element erzeugt neuen Bezeichner, um Fehlidentifikation zu vermeiden
- Unterschiede: **Quellelement**: manuell erzeugt (z.B. mit Editor); **abgeleitetes Element**: automatisch generiert (z.B. durch Übersetzer)

Version

- Eine Version ist die Ausprägung eines SEs zu einem bestimmten Zeitpunkt.
- **Revisionen** sind zeitlich nacheinander liegende Versionen (Entwicklungsstände).
- **Varianten** sind alternative Versionen (Anpassungen, oder mit alternativen Datenstrukturen/Algorithmen).

Versionsnummern (VN) Bestehen aus mindestens zwei Teilen (**Freigabe**-Nr (release), **laufende** Nr (level)); VN: Release.Level

Einbuchen/Ausbuchen (Check-In/Chack-Out)

- SE werden in Archiven gesammelt.
- **Ausbuchen**: Holt Kopie aus Archiv; reserviert Kopie für den Ausbucher, was heisst, dass nur dieser die nächste Revision ablegen darf (striktes Ausbuchen); Kopie darf geändert und wieder eingebucht werden
- **Einbuchen**: Schreibt Kopie ins Archiv zurück; löscht Reservierung; erweitert Historie um: Autor des Elements/der Änderung, Einbuchungszeitpunkt, Logbucheintrag (der Änderungen zusammenfasst)
- Eingebuchtes Element ist nicht mehr änderbar. Erst erneutes Ausbuchen erlaubt Änderungen.
- Im System können mehrere Dateien gleichzeitig ausgebucht sein. Lesezugriff durch Reservierung nicht verhindert. Unterschied: striktes und optimistisches (mehrfaches) Einbuchen/Ausbuchen

striktes Ausbuchen

- Nur eine Ausbuchung gleichzeitig ist erlaubt.
- Ausbucher hat exklusives Änderungsrecht.
- Vorteil: kein Verschmelzungsaufwand beim Zurückschreiben.
- Nachteil: immer nur einer kann eine Version ändern.

Optimistisches Ausbuchen

- Mehrere Ausbuchungen gleichzeitig erlaubt.
- Mehrere Entwickler arbeiten an der gleichen Programmversion.
- Vorteil: Mehrere Entwickler können eine Version ändern.
- Nachteil: Aufwand beim Zusammenführen der Versionen (der Schnellere gewinnt).

Varianten Sequentielle Versionsstämme reichen für Praxis oft nicht aus. Varianten erlauben das Ändern ähnlicher Versionen eines SE. Varianten können:

- parallel Entwicklungslinien darstellen
- unterschiedliche Implementierung derselben Schnittstelle sein, bei gleicher Funktion
- Anpassungen an unterschiedliche Bibliotheken, Geräte, GUIs, Nutzer sein
- auf unterschiedliche Hardware oder Systemsoftware-Konstellationen zugeschnitten sein
- ab bestimmten Abstraktionsniveau nicht mehr unterschieden werden

Variantennummern Setzen sich zusammen aus Nummer der Grundversion und Nummer der Variante:

Release.Level.Variante.Level

(Software-) Konfiguration Eine (Software-) Konfiguration ist eine benannte und formal freigegebene Menge von Software-Elementen, mit den jeweils gültigen Versionsangaben, die zu einem bestimmten Zeitpunkt im Produktlebenszyklus in ihrer Wirkungsweise und ihren Schnittstellen aufeinander abgestimmt sind und gemeinsam eine vorgesehene Aufgabe erfüllen sollen.

