E. Softwaretechnologie

Autor: Markus Möller

- Bezug: Vorlesung Softwaretechnologie bei Herrn Szwillus und die dort angegebene Referenzliteratur.
- Dieser Extrakt entstand als Vorbereitung auf meine Diplomprüfung (Teil: Praktische Informatik). Er faßt einige Themen einfach zusammen und mag etwas unorthodox erscheinen.
- Erstellt auf Apple Macintosh.

1. Anforderungsbeschreibung

Der Hauptproblempunkt der Definitionsphase liegt in der Kommunikation zwischen Entwickler und Auftraggeber beim Erstellen der Anforderungsbeschreibung und ihrer Umsetzung in eine Spezifikation.

Die Anforderungsbeschreibung begrenzt den Lösungsraum für ein Software-Entwicklungsproblem.

1.1 Erstellung und Probleme

1.1.1 Wer soll den Inhalt bestimmen?

- Zukünftiges Betreiberpersonal
- Zukünftige Benutzer
- Entscheidungsträger beim Auftraggeber
- Systemdesigner, Systemanalytiker, Entwickler des Softwarehauses

1.1.2 Was tun mit „unerwünschten“ Details?

Umformen von technischen Details, Lösungsideen in „eigentlich gemeinte“ Anforderungen. Manche Qualitätsanforderung oder Managemententscheidung sind hinter Implementationsvorschriften versteckt. Personalentscheidungen und „versteckte Anforderungen“ erfordern viel Fingerspitzengefühl.

1.1.3 Was sollte drin stehen?

- Einsatzbereich
- Ziele und erwartete Vorteile
- Umgebung
- Funktionalität
- Konzeptuelles Modell
  - abstraktes System
  - wesentliche Leistungen
  - Datenflußdiagramm
- Qualitätskriterien

Was sollte nicht drin stehen?

- Lösungsvorschläge
- unklare Bewertungen/Abgrenzungen
- Implementationsdetails
- Mehrdeutigkeiten
- „Sonstiges“

Aufgeschrieben werden sollte die AB vollständig, konsistent, nicht widersprüchlich und leicht änderbar. Reich strukturiert einschließlich Glossar und Index. Folglich ist das Hauptproblem der AB die Sprache!

1.2 Die Sprache

Im Idealfall kenn der Entwickler den Problembereich oder der Auftraggeber kennt sich mit Software aus. Das ist aber die Ausnahem.

Kleinster gemeinsamer Nenner wäre also die Umgangssprache. Diese ist jedoch unpräzise und mehrdeutig. Man kann es aber mit diszipliniertem Einsatz von Ugs. versuchen. Ggf. zusätzlicher Einsatz geeigneter, semi-formaler Notationen (Diagramme, Tabellen, Graphen, Konzeptuelles Modell).

Alternative zur Umgangssprache: Formale Notation von Anforderungen. Problem: Notation aus dem Problembereich meist nicht gegeben, so daß der Entwickler diese Notation erlernen muß. Softwareorientierte Notation (Formale Spezifikation) müßte vom Auftraggeber erlernt werden.

2. PSL

Problem Statement Language

2.1 Grundprinzip

Beschreibung eines Systems durch Objekte und Relationen zwischen Objekten (aber keine im Sinne von OO) die von verschiedenen vorgegebenen Typen sind.

Betrachtung von:

- Datenfluß
- Hierachischen Verfeinerungen
- (Kontrollstrukturen)
- Datenstrukturen
- Projektdaten
- Doku
- externe Ereignisse

2.2 Datenfluß

Objekttypen (Relationstypen):

- interface (generates input, is responsible for set, receives output)
- input (generated by interface, received by process)
- process (receives input, generates output, updates set)
- output (generated/received by process)
- set (is updated by process, responsible interface is)

Darstellung als Datenflußmodell: Objekte als Kreise, Beziehungen als entsprechend beschriftete Pfeile.

Die Objekte werden im wesentlichen als Substantive aus einer umgangssprachlichen Beschreibung extrahiert. Die Relationen entsprechen im Prinzip den Verben im Text. Welches Objekt ein Process, Interface etc. ist ergibt sich von selbst.

Dann werden jeweils alle möglichen Beziehungen pro Objekt gebildet.

Als nächstes ist eine hierachische Verfeinerung möglich: „subparts are“. Zulässig bei allen „Objekten“.

Prozesse können anderen aufrufen: „utilizes“.

Hierachische Verfeinerung von Daten: element (atomar), group (Menge aus Elementen oder Gruppen), entity („record“). set consists of …. element values is Wert.

„…relation to…via…“, „…associated data…“, „…maintains…“ u.a. zur Modellierung von Systemdynamik mit events und conditions mit while etc.

PSL ist unzureichend, weil es zu geringe Ausdruckskraft bei Datenstrukturen und bei Kontrollstrukturen hat. Kaum Berücksichtigung von Benutzungsschnittstellen-Aspekten.

PSL ist weiterentwickelt worden zu RSL. Genauere Modellierung des Systemverhaltens mit R-Netzen. Aufbau des Modells als kommunizierende Prozesse. Explizite Modellierung von Iteration und Auswahl. Echtzeitanforderungen formulierbar.

3. SADT

Graphische Spezifikationssprache Structured Analysis and Design Technique.

>>Ziel: Dekomposition in handhabbare, überschaubare Einheiten. Dokumentation ihrer gegenseitigen Beziehungen.

Modellierung von Dingen und Geschehnissen. Zwei Betrachtungsweisen. Datenmodell und Aktivitätenmodell.

Kästen mit vier Pfeilen: Aktivität mit Eingabedaten, Kontrolle, Ausgabedaten und unterstützendem Mechanismus.

Beim Datenmodell dasselbe: Daten mit erzeugende Operation, steuernde Aktivität, verwendende Operation und unterstützender Mechanismus.

Nicht modelliert werden Kontrollfluß und Synchronisation (Zeit).

Verfeinerungen: Teilaktivitäten und Teildaten. Verfeinerung des einen Modells veranlaßt Verfeinerung des anderen, legt aber nicht automatisch fest wie.

Verfeinerungen stellen einen Zoom dar. Folglich ist jedes SADT-Modell ein einziges Diagramm.

Einmalig ist hier die Unterscheidung zwischen Kontrolle und Eingabe.

Es ist nicht möglich, ein SADT-Diagramm mehrfach aus verschiedenen Diagrammen „aufzurufen“. Daher ungeeignet für Modulstruktur/Programm-Entwurf. Nur als Anforderungsbeschreibung und oberste Entwurfsphasen.

4. Structured Analysis (SA)

Datenflußorientierte Spezifikation

Wird als Standard in vielen CASE-Tools verwendet. Gewisse Verbreitung in der Software-Industrie.

>>Ziel: Anbieten eines Systemmodells vor Erstellung des eigentlichen Systems mit graphischen Mitteln, die – im Vergleich zur verbalen funktionalen Spezifikation – Ersatz sind für umgangssprachliche Anteile.

Besteht aus:

- Hierachisch gegliederte Datenflußdiagramme (data flow diagram)
- Datenbeschreibungen in einem „data dictionary“
- Prozeßbeschreibungen
- Beschreibungen unmittelbaren Zugriffs (immediate-access diagram)

4.1 Logische Datenflußdiagramme

Die (wesentlichen) Bildelemente – jeweils natürlich mit Name beschriftet:

- Quelle oder Ziel von Daten: doppelte Quadrate
- Datenfluß: Pfeil
- Prozeß (zur Transformation von Datenfluß): rundes Rechteck
- Datenspeicher: Rechteck mit offener rechter Seite

Logische Ebene, nicht physische Repräsentation.

Prozeßelemente können verfeinert (zerlegt in Teilprozesse) oder erweitert (ergänzt um Prozesse) werden. Dabei werden auch die Datenflüsse und die Datenspeicher meistens automatisch zerlegt in neue – feinere – Datenflüsse bzw. -speicher.

Ein Prozeß kann auch in einem separaten Diagramm verfeinert werden. Sozusagen ein Zoom in das runde Rechteck. Je nach Zoomtiefe kann man verschiedene Automatisierungsgrenzen betrachten: Von der Komplettlösung bis runter zur Automatisierung von „Atomprozessen“.

Jeder Datenfluß trägt eine Beschriftung (s.o.). Anfangs mit möglichst präziser Umgangssprache, später einen Verweis auf das „data dictionary“. Es sei denn, der Inhalt ist selbstverständlich!

Jeder Datenfluß ist gerichtet. Pfeil in eine Richtung. Es sei denn, der Datenfluß tritt paarweise auf.

Der Datenfluß wird zeitunabhängig betrachtet. Man kann aber den Fluß (das Eintreten) von Ereignissen (gestrichelter Pfeil) und einen kontinuierlichen Datenfluß (doppelte Pfeilspitze) als solchen kennzeichnen.

