Informatik IV - Zusammenfassung

Patrick Pletscher

15. September 2004

1 Modularität, Wiederverwendbarkeit

1.1 Modularität

Einige Prinzipien von Modularität:

- Zerlegbarkeit
- Zusammensetzbarkeit
- Kontinuität
- Information Hiding
- Das "Offen-Geschlossen" Prinzip
- Das "Einzige Auswahl" Prinzip

Zerlegbarkeit

Komplexe Probleme in Teil-Probleme zerlegen. Also Aufspaltung der Arbeit. Ein Beispiel dafür ist die Top-Down-Design-Methode (dabei wird das Programm immer feiner und feiner spezifiziert, Baum!). Ein Gegenbeispiel wäre ein Allgemeines Initialisierungs-Modul.

Zusammensetzbarkeit

Erzeugung von Software-Elementen die frei untereinander zusammengesetzt werden können, um neue Software zu erzeugen.

Direktes Abbilden

Diese Methode ergibt ein Software-System dessen modulare Struktur kompatibel bleibt mit irgendeiner modularen Struktur, die beim modellieren der Problem-Domäne erdacht wird.

Prinzip der wenigen Schnittstellen

Jedes Modul redet mit so wenigen anderen Modulen wie möglich.

Prinzip der kleinen Schnittstellen

Wenn zwei Module kommunizieren, tauschen sie so wenig Information aus wie möglich.

Prinzip der expliziten Schnittstelle

Wenn zwei Module A und B kommunizieren, ist das aus dem Text von A oder B oder von beiden offensichtlich.

Bsp. Modul A modifiziert Daten x und Modul B liest diese.

Kontinuität

Stellt sicher, dass kleine Änderungen an der Spezifikation auch kleine Änderungen an der Architektur ergeben.

Prinzip des einheitlichen Zugriffs (Uniform Access)

Funktionen, die von einem Modul verwaltet werden, sind für Kunden in der selben Weise zugreifbar, egal ob sie mit Berechnung oder Speicherung implementiert sind.

Information Hiding

Jedes Modul sollte der externen Welt durch eine offizielle, öffentliche, "public" Schnittstelle bekannt sein.
Die restlichen Eigenschaften des Moduls sind seine "Geheimnisse".

Das Offen-Geschlossen Prinzip

Offenes Modul: Kann erweitert werden.
Geschlossenes Modul: Für Kunden verwendbar. Kann freigegeben, als Grundlinie (baseline) verwendet und (falls eine Programmeinheit) kompiliert werden.

Ein Modul sollte offen und geschlossen sein.

2 Abstrakte Datentypen (ADT)

2.1 Argumente für Objekte

- Wiederverwendbarkeit: Man muss ganze Datenstrukturen wiederverwenden, nicht nur Operationen.
- Erweiterbarkeit, Kontinuität: Objekte bleiben im Zeitverlauf stabiler.

Definition (Objekt-Technologie). Objekt-orientierte Software Konstrukion ist die Herangehensweise an das Strukturieren von Systemen, die die Architektur von Software Systemen auf den Typen der Objekte basiert, die sie manipulieren - nicht auf "der" Funktion die sie erreichen.

Motto: Frage nicht zuerst was das System tut, frage worauf es etwas tut.

Definition (Objekt-Technologie 2). Objekt-orientierte Software Konstruktion ist die Konstruktion von Software Systemen als strukturierte Sammlungen von (möglicherweise partiellen) Implementierungen von abstrakten Daten-Typen.

2.2 Beschreibung von Objekten

Betrachte nicht ein einzelnes Objekt, sondern den Typ von Objekten mit ähnlichen Eigenschaften. Definiere jeden Typ von Objekten nicht durch die physikalische Repräsentation, sondern durch ihr Verhalten: die Dienste (Features) die sie der restlichen Welt anbieten.

2.3 Partielle Funktionen

Definition (Partielle Funktion). Eine partielle Funktion, hier durch $\not\to$ identifiziert, ist eine Funktion die nicht für alle möglichen Argumente definiert sein muss.

Für partielle Funktionen benutzt man in Eiffel Vorbedingungen, hier z.Bsp. für einen Stack:

```
remove(s: STACK[G]) require not empty(s)
item (s: STACK[G]) require not empty(s)
```

2.4 Axiome

Man definiert Axiome für die Funktionen des ADT, also z.Bsp.

```
remove(pop(s, x)) = s
```

2.5 Ausreichende Komplettheit

Drei Formen von Funktionen in der Spezifikation eines ADT `T`:

- Creators: `OTHER` $\to$ `T`
- Queries: `T` $\times \ldots \to$ `OTHER`
- Commands: `T` $\times \ldots \to$ `OTHER`

Ausreichend komplette Spezifikation: ein "Anfrage-Ausdruck" der Form: $f(\ldots)$ wobei f eine Anfrage ist, kann durch die Anwendung der Axiome in eine Form ohne `T` gebracht werden.