- Konfiguration umfasst bestimmte Versionen von SE.
- Konfigurations-Identifikationsdokument (KID) bestimmt Zuordnung von Konfiguration zu Versionen der SE.
- KID ist selber im KV, d.h. besitzt Versionsnummer

Bestandteile einer (Software-) Konfiguration:

- Quellelemente (z.B. Programmtext, Dokumentation, Konfigurationsdateien für zu bauendes System)
- Abgeleitete Elemente, wenn sie teuer in Erstellung sind.
- Werkzeuge (Übersetzer, ...)
 - Systemrelevante, wichtig für Ausführung des Systems (z.B. Laufzeitübersetzer)
 - Entwicklungsrelevante, bekommt Kunde nicht zu sehen
- KID von Subsystemen (hierarchische Struktur einer Konfiguration - siehe Kompositum)

Verwaltung von Versionen Vollständiges Abspeichern jeder Versionen ist platzaufwendig. Alternative:

- **Vorwärts-Deltas:** Speichere Grundversion und die daran durchgeführten Änderungen
 - Vorteil: Schneller Zugriff auf frühere Versionen.
 - Nachteil: Langsamer Zugriff auf aktuellere Version.
- **Rückwärts-Deltas:** Speichere aktuelle Version und die Änderungen für frühere Versionen

Ein Delta ist der Unterschied zwischen zwei Versionen; ein komprimiertes Änderungs-Skript, das eine Version in die andere überführt. Bei Software ist ein Delta i.d.R. etwa 1-2% des Umfangs einer (Voll-) Version.

9.2 Revision Control System (RCS)

Grundlegende Kommandos

<code>ci</code>	Check-In
<code>co</code>	Check-Out
<code>-l <filename></code>	Version im Archiv bekommt Reservierung (Lock)
<code>rcs</code>	Verwaltung des Archivs
<code>-i <filename></code>	Legt neues Archiv für Datei <code>filename</code> an; Archiv ist leer; Archiv wird in Unterverzeichnis RCS angelegt, falls existent, sonst im lokalen Verzeichnis; Beschreibung der Datei wird verlangt, nicht der Logbucheintrag
<code>rlog</code>	Verwaltungsinformation und Logbuch

Zusammenfassung

- Einfache Schnittstelle für Versionierung einzelner Dateien.
- Durch Kommandozeilenoperationen Zugriff auf alle Versionen.
- Sicht auf mehrere Dateien nicht vorhanden (→ CVS)

9.3 Concurrent Version System (CVS)

CVS

- CVS erweitert Versionierung auf ganze Verzeichnisse und deren Inhalt.
- Benutzt die Versionierung von RCS.
- Implementiert optimistisches Aus-/Einbuchen.
- Normalerweise betrachtet CVS jedes Verzeichnisse mit seinem Inhalt als Modul. Separate Module, die mehrere Verzeichnisse enthalten direkt spezifizieren. Modifiziere dazu Datei `CVSROOT/modules` und trage Modulbeschreibung ein.

Zusammenfassung

- Einfaches Werkzeug zur Verwaltung von Projekten.
- Geringer Netzverkehr.
- Standard bei Opensource-Projekten.
- Kein Umbenennen von Dateien möglich.
- Kein KV, da Benutzer bei Konfiguration alleine gelassen wird.
- Gut dokumentiert.

10 Einführung in XML 1.0

10.1 Allgemeines

XML = eXtensible Markup Language

XML ist ein wichtiger Standard, um strukturierte Dokumente im Internet von Anwendung zu Anwendung auszutauschen

- XML Dokumente sind strukturiert: (Attribut-Wert)-Paare, baumartig gegliedert
- XML Dokumente sind von Anwendungen einfach verarbeitbar
- XML Dokumente sind Unicode-Klartext und vom Menschen ohne weiteres lesbar
- XML Dokumente sind unabhängig von Anwendung, Plattform und Programmiersprache

Verantwortung World Wide Web Consortium (W3C)

10.2 Einführung

XML ist

- eine formale Sprache. Sie ist mit einer EBNF-Grammatik beschrieben.
- eine Untermenge der Sprache "Standard Generalized Markup Language", SGML, von 1969.
- wohlgeformt.
- erweiterbar, d.h. es gibt spezielle Vereinbarungen, mit der Vokabular und Struktur eines XML Dokumentes genau festgelegt werden kann (genau genommen wird damit die Grammatik eingeschränkt oder spezialisiert).

Markup Markup sind Auszeichnungen in Daten, z.B. Korrekturzeichen in einem Text.