Prozesse und Speicher werden im oberen Bereich numeriert. Bei Prozessen notiert man den „Ort“ (Abteilung, Programmname) der Ausführung im unteren Bereich.

Suchargumente in Speichern werden unter dem Pfeil aufgeschrieben.

Bei Verfeinerungen müssen die Verbindungen zwischen Speichern und externen Einheiten übereinstimmen. Aber in Fehlerfällen werden zusätzliche Verbindungen auf tieferer Verfeinerungsebene notwendig. Diese werden mit einem Kreuz auf der (Verfeinerungs-)Grenzüberschreitungsstelle markiert.

Datenflußdiagramme zeigen den Fluß von Daten. Aber zugehöriger Materialfluß ist teilweise auch wichtig beim Modellieren. Daher wird Materialfluß mit Doppelpfeilen und Materialspeicher mit Doppellinien eingeführt.

4.2 Data Dictionary

Hierachische Beschreibung der beteiligten Daten.

4.2.1 Strukturprinzipien:

- Folgen (Sequenzen): recordmäßig
- Wiederholungen (Iterationen): Datenbank mit vielen Datensätzen (records)
- Auswahlen (Alternativen, Selektionen): „alternativer record“, Feld ist entweder x oder y (geschweifte Klammer) und Feld ist optional (Eckige Klammer)

4.2.2 Datenelemente

Angabe von:

- Name
- Wertebereich
- Bedeutung der Werte (optional)
- Beschreibung des Datenelements

4.3 Beschreibung der Prozeßlogik

>>Ziel: Beschreibung der ausgeführten Transformationen in einem Prozeß

- Jeder **primitive** Prozeß erhält eine Verhaltensbeschreibung (Mini-Spec)
- Beschreibt die Transformation „Eingangsdatenflüsse“ zu „Ausgangsdatenflüsse“.
- …und nur diese, nicht die Methode (**was**, nicht **wie**)

Techniken:

- Strukturierte Umgangssprache („structured english“)
- Pseudocode
- Entscheidungsbäume („decision trees“)
- Entscheidungstabellen („decision tables“)

4.3.1 Strukturierte Umgangssprache, Pseudocode

Hierachische Strukturierung (Schachtelung) einfacher Elemente (-> strukturierte Programmierung):

- Folgen von Elementen
- eindeutige Alternativen
- eindeutige Iterationen.

Elemente:

- einfache Sätze, die Handlungen ausdrücken
- unterstreichen der Elemente, die im „data dictionary“ vorkommen

Alternative Form (Pseudocode): Verwendung von Schlüsselworten (if-then-else, repeat-until, do)

4.3.2 Entscheidungsbäume

>>Ziel: Kompakte Darstellung von geschachtelten Entscheidungen (**statt** geschachtelter if-then-else-Strukturen).

4.3.3 Entscheidungstabellen (ET)

>>Ziel: Darstellung von Entscheidungskaskaden (wie bei Entscheidungsbäumen), aber als Tabelle.

Komponenten:

- Aufstellen aller abzufragenden **Bedingungen**
- Aufstellen aller möglichen **Aktionen**
- Tabellarische Definition der **Abhängigkeiten**

4.4 Strukturierung der Datenspeicher

Eintreffende Datenflüsse als Ausgangspunkt. Kombination zu einer sinnvollen Datenstrukturierung. Berücksichtigung der ausgehenden Datenflüsse.

>>Ziel: Kompakte, nicht-redundante Datenspeicherung und effiziente Durchführbarkeit der Ausgabe-Datenflüsse.

Zwei Schritte:

1. Erkennen und Beseitigung von Redundanz durch Inspektion
2. Gute Zugriffsmechanismen durch Normalisierung

Die Redundanz wird „mit klarem Verstand“ beseitigt.

Zur Normalisierung verwendet man die Codd’schen Normalformen.

>>Ziel: Abbildung von Datenstrukturen auf Relationen (relationale Datenbanken) vor allem **Entfernung von Iterationen**.

Technik: Aufspalten einer komplexeren Struktur in **mehrere** Relationen.

> Codd’s erste Normalform (1NF): Datenstruktur ohne Iterationen. („normalisiert“).

Verknüpfung der Relationstupel durch Schlüsselfelder. Kennzeichnet jedes Tupel eindeutig, möglichst wenige Datenfelder und immer definiert: so sollte der key sein.

> Codd' zweite Normalform (2NF): Datenstruktur in 1NF und alle Nicht-Schlüssel sind voll funktional abhängig vom Schlüssel.

Alle Datenfelder eines „record" müssen von allen Schlüsselfeldern abhängig sein. Es darf nicht vorkommen, daß einige Nicht-Schlüssel nur von einem Teil der Schlüsselfelder abhängen.

Eventuell gibt es noch Nicht-Schlüssel, die von anderen Nicht-Schlüsseln abhängig sind. Dafür nimmt man eine eigene Relation bzw. läßt das redundante Feld weg.

> Codd's dritte Normalform (3NF): Datenstruktur ist in 2NF und kein Nicht-Schlüssel ist funktional abhängig von anderen Nicht-Schlüsseln.

Warum möglichst dritte Normalform?

- Charakterisiert die elementaren unterliegenden Datenstrukturen (ohne Redundanz)
- Direkte Implementierungsgrundlage
- Einfache (kleine) Tabellen für die Datenrepräsentation

4.5 Definition des Zugriffs

Unterscheidung von **operationalem** und **informierendem** Zugriff.

- operational: Operationen (Prozesse) greifen auf Daten zu
- informierend: Benutzerzugriff auf Datenspeicher unabhängig von Systemoperationen (zur **Information** der Entscheidungsträger innerhalb der Organisation)

Problem: Sehr variierende Art, Detailiertheit, Zeitpunkt und Geschwindigkeit des informierenden Zugriff.

Zwei Arten der Unterstützung: **Indizierung** und **Satzzuordnungen**.

Darstellung in einem „data immediate access diagram" (DIAD)

1. Indizierung von Datenstrukturen (index):
   - möglich für **alle** Felder, nicht nur Schlüssel
   - schneller Zugriff auf Sätze mit entsprechendem Feldwert
2. Satzzuordnungen:

- Gegeben ein Satz von Relation A; liefere alle **zugehörigen** Sätze der Relation B
- Zugehörigkeit muß im einzelnen definiert werden.

DIAD:

- Indizierung:
  Record wird untereinander aufgeschrieben elementweise. Oben der Titel und das Indexfeld in einem Rahmen. Sekundärindex besteht nur aus Titel (wie oben) und dem Sekundäreindexfeld. Beides wieder gerahmt mit Pfeil auf den record.
- Satzzuordnung:
  Verbinden der zugeordneten records mit ihrem Bezugsrecord (andere Datenbank)

5. Realzeit-System-Spezifik. (RT)

Echtzeitsysteme

- Umgebung ausgestattet mit Sensoren (Sinne)
- und agierenden Geräten (Handlungen)
- Parallelverarbeitung mehrerer Eingaben (kontinuierlich, gleichzeitig, zeitkritisch)
- Prozeßsteuerung und -überwachung typischerweise

Erweiterung der Notation/Methode der structured Analysis um Modellierung von Zeit.

Erweiterung des SA-Konzeptes zum RT-Konzept:

- Bei den Datenflußdiagrammen
  - Ereignisflüsse und kontinuierliche Datenflüsse
  - Ereignisse: Freigabe, Sperren, Auslöser
  - Steuertransformationen
  - Ereignisspeicher
- Bei den Prozeßbeschreibungen
  - Zustandstransformation

5.1 Kontinuierliche Datenflüsse

Zeitkontinuierlicher Datenfluß, dargestellt durch Pfeil mit Doppelspitze und Beschriftung.

Kontinuierliche Datentransformation: Doppelpfeil in Prozeß, Doppelpfeil raus.

Kann auch mit diskretem Datenfluß kombiniert werden.

Diskreter Datenfluß=Inhalt + Ereignis

5.2 Ereignisse

Ereignisfluß: Ereignis=diskreter Datenfluß „ohne“ Inhalt. Dargestellt durch gestrichelten Pfeil. Z.B. „Zug fährt über Sensor“. Kombination von Ereignis- und Datenfluß: Umschalten eines kontinuierlichen Addierers auf Subtrahieren.

Spezielle Ereignisflüsse:

- Start (Freigabe): Aktivieren (Einschalten) einer Transformation
- Stop (Sperren): Deaktivieren (Abschalten) einer Transformation
- Trigger (Auslöser): Kurzzeitiges Ein- und wieder Ausschalten einer Transformation

Entstehung von Ereignisflüssen im wesentlichen in…

5.3 Steuertransformationen

Das sind **ausschließlich** an Ereignisflüsse angeschlossene Transformationen. (**Keine** Datentransformation, da keine Daten gegeben.) Darstellung als gestrichelter Prozeß.