2.6 Implementierung eines ADT

Drei Komponenten:

(E1) Die Spezifikation des ADT: Funktionen, Axiome, Vorbedingungen

(E2) Die Auswahl einer Repräsentation

(E3) Eine Menge von Subprogrammen (Routinen) und Attributen, die jeweils eine der Funktionen der ADT Spezifikation (E1) mit der gewählten Repräsentation (E2) implementieren

3 Objekte und Klassen

Eine Klasse ist die Implementierung eines abstrakten Datentyps. Es ist gleichzeitig ein *Modul* und ein *Typ*.

Eine Klasse wird durch eine Menge von *Features* beschrieben. Features umfassen *Attribute* (Speicherzellen von Instanzen der Klasse) und *Routinen* (Operationen auf den Instanzen). Bei Routinen unterscheidet man zwischen *Prozeduren* (beeinflusst die Instanz, kein Rückgabewert) und *Funktionen* (berechnet Rückgabewert, normalerweise kein anderer Effekt).

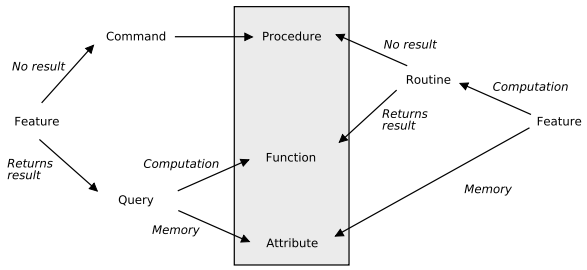


Abbildung 1: *Feature Kategorisierung*

3.1 Benutzung der Klasse beim Klienten

```
class GRAPHICS

feature
  p,q: POINT

  some_routine is
    local
    do
      create p
      create q
      p.move(4.0,-2.0)
    end
end
```

3.2 Shallow Clone vs Deep Clone

Bei shallow clone wird nur das Objekt selbst gecloned, nicht aber Attribute, welche wiederum auf andere Objekte zeigen. Bei deep_clone werden diese Objekte (rekursiv) auch wieder kopiert.

4 Design by Contract

4.1 Precondition/Postcondition

Es können Preconditions und Postconditions wie folgt definiert werden.

```
require
  key_not_present: not has(key)
  -- ..
ensure
  insertion_done: item(key) = new
```

```
-- ..
```

4.2 Weitere Assertions

check Kann irgendwo im Code einer Methode benutzt werden um eine Eigenschaft zu prüfen.

invariant (in loop) Für Korrektheit einer Schlaufe, diese Eigenschaft muss am Ende jedes Schlaufendurchlaufs gelten.

variant (in loop) Für die Terminierung einer Schlaufe, Ausdruck muss ein Integer sein.

4.3 Axiome/Invarianten

Invarianten können wie folgt für die ganze Klasse nach `feature` definiert werden:

```
invariant
  not_under_minimum: balance >= Minimum_balance
```

4.4 Korrektheit einer Klasse

Für jede Erstellungsprozedur *cp*:

$$\{Pre_{cp}\}\ \mathsf{do}_{cp}\ \{Post_{cp}\ \mathsf{and}\ INV\}$$

Für jede exportierte Routine r:

$$\{INV\ \mathsf{and}\ Pre_r\}\ \mathsf{do}_r\ \{Post_r\ \mathsf{and}\ INV\}$$

5 Design by contract und Exception Handling

5.1 Contracts und Invarianten

Invarianten Verberbungs Regel

Die Invarianten einer Class beinhaltet automatisch auch die Invarianten ihrer Parents, verundet.

5.2 Assertion Redeklarations Regel

Wenn man eine Routine redeklariert, so kann die Precondition nur geschwächt werden und die Postcondition nur verstärkt.

5.3 Assertion Redeklarations Regel in Eiffel

Die redefinierte Version kann keine neuen `require` oder `ensure` beinhalten (die Assertions werden übernommen), oder

```
require else new_pre
ensure then new_post
```

Die resultierenden Assertions sind:

```
original_precondition or new_pre
original_postcondition and new_post
```

5.4 Exception Handling

Zwei Konzepte falls der Contract gebrochen ist:

Failure Einer Routine, oder anderen Operation ist es unmöglich den Contract zu erfüllen.

Exception Ein ungewünschtes Ereignis tritt auf während der Ausführung einer Operation - als ein Resultat eines Failure, einer Operation welche von der Routine aufgerufen wurde.

In Eiffel gibt es ein Konstrukt `rescue`, welches man für das Exception Handling einer Routine benutzen kann. Dieser Block wird ausgeführt, falls im `do` Teil der Routine ein Fehler auftritt. Hier kann `retry` benutzt werden um das Ganze nochmals zu versuchen.

6 Genericity

6.1 Eine generische Klasse

```
class STACK[G]

feature
  put (x: G) is ...
  item: G is ...
end
```

Um die Klasse zu benutzen:

```
account_stack: STACK[ACCOUNT]
```

6.2 Typing im O-O Kontext

Definition. Eine objekt-orientierte Sprache ist statisch getypt genau dann wenn es möglich ist einen statischen Checker zu schreiben, welcher ein System akzeptiert, garantiert das zur Laufzeit für eine Ausführung von einem Feature `x.f`, das Objekt welches zu `x` gehört (wenn es eines gibt) mindestens eine Feature `f` hat.