Formatierungsanweisungen wie `<p>` in HTML sind auch eine Form der Textauszeichnung; ferner Trenner, Klammerungen oder Markierungen. Generell versteht man unter Auszeichnung (Markup) alle zusätzlichen Zeichen und Zeichenfolgen, die in einem Dokument enthalten sind, aber nicht zu den eigentlichen Daten gehören.

Aufbau von XML Dokumenten Ein XML Dokument besteht aus einem Prolog gefolgt von einem Wurzelement. Der Prolog gibt die XML Version an, plus optional den Zeichensatz und den Dokumenttyp. Das Wurzelement enthält die Daten in Form von geschachtelten Elementen und Attributen. Kommentare sehen wie folgt aus:

```
<!-- das ist ein Kommentar -->
```

Ausserdem gibt es noch optionale Verarbeitungsanweisungen.

Beispiel

```
<?xml version="1.0" encoding="utf-8" ?>

<!-- Personalverzeichnis mit nur einer Person -->

<Universität Name="Universität Karlsruhe">
  <Person Bezeichner="wtichy" EPost='tichy@ipd.uka.de'>
    <Name>
      Walter Tichy
    </Name>
    <Telefon>
      0721-608-7343
    </Telefon>
    <Stadt PLZ='76128'>
      Karlsruhe
    </Stadt>
```

```

    </Person>
</Universität>

```

XML Format

- Ein Element besteht aus einer Startmarke, einer Endmarke und der Information zwischen den Marken.
- Der Inhalt eines Elements kann eine nahezu beliebige Zeichenfolge sein, oder eingebettete Elemente, oder leer (leeres Element).
- Elemente können mit Attributen annotiert werden. Attribute enthalten zusätzliche Daten über das Element und dessen Inhalt. Die Attribute erscheinen nach dem Namen der Startmarke vor dem schließenden >-Zeichen.

XML vs HTML

- Bei XML ist Groß-/Kleinschreibung bedeutsam, bei HTML nicht.
- XML Dokumente müssen wohlgeformt sein, HTML Dokumente nicht.
- HTML ist eigentlich eine Formatierungssprache, nicht ein Datenaustauschformat.

wohlgeformtes XML Regeln für wohlgeformtes XML:

- Alle Elemente müssen sowohl eine Start- als auch eine gleichnamige Endmarke haben. Ausnahme: Leere Elemente (z.B. <Nichts/>)
- Alle Start- und Endmarken müssen richtig geschachtelt sein.

Element

- Information soll weiter untergliedert werden.
- Enthält eine längere Informationseinheit.
- Es gibt mehrere solche Elemente (z.B. <Person>).

Attribut

- Kurze Informationseinheit (nur ein Wort).
- Nur bestimmte Standardwerte zugelassen.
- (Meta-) Information von höherer Ordnung.

gültiges XML

- wohlgeformt
- Dokumenttypdefinition vorhanden (sie gibt vor, welche Elemente und Attribute an welchen Stellen vorkommen dürfen)
- enthalten genau die Elemente und Attribute, die von der Dokumenttypdefinition vorgegeben sind

Beispiel einer DTD (zum XML Beispiel oben)

```

<!DOCTYPE Universität [
  <!ELEMENT Universität (Person*)>
  <!ATTLIST Universität
    Name CDATA #REQUIRED>
  <!ELEMENT Person
    (Name, Telefon?, Stadt, Land?)>
  <!ATTLIST Person
    Bezeichner ID #REQUIRED
    EPost CDATA #IMPLIED>
  <!ELEMENT Name (#PCDATA)> <!ELEMENT Telefon (#PCDATA)>
  <!ELEMENT Stadt (#PCDATA)> <!ELEMENT Land (#PCDATA)>
  <!ATTLIST Stadt PLZ CDATA #REQUIRED>
] >

```

Verarbeitung (in Programmen)