Spezifikation des Verhaltens von Steuertransformationen mit endlichen Automaten (Transitionsnetzwerke, Mealy-Automat->Rechnerarchitektur!)

Idee: Diskrete Zeit, „nächster“ Zeitpunkt definiert. A ist zu jedem Zeitpunkt in genau einem Zustand s aus S. Anfangs im Startzustand. Zu jedem Zeitpunkt erfolgt genau eine Eingabe x aus X. A gibt dann das Ausgabezeichen a(s,x) aus und nimmt zum nächsten Zeitpunkt den Zustand ü(s,x) an, Nachfolgezustand.

Graphische Notation mit Kreisen als Zustände, Pfeile als Übergänge mit Beschriftung: oben Eingabe x, unten Ausgabe a. Auf den Startzustand zeigt ein einfacher Pfeil aus dem Nichts. Weglassen von zustandserhaltenden Übergängen.

5.4 Ereignisspeicher

Dargestellt als gestrichelter Datenspeicher.

Funktion: Warteschlange von Ereignissen

- Verzögerung kein Verlust von Ereignissen
- Spezielle Ereignisse:
  - Rücksetzen eines Ereignisspeichers
  - Warten auf einen Ereignisspeicher

6. Object-Oriented Analysis

6.1 Das Paradigma der Objektorientierung (Booch)[1]

6.1.1 Das Objektmodell

Konzeptionelle Basis der Objektorientierung:

1. Abstraktion
   Vereinfachende Betrachtung, die für den Betrachter relevante Aspekte betont. Konzentration auf Ähnlichkeiten. Unabhängig von der Realisierung.
   - Abstrakte Betrachtung des Verhaltens
   - Abstraktion von der Implementation
   - Objekte bieten sich gegenseitig diese abstrakte Sicht
2. Einkapselung
   Unsichtbarmachen von Implementationsdetails. Kein Teil eines Systems sollte von Interna eines anderen Teils abhängig sein.
   - Verstecken von Repräsentation und Implementation des Verhaltens
3. Modularität
   Gehört zum Entwurf, weniger zur Analyse. Module dienen als physikalische Container, in die man Klassen und Objekte des logischen Entwurfs (Analyse) hineindeklariert.

1. Soll allgemein sein, ist aber aus „Booch“ abgeschrieben.

Modularität ist die Eigenschaft eines Systems, das in eine Menge von kohärenten und lose gekoppelten Modulen zerlegt wurde.

4. Hierachie
   Ordnen von Abstraktionen. Wesentliche Prinzipien:
   - „kind of“-Beziehungen: „is a“ (Verallgemeinerung/Spezialisierung, Vererbung)
   - „part of“-Beziehungen: „has a“ (Aggregation)

   Spezifikation von Spezialisierungen durch Vererbung (inheritance). Weitergabe von Repräsentation und Implementation von Verhalten. Spezifikation von Aggregation wie üblich (Verbund).
5. Typisierung
   …ist der Zwang, Objekte in verschiedene Klassen einzuteilen. Bei der Verwendung dürfen Objekte verschiedener Typen gar nicht oder nur eingeschränkt miteinander kombiniert werden.
   Beschreibung von Klassenhierachien, Instantiierung von Objekten einer Klasse (Vererbung der Klasseneigenschaften auf die Objekte), statische (Compilezeit) oder dynamische (zur Laufzeit) Zuordnung von Eigenschaften.
6. Nebenläufigkeit
   Verteilung der Aktivität auf eines oder mehrere Objekte. Nebenläufigkeit erlaubt verschiedenen Objekten gleichzeitig zu agieren.
7. Persistenz
   Bewahrung von Zustand und Klasse eines Objektes über längere Zeit und/oder Raum (Adreßraum). Also Unabhängigkeit seiner Existenz vom Zeitverlauf und räumlicher Position.

6.1.2 Eigenschaften von Objekten

>>Definition (Objekt): Ein Objekt hat einen Zustand, ein Verhalten und eine Identität. Struktur und Verhalten ähnlicher Objekte werden in der gemeinsamen Klasse definiert; die Objekte sind Instanzen dieser Klasse.

>>Objektmodell ist (Software-) Modell eine „real-world“-Objekts.

1. Der **Zustand** eines Objektes beinhaltet die Eigenschaften und deren aktuellen Werte. Attribute sind Eigenschaften. Der Zustand ist eingekapselt, versteckt.
2. **Verhalten** sind Aktionen und Reaktionen von Objekten in Form von Zustandsänderungen und gegenseitigem Aufruf der Aktionen. Verhaltensbeschreibung eines Objektes=Definition der Methoden.
3. **Identität** ist die Eigenschaft eines Objektes, die es von allen anderen Objekten unterscheidet.

Methoden von Objekten:

- Konstruktor
- Destruktor
- Modifikator (Zustandsänderung)
- Selektor (Reader)
- Iterator (Zugriff auf alle oder einige Teile eines Objekts in definierter Reihenfolge)

6.1.3 Beziehungen zwischen Objekten

Es gibt zwei Arten von Beziehungen:

- Gebrauch („uses“)
- Enthaltensein („containment“)

6.1.3.1 Gebrauch

Gegenseitiger Gebrauch ist Gegenstand von Verträgen („contracts“): Ein Objekt (initiator) verwendet/ruft auf (invokes, action) einen Dienst (a service) eines anderen Objekts (of a participiant).

Wenn ein Objekt eine Nachricht zu einem anderen über einen link schickt, dann sagt man, diese zwei Objekte sind synchronisiert.

Ein aktives Objekt ändert seinen Zustand auch von allein. Ein passives nur durch Anstoß von außen. Da aktive Objekte –per Definition– sich selbst kontrollieren, gibt es keine Probleme, wenn mehrere andere aktive Objekte mit ihnen kommunizieren wollen.

Anders sieht es aus, wenn aktive Objekte auf ein passives zugreifen: Wie wird die Kommunikation dort geregelt? Die Semantik des passiven Objekts muß gewahrt bleiben! Dazu gibt es drei Möglichkeiten der

>>Synchronisation zwischen initiator und participiant: Participiant arbeitet

- sequentiell: Funktioniert nur richtig, wenn es immer nur ein einzelnes aktives Objekt gibt; message-passing = Unterprogrammaufruf.
- blocking/guarded: Mehrere aktive möglich , funktioniert aber nur richtig, wenn die aktiven untereinander sicherstellen, daß sie sich gegenseitig ausschließen.
- concurrent/synchronous: Mehrere möglich und der passive garantiert gegenseitigen Ausschluß.

6.1.3.2 Enthaltensein

Wo links (Gebrauch) eine peer-to-peer-Beziehung oder client/supplier-Beziehung bezeichnen, bedeutet aggregation (Enthaltensein) eine Gesamtheit/Teil- (whole/ part-) Hierachie – part-of-Beziehung; mit der Möglichkeit, vom aggregate zu seinen parts (bzw. attributes) zu navigieren. Umgekehrt nur, wenn die Kenntnis über den container ein Teil des Zustandes von Part ist.

Aggregation bedeutet nicht unbedingt physikalisches Enthaltensein. Ein Aktionär besitzt allein seine Aktien, aber sie sind kein physikalischer Teil von ihm.

>>Trade-offs zwischen links und aggregation. Aggregation ist manchmal besser, weil es parts als secrets des whole einkapselt /versteckt. Links sind manchmal besser, weil sie eine losere zwischen Objekten erlauben.

Ein Objekt, das ein attribute eines anderen ist, hat automatisch auch einen link zu seinem aggregate. Über diesen link kann das aggregate Nachrichten zu seinen parts senden.

6.1.4 Eigenschaften von Klassen

>>Definition (Klasse): Eine Menge von Objekten mit gleicher Struktur und gleichem Verhalten.

Ein einzelnes Objekt ist eine Instanz der Klasse. Ein Objekt ist keine Klasse. Eine Klasse kann wieder als Objekt betrachtet werden.

Während ein individuelles Objekt eine konkrete Einheit ist, die eine Rolle im Gesamtsystem erfüllt, deckt eine Klasse die Struktur und das Verhalten ab, was allen dazugehörigen Objekten gemeinsam ist. Klassen definieren ihre attributes und methods genau wie Objekte.

>>Sichtbarkeit Es gibt eine Trennung in Schnittstelle (interface) und Realisierung (implementation). Vorzeigen von Deklarationen von Operationen und Teilen der Strukturbeschreibung in der Schnittstelle. Verbergen von Teilen der Strukturbeschreibung und der Implementation des Verhaltens in der Realisierung. Die Definition der Attribute und Methoden einer Klasse erfolgt also nicht einfach so, wie bei Objekten, sondern wird in zwei Unter„records“ gesplittet: „interface“ und „implementation“. Die Implementation-Attribute umfassen auch die aus dem Interface,

die Implementation-Methoden implementieren (mit procedure) u.a. die Methoden aus dem Interface.

Das Interface einer Klasse kann weiter unterteilt werden in:

- public: eine Deklaration, die *für alle* „clients“ zugreifbar ist.
- protected: eine Deklaration, die nur *für* die Klasse selbst, ihre *Unterklassen* und ihre Freunde zugreifbar ist.
- private: eine Deklaration, die *nur für die Klasse selbst* und ihre Freunde zugreifbar ist.

„Freunde“ brechen die Einkapselung einer Klasse und müssen so, wie im Leben, sorgfältig ausgewählt werden.

>>Klassifikation o.a. Klassenbildung ist das Aufspüren von Gemeinsamkeiten, Zusammenfassen in einer Klasse (Abstraktion): schwer!

Klassifizierungstechniken:

- Kategorisierung (klassisch)
  - Finden klar feststellbarer Eigenschaften zum Trennen oder Zusammenfassen von Objektgruppen
  - Ideal: „orthogonale“ Eigenschaften, d.h. sie sind unabhängig voneinander.
  - Bedeutung jeder Kategorisierung ist „subjektiv“ aus der Sicht des Betrachters.
- Konzeptuelles Gruppieren (moderner)
  - Beschreibung eines Konzepts
  - Anlegen eines Maßstabs an einen konkreten Text im Vergleich zu der Beschreibung
  - Unscharfes probabilistisches Maß
- Prototypen (am neuesten/wenn die anderen nicht helfen)
  - Repräsentation einer Klasse durch ein (prototypisches) Objekt, Argumentation über Beispiele
  - Entscheidung über Klassenzugehörigkeit nach Ähnlichkeiten

6.1.5 Beziehungen zwischen Klassen

6.1.5.1 Arten von Klassenbeziehungen

- Generalisierung, Spezialisierung
  is a, kind of
- Aggregation
  has a, part of
- Assoziation
  semantische Verbindung

6.1.5.2 Definition von Klassenbeziehungen

Verwendete Techniken:

- Vererbung
- Verwendung (uses)
- Instantiierung
- Metaklassen

Vererbung

- Ausdruck von Generalisation und Assoziation
- Übernahme von gemeinsamen Anteilen von der Oberklasse
- Spezialisieren von (einigen) Anteilen in der Unterklasse
- Kombination von semantisch in Beziehung stehenden Klassen in einer neuen Klasse.

Definition einer Klasse als Unterklasse einer anderen, dadurch Übernahme (Erben) von Struktur und Verhalten der Oberklasse, Modifizieren (Spezialisieren) von Teilen dieser Struktur oder des Verhaltens, Hinzufügen spezieller Struktur- oder Verhaltenselemente.

Polymorphie: Wenn eine geerbte Methode verändert wird, dann ist der Aufruf dieser Methode offensichtlich abhängig von der Klasse des angesprochenen Objektes.

Wenn eine Klasse als Unterklasse mehrerer Oberklassen definiert wird, dann ist das mehrfache Vererbung. Problematisch dabei ist, wenn von verschiedenen Oberklassen Methoden mit gleichem Namen geerbt werden.

Verwendung

Gebrauch (der Schnittstelle) einer Klasse zur Schnittstelle oder zur Implementation einer anderen Klasse ohne daß die eine Klasse eine Spezialisierung der anderen ist:

- Die Schnittstelle einer Klasse verwendet eine andere Klasse (open use), sozusagen als Parameter einer Methode.
- Die Implementation einer Klasse verwendet eine andere Klasse (hidden use)

Kardinalität der Gebrauchsbeziehung z.B. 1:n. Eine Klasse verwendet n Objekte einer anderen.

Man sollte multiple Vererbung nicht verwenden, um einen Gebrauch auszudrükken! Empfehlung: Gebrauch, wenn Klasse mehr ist als die Summe ihrer Teile. Vererbung, wenn Spezialisierung oder gleich der Summe seiner Teile.

Instanziierung

Instanziierung einer Klasse als spezielle Version einer strukturgebenden Klasse (container classes, generic classes, parametric classes, genericity) abhängig von Parameter(klasse)n. Z.B. Menge_von(Typ).

Metaklassen

Manipulieren von Klassen durch Metaklassen analog zur Manipulation von Objekten durch Klassen. Klassen von Klassen. Eine Metaklasse ist ein Klasse, deren Instanzen wieder Klassen sind.

6.2 Objektorientierte Software-Entwicklung

> Objektorientierung ist ein Paradigma, das den Softwareprozeß von der Analyse bis zum Programm abdeckt.

>>Programmierung: Objektorientierte Programmierung (OOP) ist eine Implementationsmethode, die ein Programm auffaßt als kooperierende Sammlung von Objekten. Diese sind Instanzen von Klassen; die Klassen sind Elemente einer Vererbungshierachie.

>>Entwurf: Objektorientierter Entwurf (OOD) hat eine objektorientierte Zerlegung des angestrebten Systems zum Ziel. OOD-Methoden umfassen eine Notation für

die Modellierung der logischen und physischen und der statischen und dynamischen Aspekte. (Logisch: Klassen und Objekte; physische: Module und Prozesse)

Klassische Entwurfstechniken haben eine Zerlegung in Funktionen zum Ziel.

>>Analyse: Objektorientierte Analyse (OOA) analysiert die Anforderungen an ein System aus der Sicht der Klassen und Objekte, die im Problembereich verwendet werden.

Klassische Analysetechniken basieren auf einer Datenflußsicht auf das System.

Fazit:

> Ziel objektorientierter Analyse ist das Aufstellen einer Hierachie von Klassen und Objekten, die das System modellieren und Aufbau der kooperativen Strukturen, die das notwendige Verhalten erlauben.

Objekte und Klassen = wesentliche Abstraktionen.
Kooperative Strukturen = Mechanismen.

Konzentration auf die Sicht von außen = Schnittstellen
(nicht Repräsentation und/oder Implementation)

6.3 Objektorientierte Analyse nach Booch

Komponenten: Zwei Diagrammarten…

- Klassendiagramme
  - Klassenkategoriendiagramme
  - Zustandsüberführungsdiagramme
- Objektdiagramme (Timing-Diagramme)

…mit zugeordneten Textschemata (templates) für

- Klassen
- Methoden (operations)
- Zustände
- Objekte
- Meldungen (messages)

6.3.1 Klassenstruktur

6.3.1.1 Klassenbeziehungen

Ziel: Graphische Mittel zum Aufzeigen der Existenz von Klassen und der Anordnung in der Klassenstruktur (Vererbung, Gebrauch, Instanziierung).

Bildelemente:

- Klasse: gestrichelte Wolke
- verwendet (offen, in der Schnittstelle): offener Punkt mit Doppellinie
- verwendet (hidden, zur Implementation): schwarzer Punkt mit Doppellinie
- instanziiert (kompatibel): gestrichelter Pfeil
- instanziiert (neuer Typ): gestrichelter Pfeil mit Querstrich
- erbt von (kompatibel): Pfeil
- erbt von (neuer Typ): Pfeil mit Querstrich
- semantische Verbindung: Strich

Kardinalitätswerte an „verwendet“-Kanten:

- 0
- 1
- *: unbekannte Anzahl
- +: unbekannt, ≥1
- ?: 0 oder 1
- n

6.3.1.2 Klassenbeschreibungen (class templates)

Textuelles Mittel zum Beschreiben der Klassendetails. (Maximal-)Rahmen für eine vollständige Dokumentation. Nicht immer sind alle Felder notwendig! Wesentlich: Fields, Operations der Schnittstelle.

6.3.1.3 Klassenkategorien

Graphisches Mittel zur Strukturierung großer Klassendiagramme in logisch zusammenhängende Klassengruppen. Abstraktion von Klassendiagrammen auf eine höhere Ebene zwecks Übersichtlichkeit.

Bildelemente:

- Klassenkategorie: Rechteck. Jedes Rechteck enthält ein Klassen(kategorien)diagramm.
- Sichtbarkeit: Pfeil (Quelle sieht Spitze), Klassen(kategorien) –Inhalt– von Ziel ist in Quelle sichtbar.

Da eine Klassenkategorie einen eingekapselten Namensraum darstellt, können einige der Einheiten darin sichtbar außerhalb der Kategorie sein, manche privat zur Kategorie und wieder andere können importiert sein von anderen sichtbaren Kategorien:

- unverziert: privat und darf nicht referenziert werden von außen
- gerahmt: enthalten und exportiert, d.h. sichtbar in Kategorien, die diese Klasse importieren
- unterstrichen: wurde importiert

Der Vermerk „global“ bedeutet vereinfacht die Sichtbarkeit für alle Klassen zur Vermeidung von zu vielen Pfeilen.