- Lösung 1: Document Object Model (DOM)
Generiert aus einem XML Dokument eine komplette Objektstruktur im Hauptspeicher. Dieses kann dann direkt verarbeitet werden.
Vorteilhaft, falls das Dokument komplett und mehrmals benötigt wird. Nachteilig, falls das Dokument sehr groß ist oder nur teilweise benötigt wird, denn der Speicherbedarf kann zu hoch werden.
- Lösung 2: Simple API for XML (SAX)
Es wird keine Objektstruktur erzeugt. Statt dessen wird das XML Dokument durchlaufen und beim Vorkommen von Elementen, Verarbeitungsanweisungen etc. wird ein Ereignis ausgelöst. Dieses Ereignis bewirkt das Anstoßen einer vom Nutzer definierten Methode. Diese kann dann zusätzliche Daten auslesen und verarbeiten.
Vorteil: benötigt weniger Speicherbedarf, wenn nicht alle Daten in den Hauptspeicher übernommen werden müssen. Nachteil: mehr Programmieraufwand.

XML Namensräume

- XML Namensräume stellen eine XML-basierte Syntax zur Verfügung um Element- und Attributnamen eindeutig zu identifizieren.
- Grundidee: Erweiterte Element- und Attributnamen so, dass auch nach der Vereinigung beliebiger Dokumente wieder eindeutige Bezeichner entstehen.
- Identifikationsmechanismus sowohl mächtig als auch einfach zu handhaben.
- Namensschema der Uniform Resource Identification (URI)
- Die Idee ist die URIs als weltweit eindeutige Bezeichner zu verwenden.

Beispiel für Namesräume:

```
<?xml version="1.0" encoding="utf-8" ?>

  <Disney:order xmlns:Disney="http://www.disney.com">
    <Disney:blanket>Mickey Mouse blanket</Disney:blanket>
  </Disney:order>

  <SAP:order xmlns:SAP="http://www.sap.com">
    <SAP:Websphere/>
  </SAP:order>

  <order xmlns="http://www.intel.com">
    <Chipmultiprocessor/>
  </order>
```

XML Schema DTD ist als Definition von XML Dokumenten unbefriedigend, da keine Hierarchie möglich ist und nur primitive Typen möglich sind. Aus dieser Motivation wurde XML Schema entwickelt. Besonderheit: ein XML Schema ist selbst ein XML Dokument (DTD nicht).

- Ein XML Schema wird aufgebaut mit Elementen des Namensraums <http://www.w3.org/2001/XMLSchema>.
- Es nennt den Zielnamensraum, dem targetNamespace.
- In einem XML Dokument wird das Schema nur für diejenigen Elemente zur Validierung angewandt, die innerhalb des Zielnamensraums definiert sind. Andere werden nicht validiert.

Soll ein XML Dokument für das Schema gültig sein, so müssen die Elemente und Attribute im Zielnamensraum die Eigenschaften aufweisen, die im Schemadokument vorgeschrieben sind.

Beispiel für XML Schema (für erstes Beispiel)

```
<xsd:complexType name="PersonTyp">
  <xsd:attribute name="Bezeichner" type="xsd:string"/>
  <xsd:attribute name="EPost" type="xsd:string" use="optional"/>
  <xsd:sequence>
    <xsd:element name="Name" type="xsd:string" default="Max Mustermann"/>
    <xsd:element name="Telefon" type="xsd:string" minOccurs="0"/>
    <xsd:element name="Postleitzahl" type="xsd:string"/>

    <xsd:element name="Stadt">
      <xsd:complexType <!-- anonymer Typ -- >
        <xsd:simpleContent>
          <xsd:extension base="xsd:string">
            <xsd:attribute name="PLZ" type="xsd:int" use="required"/>
          </xsd:extension>
        </xsd:simpleContent>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="Land" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
```

Zusammenfassung

- XML gut geeignet für strukturierte Daten
- leicht maschinell zu erzeugen und zu verarbeiten
- leicht mit Texteditor zu erzeugen und zu bearbeiten
- Spezialsprachen können definiert werden (z.B. WardML, MATHML, XHTML, ...)