6.3.2 Klassendynamik

6.3.2.1 Methoden, Operationen

Textuelles Mittel zum Beschreiben der Methodendetails. Wiederum: (Maximal-) Rahmen für eine vollständige Dokumentation. Nicht immer alle Felder nötig. Oft sind statt dieses Schemas sinnreiche Namen ausreichend.

6.3.2.2 Zustandsüberführungsdiagramme mit Textschemata

Graphische und textuelle Repräsentation der Zustandsübergänge (der Objekte) einer Klasse.

Bildelemente (wie bei endlichen Automaten üblich):

- Kreis: Zustand
- Pfeil: Übergang
- Markierung am Pfeil: eingetretenes Ereignis / produzierte Eingabe
- Doppelkreis: Startzustand
- Fettkreis: Endzustand

Textelemente –für jeden Zustand (wenn nötig)–:

- Events: Liste von Bezeichnern
- Docu: Text
- Action: PDL, Verweis auf Objektdiagramm.

6.3.3 Kooperationen der Objektwelt

Beschreibung der Kooperation der Objekte. Wer schickt wem wann welche Meldungen? Konsistent mit den Klassenbeschreibungen.

6.3.3.1 Objektbeziehungen

Graphische Repräsentation der wesentlichen Mechanismen (key mechanisms, contracts). Schnappschußartige Situationsdarstellungen.

Bildelemente:

- Objekt: Wolke mit durchgezogener Linie.
- Nachrichtenaustausch zwischen Objekten: ungerichtete Linie
  - innerhalb des Softwaresystems: durchgezogen
  - externe Verbindung: gestrichelt

6.3.3.2 Objektbeschreibungen

Textuelle Festlegung von Objektdetails, falls nötig und sinnvoll. Aufbau: Name, Doku, Klasse (Typ des Objekts, Klassenzugehörigkeit), Persistenz (immer/persistant, während des ganzen Programms/statisch, kreiert und zerstört mehrmals während Laufzeit/dynamisch). Klasse legt schon die meisten Eigenschaften fest. Persistence muß zur Klasse passen.

6.3.3.3 Beschreibung von Meldungen

Textuelle Festlegung von Details einer Meldung, falls nötig und sinnvoll.

6.3.3.4 Zeitbeziehungen (timing diagrams)

Graphische Beschreibung von komplexeren zeitlichen Abhängigkeiten zwischen Operationen.

Methoden: Durchnumerieren der Meldungen, Beschreibung durch Pseudocode, Structured English (PDL).

6.4 Objektorientierte Analyse nach Rubin & Goldberg (Object Behavior Analysis,OBA)

Die Analyse ist das *Untersuchen und Modellieren* eines Problembereichs *bzgl. festgelegter Ziele.* Sie konzentriert sich auf Annahmen, **was** das System tut, nicht wie es es tut. Außerdem muß sie nachvollziehbar sein, so daß man die Ergebnisse begründen kann.

Die Objekt-orientierte Analyse modelliert eine Situation durch miteinander handelnde Einheiten. Dabei ist es wichtig, die Natur der benötigten Informationsverarbeitung als Ausdruck zu erbringender Dienste zu verstehen. Danach kann festgelegt werden, welche Einheiten am besten diese Dienste ausführen.

Man braucht einen effektiven Weg, die Objekte zu finden. Dieser Ansatz untersucht zuerst das Systemverhalten, d.h., was stattfindet im System. Dann wird das Verhalten einzelnen Systemteilen zugeordnet und geschaut, wer ein Verhalten auslöst (Initiator) und wer noch daran beteiligt (Ausführende Einheit, Partizipient) ist. Initiatoren und Partizipienten, die signifikante Systemrollen spielen, werden die Objekte.

Dieser Analyseansatz wird „Object Behavior Analysis“ oder OBA genannt. Das Ergebnis besteht aus:

1. **Skripten**, die den Gebrauch des (geplanten) Systems aufzeichnen
2. **Glossaren** von:
   - Initiator-Partizipient Namen
   - Dienste von Partizipienten
   - Attribute
   - Zustandsdefinitionen
3. **Objektmodellen**
   - Hierachische Beziehungen
   - Vertragliche Beziehungen
4. **Dynamische Systemmodellen**
   - Objektlebenszyklen
   - Sequenzen von Operationen

OBA ist ein iterativer Ansatz. Wenn man jedoch bemerkt, daß eine Information fehlt, dann geht man die entsprechenden Schritte zurück.

OBA besteht aus fünf Schritten:

1. Kontext der Analyse festlegen.
2. Untersuchung des Verhaltens zum Verstehen des Problems.
3. Objekte definieren, die Verhalten aufweisen.
4. Klassifizierung der Objekte und Identifizieren von Beziehungen.
5. Systemdynamik modellieren.

6.4.1 Schritt 0, Analysekontext

Die Numerierung beginnt mit 0, weil dieser Schritt oftmals als außerhalb des traditionellen Analysebereichs angesehen wird.

1. **Ziele** identifizieren für
   Zeit, Ressourcen und Qualität des Projekts „objective“; Geschäftsziele (warum dieses Projekt, „goals“)
2. **Quellen** für die Analyse identifizieren
   Dokumente, Wörterbücher, Benutzer, Experten.
3. **Kernaktivitäten-Bereiche** identifizieren
   Basis für Scripting, Arbeitsaufteilung und parallele Entwicklung. Diese Bereiche dienen als Anfang und können im Laufe der Entwicklung des Projektes Änderungen unterliegen. (Z.B. Create, Modify, Save,…)
4. **Vorläufigen Analyseplan** erstellen
   Prioritäten über die Kernbereiche festlegen sowie Zeit- und Ressourcenschätzung für Analyseaktivität.

6.4.2 Schritt 1, Problem verstehen

1. **Szenarien planen**
   - Hauptszenarien wählen, die alle möglichen Wege durch die Systemfunktionen abdecken.
   - Szenarien Kernaktivitätsbereichen zuordnen
2. **Scripting**
   Tabellarisches Aufschreiben der Benutzungsszenarien. Sie enthält die Einträge „Initiator, Action, Participiant, Service“. Durch Action des Initiator führt Participiant Service aus. Ein „Contract“ ist ein Abkommen, daß einen Einheit den Service einer anderen Einheit benutzt (aufruft). „Service“ – zur Verfügung gestellter Dienst. „Contracted Service“ – Benutzter

Dienst von anderer Einheit. Jede Reihe der Tabelle (Script) stellt einen Contract dar. Ferner werden noch im Kopf angegeben:

- Name oder Identifier
- Autor
- Interviewpartner oder Referenzen
- Version
- Vorbedingungen
- Nachbedingungen
- Verbindung zu einem Ziel, Kernaktivität oder anderem Script. (Trace, warum dieses Skript?)

Scripts müssen verständlich sein für Interviewte und für Analytiker. Actions in Scripts können markiert sein:

- A?: noch Zusatzinfos nötig, um Analyse machen zu können.
- A>: Erklärt in anderem Script.
- D: In der Designphase aufgelöst.
- E: Außerhalb des Sytembereichs.

3. **Glossare erstellen**
   In Scripten wird möglicherweise eine neue Ausdrucksweise benutzt. Daher werden Glossare aufgestellt.
   - Party-Glossar
     definiert für jede „Party“ eine „Definition“ als Beschreibung, „Trace“ als Verweis zum Script wo die Party vorkommt, „Role“ mit Angabe ob (und-oder) Initiator/Participiant. Anschließend können einige Party-Namen generalisiert werden, was ein Aliasverzeichnis erforderlich macht.
   - Service-Glossar
     definiert für jeden Service-“Name“ eine „Definition“, alle „Participiants“, die den Service bieten, und die übliche „Trace“.
4. **Attribute erstellen**
   In einem Attribut-Glossar werden für jede Party ihre Eigenschaften in verschiedenen Dimensionen angegeben. (Für Reihe z.B. Höhe in Pixeln.) Aufgeführt werden:
   - Attributname
   - Definition
   - Contract: Vertrag zwischen Attribut und Partei?
   - Accessor: Zugriffsfunktion der Partei auf das Attribut?
   - Mutator: Veränderungsfunktion der Partei auf das Attribut?
   - Multi/Single Value: Enthält Attribut genau einen Wert oder eine Sammlung?
   - Range of Value
   - State Definition: Ist das Attribut Teil der Zustandsbeschreibung der Partei? (falls immer nein, dann ist Attribut keine Zustandsvariable)

Ein Attribut ist eine logische Eigenschaft einer Party, die mit den Anforderungen zusammenhängt, einen oder mehrere Contracts zu erfüllen. Wenn es um einen Initiator geht, dann kann angenommen werden, daß der Initiator dieses Attribut benötigt, um einen Service aufzurufen. Wenn es um einen Participiant geht, dann benötigt er es wahrscheinlich, um den Service zu erbringen. Attribute sind logische und nicht unbedingt physikalische Eigenschaften. Die Entscheidung, wie ein logisches Attribut physikalisch realisiert wird, ist eine Designaufgabe. Die Beziehung zwischen Attribut und Party ist jedoch eine Analysenaufgabe.

6.4.3 Schritt 2, Objekte definieren

Erstellen von „Object Modelling Cards“. Sie enthalten:

- Name des Objektes
- Name der Objekte, von denen Attribute und Verhalten geerbt werden. (3)
- Version (2+3)
- Informationen und Verhalten, die neu dazukommen.
  - Attribute
  - Services
- Contracted Services (3)
- Objects (die diese Dienste anbieten) (3)
- Card-Trace (3)

Welche Parties werden Analyse-Objekte? Eigentlich alle außer Initiatoren, die keine Participiants sind, denn sie haben keine Service-Interface, welches ein Objekt erst ausmacht.

6.4.4 Schritt 3, Klassifizierung der Objekte und Indentifizieren von Beziehungen

1. **Contract-Beziehungen beschreiben**
   Für jede Object-Modelling-Card wird noch ausgefüllt, welche Contracted Services, Card Traces und Vererbungen es gibt. Damit ist die Karte voll. Entspricht Austausch von Meldungen zwischen Objekten.

2. **Objekte in Hierachien organisieren**
   Hierbei wird gearbeitet mit:
   - Abstraktion: Services oder Attribute, die zwei oder mehr Objekte gemeinsam haben werden in einem neuen Objekt zusammengefaßt. Dieses vererbt dann.
   - Spezialisation: Services oder Attribute können als eine Verfeinerung von Services oder Attributen eines anderen Objekts beschrieben werden.
   - Faktorisierung: Aufteilen von Services auf separate Objekte anstatt auf eines. Für verschiedene Verantwortungsbereiche.

Um das nachvollziehbar zu machen wird eine Reorganisationstabelle erstellt, die jeweils den Reorganisationstyp, die Input-Objekte und die neuen Objekte enthält.

6.4.5 Schritt 4, Systemdynamik modellieren

Hier werden die Aspekte behandelt, die sich im Laufe der Zeit verändern. Bisher wurde das System statisch betrachtet. Der Status eines Objekts wird durch die Vor- und Nachbedingungen in Skripten bestimmt. Veränderungen des Status resultieren gewöhnlich in Veränderungen im Verhalten. (Tabelle beenden liefert Sicherndialog in Abhängigkeit vom Status „modified“).

1. **Status-Definitions-Glossar erstellen**
   für jedes Objekt, das sein Verhalten ändert bei Statusänderung wird der „Status, Definition dessen, Beschreibung, Trace“ angegeben. Diese Objektzustände entstehen aus Vor- und Nachbedingungen von Scripts.
2. **Objekt-Lebens-Zyklen bestimmen**
   Ein Zyklus bestimmt die Zustandsübergänge eines Objektes als Antwort auf Events (Event=Script). Event ist ein Ereignis oder Veränderung im System oder der Umgebung, die ein oder mehr Objekte zu einem Statuswechsel anstößt, der das Verhalten des Systems ändert. Scripts sind Gruppen von Aktivitäten, die als einzelne Events angesehen werden können.
   Evtl. können auch Scripts gesplittet werden, wenn ein interessanter Zwischenzustand besser gewürdigt werden soll.
3. **Sequenzen von Operationen bestimmen**
   Sequenzen von Operationen werden auch Kontrollfluß genannt. Also Sequenz von Operationen, die als Antwort auf einen Event passieren. Skripte sind zwar sequentiell aufgeschrieben, aber es ist auch möglich das eine andere Anordnung nötig ist. Z.B. können Skripte concurrent, wiederholend, selektiv oder optional durchlaufen werden. Es geht darum, die gewünschten möglichen Anordnungen im Skript zu notieren, damit Designer verstehen, was nötig und was optional ist. Hier wird vorgeschlagen jedem Skript ein Diagramm zu verpassen, in dem die Ordnung der Aktivitäten innerhalb des Skriptes beschrieben wird.

6.5 Fragen

6.5.1 Wie sind Szenarien miteinander verbunden?

Ein Szenario ist ein konkretes Benutzungsbeispiel. Alle Szenarien decken alle relevanten Vorgänge ab. Die Verbindung ist nur das Gesamtsystem.

6.5.2 Gehen Glossare direkt in Objekte ein?

Ja. Z.B. Objektname aus Party-Glossar (row), Attribute (style, height) aus Attribute-Glossar, Provide Services aus dem Service-Glossar (set text style to bold, resize height).

Nein. Falls ein Objekt verfeinert wird muß zurückiteriert werden zum Glossar.

6.5.3 Welche Teilschritte können automatisiert werden, welche nicht?

- automatisierbar:
  - Erstellung des **Party-Glossar** aus den Scripts. Außer „Definition“.
  - **Aliasliste** aus dem verallgemeinerten Party-Glossars.
  - Erstellung des **Services-Glossar** aus den Scripts. Außer „Definition“.
- nicht automatisierbar:
  - Der **verallgemeinerte Party-Glossar** ist kreativ und nicht automatisierbar!
  - **Object-Modelling-Cards**, weil z.B. alle Nur-Initiatoren rausfallen und bei der Reorganisation Objekte kreativ verändert werden.
  - **Attribut-Glossare** mit Hilfe des Service-Glossar (Attribute indirekt, z.B. resize-service impliziert ein Größen-Attribut).
  - **Zustands-Glossar**, da durch eventuelles Splitten von Skripts die Vor- und Nachbedingungen ändern, also die Zustände von Objekten.

6.5.4 Wie nah ist man nach OBA an Implementierung?

Ziemlich nahe.

6.5.5 Kann man die OBA Objekte wirklich so programmieren?

Denke schon.

6.5.6 Wird in einem OBA Skript der Dienst am Ende der Zeile erbracht (oder folgenden) oder zwischen der 1. und 2.?

Der Service, der von dem Initiator beim Participiant angefordert wird steht am Ende der Zeile in der vierten Spalte.

Er kann auch (in dreispaltiger Variante) als Action in der nächsten Zeile aufgeführt werden.

Ein Service ist keine Reaktion des Participiant auf eines Initiators Action. Vielmehr ist er eine Spezifikation, was für ein Interface der Participiant haben muß, um den Contract zu erfüllen. Jede Reaktion auf Seiten des Participiants würde normalerweise in der nächsten Zeile des Scripts stehen.

Z.B.: 1 erfragt Info von 2. Also muß 2 den Service „kann Info liefern“ bieten.
Dann: 2 liefert Info an 1. Also muß 1 den Service „Info annehmen“ bieten.
Durchführung des Dienstes also in der zweiten Script-Zeile.
In der vierten „Service“-Spalte wird nur das Interface von Participiant bestimmt.

6.5.6.1 Erfolgt die Kommunikation synchron oder asynchron?

Ist Interpretation. Kommt darauf an, ob man einen Trigger-Effekt oder einen Funktionsaufruf modellieren will.

6.5.7 Wo entstehen in OBA Objekte?

In Schritt 2: Einige Elemente des Party-Glossars werden Objekte. Ferner in Schritt 3.2, wo die Objekte reorganisiert werden.

6.5.8 Was ist der grundlegende Schritt bei der OBA (Was verstehen Sie unter Scripts)?

Aufstellen von Benutzungsszenarios, die alle möglichen Pfade durch das System überdecken. Dann wird für jedes Szenario ein Script erstellt, das eine Folge von Ereignissen („Trigger“, Eingaben), Informationsanforderungen und geleisteten Diensten im Rahmen eines Szenarios enthalten.

6.5.9 Was sind eigentlich Szenarien?

Ein Szenario ist ein konkretes Benutzungsbeispiel (Benutzungssituation). Alle Szenarien zusammen decken alle relevanten Vorgänge ab.

6.5.10 Welche Glossare gibt es?

- Initiator-Participiant Namen (Party-Glossar)
- Participiant-Services (Service-Glossar)
- Attribut-Glossar
- Zustands-Glossar

6.5.11 Kann der Dienst-Glossar vollständig aus den Skripten abgeleitet werden?

Außer dem Feld „Definition“ ja.

6.5.12 Schritte der OBA

- **Kontext** der Analyse
  Was ist das Ziel des Projektes? Welche Informationsquellen benutze ich? Hauptsächliches Verhalten / Grobstruktur? Erster Managementplan?
- Verstehen des Problems durch Betrachtung von **Verhalten**
  Szenarien, Scripts, Glossare
- **Objekte** definieren
  Modelling Cards
- **Klassifizieren** von Objekten, Beziehungen untereinander finden
  Contracts eintragen, Objekte in Hierachien reorganisieren
- System**dynamik** modellieren
  Zustandsglossar, Zustandsübergänge bestimmen, Scripte intern Ablauf regeln (sequentiell?)

7. Attributspezifikation

Attribute sind quantitative und qualitative Eigenschaften des Systems. –wie gut, wie schnell, wieviel Speicher/Zeit– Teilweise naheliegendes, numerisches Maß. Manchmal auch nur „vage“ Festlegbarkeit.

Bedeutung von Attributen: Die funktionale Korrektheit ist zwingend; das Einhalten von **Qualitätsmerkmalen** ist implizit genauso zwingend. Beispiele: schnelle

Vermittlungsalg. bei Telefonvermittlungssystemen, hohe Benutzerfreundlichkeit eines Textverarbeitungssystems, Ausfallsicherheit der Software für Flugzeuge.

Spezifikation von Attributen:naheliegend bei „numerischen“ Attributen, umgangssprachlich bei qualitativen Aussagen („möglichst portabel“, „sehr schnell“). Problem dabei ist, daß Zahlen ein höheres Gewicht haben für den Betrachter und dadurch quantifizierte Maße stärker beachtet werden.

>>Ziel der Attributspezifikation: Festlegen von Attributwerten **vor** Entwicklung des Systems (postulativer Charakter), Vergleich mit erreichten Werten **nach** Fertigstellung des Systems. Festlegung (**vorher**) der Beziehungen zwischen Vorgaben und Erreichtem.

Technik: Ersetzen qualitativer Angaben durch quantitative. **Alle** Attribute werden durch Zahlen festgelegt.

Prinzipien:

- Vollständigkeit:
  Erfassung aller relevanten Attribute
- Meßbarkeit:
  praktisch als Zahl ermittelbar
- Hierachie:
  Manchmal müssen Attribute als Hierachie von Attributen und Teilattributen formuliert werden.
- Verständlichkeit:
  Begriffswelt des Endbenutzers
- Trennung von Attributen und Lösungen:
  Überprüfung, in welchem Ausmaß eine Lösung Attribute erfüllen hilft oder nicht.

Methode:

1. Aufstellung einer Liste aller **Attribute**
   Textuelle Aufzählung. Nur welche, nicht ihre Werte!
2. Aufstellen einer **Attibuthierachie**
   Kreativer Schritt. Zusammenfassen von Gruppen zu „höherwertigen“ Attributen. Aufteilung zu „abstrakter“ Attribute in konkretere. Evtl. zurück zu 1. Re-Analyse des Problems und der Zielsetzung. „Warum ist dieses Attribut wichtig?“
3. Definieren von **Werteskalen**
   Meßeinheit für ein Attribut. Konkrete Zahl als Wert. Skala als Interpretation dieser Zahl. Schlüsselwort: Skala.

4. Definieren von **Tests**
   Angabe eines Verfahrens zum praktischen Ermitteln der Werte. Schlüsselwort: Test. Teilweise überflüssig, da klar.
5. Definieren des **schlechtesten** möglichen Ergebnisses
   Gerade noch akzeptabler Wert des Attributes. Nur lokal auf dieses Attribut bezogen. Schlüsselwort: Schlechtester Fall. Nicht-Erreichen dieses Wertes in einem einzigen Attribut kippt das Projekt!
6. Definieren des **geplanten** Ergebnisses
   Anfang des Erfolges. Schlüsselwort: Plan. Auslieferung erst, wenn alle Attribute Plan-Niveau haben! Definition so, daß alle gleichzeitig erreichbar. Veränderung der Planwerte nur in Absprache mit dem Auftraggeber.
7. Definieren des **bestmöglichen** Ergebnisses
   Schlüsselwort: Bester Fall. Bei gegebenen technischen Möglichkeiten bestenfalls erreichbarer Wert (Traumnote). Machbarkeitsgrenzen, Einordnung von Plan und Ist-Zustand, Motivation.
8. Festhalten des zur Zeit **gegebenen** Zustandes
   Ist-Zustand (falls möglich). Parameter eines Organisationsablaufs mit „Bleistift und Papier“. Schlüsselwort: Jetzt. Qualifizierungsmechanismus: „Jetzt“ für zwei verschiedene „Fälle“ (Konkurrenzprodukte).
9. Notation der **Autorisierungsinformation**
   Schlüsselwort: Quelle. Welche „Autorität“ legte Attribut fest. Autor oder Begründung für die Wichtigkeit o.ä.

>>Beispiel: Maximale Größe bearbeitbarer Dateien

- Skala: KByte
- Schlechtester Fall: 200 KB
- Plan: 500 KB
- Bester Fall: unbegrenzt
- Jetzt (Mikroweich Wort): 2 MB
- Jetzt (WortPerfekt): 1,2 MB
- Quelle: Abteilungsleiterbesprechung Bereich 4.2 vom 04.05.90. Wir müssen die Konkurrenz in *diesem* Punkt deutlich schlagen.

8. Gütekriterien für Entwürfe

Bewertung in Hinblick auf Kohäsion und Kopplung.

>>Kohäsion: Zusammenhang von Aktivität und Arbeitskraft innerhalb eines Moduls. Kooperation von Funktionen und Programmteilen eines Moduls. Maß für die Gemeinsamkeit von Zielen und Aufgaben.

>>Kopplung: Zusammenarbeit zwischen Modulen. Grad der Notwendigkeit des Datenaustauschs, der Synchronisation der gegenseitigen Abstimmung. Maß für die gegenseitige Abhängigkeit der Module.

Gute Entwürfe besitzen

- maximale Kohäsion
  sorgt für effektive Zusammenarbeit innerhalb der Module (kurze Kommunikationswege), Arbeitslast wird *in* den Modulen erbracht.
- und minimale Kopplung
  sorgt für seltene, wenig umfangreiche (langsame) Kommunikation *zwischen* Modulen.

Bei optimaler Kohäsion und Kopplung spielen sich Operationen auf einer Datenstruktur innerhalb des Heimatmoduls der Datenstruktur selber ab. Kommunikation nur über von Prozeduren ausgetauschten Werten. „Idealstruktur“ praktisch oft unmöglich wegen Effizienzüberlegungen: ein Modul kann mit den Aufgaben zu einer Datenstruktur überfrachtet sein. Wertzuweisung wesentlich effizienter als Prozeduraufruf. Aber das Objektparadigma verlangt kleine Module. Geringe Kopplung ist kein information hiding.

9. Jackson-Entwurf (JSD)

Durchgängige Methode von Spezifikation bis Implementation. Sehr operational, fast objektorientiert. Modellierung von Prozessen der realen Welt, „Implementation“ dieser Prozesse.

9.1 Jackson-Diagramme

>>Ziel: Kompakte Darstellung von Sequenz, Iteration, Alternative. Wichtige Konzepte für Daten und Operationen. Ergänzungen um textuelle Version.

9.1.1 Sequenzen

- Daten: Endliche Folge von Teildaten.
- Aktionen: Nacheinanderausführung.

Graphische Darstellung durch Baum. Wurzel ist das Gesamtding. Die Schicht darunter sind die Teile, aus denen es besteht (Daten oder Aktionen).

Schachtelung durch „Unterbäume“. Nix besonderes daran. Bei der Textversion entweder direkte Einfügung von Teilstrukturen oder Verweis auf externe mit „use“ bei Daten und „do“ bei Operationen.

9.1.2 Iterationen

- Daten: potentiell unendliche Folge einer Komponente.
- Aktionen: Potentiell unendlich wiederholte Ausführung einer Teilaktion.

Darstellung durch Stern in der zu wiederholenden Struktur.

9.1.3 Selektionen

- Daten: Auswahl genau einer Komponente.
- Aktionen: Ausführung genau einer ausgewählten Teilaktion.

Darstellung: Punkt in der Teilstruktur.

9.2 Methode

J(ackson)S(ystem)D(evelopment)-Spezifikation ist eine Menge von kommunizierenden nebenläufigen (endlos) Prozessen. Betriebssystem verwaltet Menge von diesen Prozessen. Mißverhältnis in der Zahl und im Zeitverhalten (lange Lebensdauer, kurze Laufzeiten). JSD-Implementation ist die Überbrückung dieses Mißverhältnisses durch Transformation.

1. **Entity-Action**-Step
   Sehr ähnlich OOA.
   - *Liste* relevanter Einheiten und deren Aktionen/Ereignissen (aktive/passive „Form“)
   - Zu jeder Aktion:
     (a) textuelle Beschreibung
     (b) mögliche Attribute (– „funktionale“ – Parameter).
     Z.B. Einzahlung(Datum, Betrag).

     Eine „Aktion/Ereignis“ kommt in der realen modellierten Welt vor (nicht „im“ System). Ist elementar und nicht zerlegbar.
     Eine „Einheit“ führt aus oder erleidet Aktion in zeitlicher Folge. Element der realen modellierten Welt (nicht nur „im“ System). Erkennbar als Individuum.
     Verwendung einer Aktion oder Einheit im Modell: System benötigt oder erzeugt Information über sie. Gegenstand/Ziel des Systems.

2. **Entity-Structure**-Step
   Zeitliche Anordnung der Aktionen zu den Einheiten (mit Jacksondiagrammen). Mischung von Daten und Aktionen in einem Diagramm. (Kunde; investieren, beenden)
3. **Initial-Model**-Step
   Bisher Modellierung der Realwelt. Jetzt erstes Modell des Systems: Prozesse und Datenflußbeziehungen dazwischen.
   Aktion einer Einheit erzeugt Eingabe an das System (meist sequentieller Datenstrom).
   Darstellung mit „System Specification Diagram“: Obige real-world-Einheit („entity“) erhält ein system-Pendant („process“), der die Eingabe entgegennimmt. Der Prozeß („Name-1“) simuliert Vorgänge der Real-Einheit („Name-0“) im System.
   Die innere Funktion dieses Systemprozesses wird wie gehabt beschrieben (textuell-logisch oder Diagramm).
4. **Function**-Step
   Spezifikation von Ausgabe-Funktionen: Kombination von Ereignissen in der Realwelt erzeugt eine bestimmte Ausgabe.
   Technik: Informelle Beschreibung in der Sprache von Einheiten und Aktionen. Überarbeiten des SSD, wie für Funktion erforderlich. Jacksondiagramm für genaue Verhaltensbeschreibung.
   Man fügt hier praktisch nur die Ausgabefunktionen zu obigem Diagramm hinzu.
5. **System-Timing**-Step
   Festlegung der zeitlichen Abfolge von Ausgaben des Systems. Tolerierbare Verzögerungen, gewünschte Zeitabläufe. Keine formalen Techniken.
6. **Implementation**-Step
   Transformationsschritt zur praktischen Umsetzung: Abbilden der nebenläufigen, sequentiellen Prozesse der Spezifikation auf tatsächlich vorhandene Prozessoren. Wenig standardisiert, daher nicht gesondert strukturiert.

9.3 Zusammenfassung

Ähnlichkeiten zu

- OOA
  Auffinden von Realweltobjekten. Entity (process) besitzt Zustandsvektor und führt Aktionen aus (Initiative). Aber keine Fähigkeiten im Sinne von Methoden. (Aktionen an/mit Einheiten)
- SA
  Datenflußbeziehungen (SSD), auch SADT, Hierachie?, Data Dictionary, kein Datenmodell
- RT
  Zeitverhalten von Prozessen, Präzision?

Starke Orientierung an den Ein- und Ausgaben eines System! Transformation strukturierter Daten in andere Strukturen. Konzeptionell durchgängige Technik von Spezifikation bis Implementation. Gut geeignet bei hoch interaktiven, transaktionsorientierten Problemen.

10.Structured Design (SD)

10.1 Einführung

- Zerlegung von Funktionen in Module
- Konzentration auf Kooperation dieser Module
  - Datenaustausch
  - Aufrufstruktur

>>Modul: Funktion, Unterprogramm. Definition durch

- übernommene Eingaben
- erzeugte Ausgaben
- realisierte Funktion (Singular!)
- seine internen Datenstrukturen

Wesentliches Strukturierungskonzept: Modul A benutzt Modul B. D.h. A ruft B auf, B tut sein Werk, B übergibt Kontrolle zurück an A unmittelbar hinter den Aufrufpunkt.

10.2 Structure Charts

Zentrales Ausdrucksmittel des SD. Graphische Darstellung

- der Aufrufbeziehungen und
- des Datentransfers

zwischen Modulen.

Bildelemente:

- Rechteck: Modul
- gerichtete Linie: „ruft auf“
  Orientierung oben-unten ist meistens relevant, Pfeilspitze zeigt Richtung.
- Speziell Symbole für
  - Datenkopplung
  - Übergabe von „flags“
- Rechteck mit Doppellinien: Vordefiniertes Modul.
  Keine Verfeinerung nötig. Vorheriges Projekt oder Systemfunktion.

Daten werden verarbeitet und sind Teil des Problems. „Flags“ werden gesetzt und getestet. Sie sind Kontrolldaten, Informationen über Daten.

Beschriftungen an Modulen, Datenkopplung und Flags. Modulbeschreibung zusätzlich mit strukturiertem Englisch, Pseudocode oder Entscheidungstabellen.

10.3 Entwurfsqualität

Konkreter Bezug von Kopplung und Kohäsion auf SD-Module:

10.3.1Kopplungsarten

- Normale Kopplung
  A ruft B auf und übergibt Parameter X, erhält Parameter Y zurück.
  - Einfache Datenkopplung
    Alle Kommunikation über Parameter. Exakt nur das, was notwendig ist.
  - Strukturierte Datenkopplung
    Strukturierte Parameter, die nur teilweise gebraucht werden.
  - Kontrollkopplung
    Übergabe von Kontrollinformation, womit das Empfängermodul gesteuert wird.
- Speicherkopplung
  Gemeinsamer Zugriff mehrerer Module auf gemeinsamen Speicher, globale Variablen.
- Inhaltskopplung
  Abhängigkeit eines Moduls von der Implementation eines anderen Moduls. Z.B. Springen in einen anderen Programmteil, Setzen interner Daten eines anderen Moduls.

10.3.2Kohäsionsarten

- Funktionale Kohäsion
  Bearbeitet eine einzige problembezogene Funktion.
- Sequentielle Kohäsion
  Stellt verschiedene Funktionen dar, sind aber streng sequentiell aufgereiht.
- Kommunikations-Kohäsion
  Tätigkeiten (im Modul) arbeiten mit den gleichen Ein- oder Ausgabedaten.
- Prozedurale Kohäsion
  Tätigkeiten arbeiten in einer gewissen kontrollierten Reihenfolge.
- Zeitliche Kohäsion
  Tätigkeiten müssen in zeitlichem Zusammenhang arbeiten.
- Logische Kohäsion
  Tätigkeiten haben ähnliche Aufgaben.
- Zufällige Kohäsion
  Tätigkeiten haben nichts miteinander zu tun.

10.4 Systematische Entwicklung einer Aufrufstruktur

Viele Programme haben die Struktur „Eingabe-Verarbeitung-Ausgabe“ oder bilden eine Transaktionsstruktur.

> Transaktion ist eine Einheit (Kommando, Datensatz,…),
> mit der eine von mehreren Aktionen geschehen soll.

1. Entwickeln einer groben funktionalen Sicht auf das System. Etwa eine globale funktionale Dekomposition in sehr wenige Teilfunktionen.
2. Ablesen der „großen“, abstrakten Datenströme des Systems.
3. Identifikation eines Hauptdatenstroms. Auswahl eines Datenstroms, der im Normalfall die treibende Kraft des Systems darstellt. Also Identifikation von Haupteingabestrom und Hauptausgabestrom.
   Finden des Punktes höchster Abstraktion. Übergang von Eingabedaten in Ausgabedaten. Dazwischen liegen die zentralen Transformationen.
4. Entwurf der obersten Ebene des Strukturgraphen gemäß 3b.
5. Verfeinern der Module rekursiv.

11.Aufwandsabschätzung

>>Ziel: Voraussagen über Aufwand und Zeitbedarf zum Abschätzen von Projektkosten und Planen des Einsatzes der Mitarbeiter.

- Analogie
  Vergleich mit schon existierendem System
- Kampfpreismethode
  Soviel Aufwand planen, wie das Produkt einbringt.
- Top-down
  Ausgehend von der Anforderungsbeschreibung Ermitteln von vermuteten Softwarekomponenten, diese abschätzen bzgl. Aufwand und Zeit.
- Bottom-Up
  Nach Erstellen der Modularisierung, Einzelschätzungen von Aufwand und zeit für Einzelfunktionen.
- Produktivitätsfaktoren
  Schätzung der Codegröße, Annahme der Programmiererproduktivität und daraus Aufwand und Zeitbedarf.
- Aktivitätenpläne
- Komplexitätsmodelle
  Ermitteln eines „Kompliziertheitsmaßes“ von Software, welches eng mit dem Erstellungsaufwand zusammenhängt. Z.B. Aufwand * Zeitbedarf = Konstant.
- Faktorenanalyse
  Aufwand als Ergebnis einer linearen gewichteten Summe, Koeffizienten sind Faktoren.