7 Inheritance

```
class
  RECTANGLE
inherit
  POLYGON
   redefine perimeter end
..
feature
  perimeter: REAL is
  ..
```

7.1 Polymorphism und dynamic Binding

An Vorgänger kann Nachfolger zugewiesen werden, aber nicht umgekehrt.

```
p: POLYGON; r: RECTANGLE;
```

```
-- Permitted
p := r
-- Not permitted
r := p
```

Redefinition Eine Class kann vererbte Features ändern.

Polymorphism p kann verschiedene Formen haben zur Laufzeit.

Dynamic binding Effekt einer Funktion von p hängt von der Laufzeit Form von p ab.

Um die Originale Version eines redefinierten Features zu benutzen:

```
Precursor {WINDOW}
-- my new code here
```

7.2 Genericty

Assignment attempt

```
x: A
..
x ?= y
```

Falls `y` von einem Typ ist, der konform ist mit `A`, so führe ein normales Reference Assignment aus. Sonst ist `x` void.

7.3 infix Operatoren

```
class
  VECTOR[G]
feature
  infix "+" (other: VECTOR[G]): VECTOR[G] is
  ....
```

7.4 Erstellen mit einem spezifizierten Typ

```
a1: A
..
create {B} a1.make(..)
```

Erstellt `a1`, aber mit Typ `B`

7.5 Once Routinen

```
r is
      once
        --instructions
      end
```

So werden die Instruktionen nur ausgeführt, für den ersten Aufruf bei einem Client während der Ausführung. Weitere Aufrufe returnen sofort. Falls es Funktionen sind, wird bei weiteren Aufrufen, das Resultat vom ersten Aufruf zurückgegeben.

7.6 Terminologie

Parent Eine Klasse von der die gegebene Klasse vererbt.

Child Eine Klasse die von der gegebenen Klase vererbt.

Heir = Child

Ancestor Die Klasse selber oder ein direkter oder indirekter Parent davon.

Proper ancestor Ein direkter oder indirekter Parent der Klasse.

Descendant Die Klasse selber oder einer seiner direkten oder indirekten Kindern.

Proper descendant Ein direktes oder indirektes Kind der Klasse.

Instance Ein Objekt das nach der Form die durch die Klasse oder einer seiner proper descendants definiert ist, erstellt wurde.

Direct instance Ein Objekt, das nach der Form die durch die Klasse definiert ist, erstellt wurde.

7.7 Multiple inheritance

```
class
  COMPOSITE_FIGURE
inherit
  FIGURE
    redefine display, ... end
  LIST[FIGURE]
feature
  display is
     do
       from start until after loop
          item.display
          forth
       end
     end
```

7.8 Richtiges Benutzen von Inheritance

Zwei Relationen: Client, Inheritance

Client drückt aus, dass Instanzen von B Informationen über Instanzen von A speichern muss.

Inheritance drückt aus, dass jede Instanz von D als Instanz von C betrachtet werden kann; wenn C ein Ancestor von D ist.

Ausser für polymorphe Benutzungen, wird Inheritance nie benutzt: Statt dass B von A vererbt, kann B immer ein Attribut vom Typ A beinhalten, ausser wenn ein Eintrag vom Typ A möglicherweise Werte vom Typ B repräsentieren soll.

7.9 Contracts und Inheritance

Was passiert mit Klasseninvarianten und Pre-/Postconditions unter Vererbung?

- Die *Invarianten* einer Klasse beinhalten automatisch auch die Invarianten von all ihren Eltern welche "verundet" werden.
- Die *Preconditions* können nur behalten oder abgeschwächt werden.
  `new_pre or else original_precondition`
- Die *Postconditions* können nur behalten oder verstärkt werden.
  `original_postcondition and then new_post`

7.10 Features verschmelzen

Falls es von den vererbten Klassen jeweils Features gibt, die in mehreren Klassen definiert sind, wobei aber alle "Implementationen" bis auf eine deferred sind, so wird die effektive Implementierung gewählt. Sonst muss man sich mit folgenden Tricks behelfen.

Undefinieren eines Features

```
deferred class
  B
inherit
  A
    undefine
      f
    end
feature
  ..

end
```

Umbenennen eines Features

```
..
inherit
  A
    rename
      g as f
    end
```

Repeated Inheritance Probleme

Angenommen eine Klasse ist das gemeinsame Child von zwei Klassen. Was passiert mit Features die zweimal von dem common ancestor (der beiden) vererbt wurde?

- Features die nicht umbenannt wurden entlang dem Vererbungspfad werden gemeinsam benutzt.
- Features die unter verschiedenen Namen vererbt wurden, werden repliziert.
  So kann es aber zu Problemen durch dynamic binding und Polymorphismus kommen. Dafür wird `select` benutzt.

7.11 select

```
class
  SWISS_US_TAXPAYER
inherit
  SWISS_TAXPAYER
    rename
      address as swiss_address,
      tax_id as swiss_tax_id,
      ..
    select
      swiss_address,
      swiss_tax_id,
      ..
    end
  US_TAXPAYER
    rename
      address as us_address,
      tax_id as us_tax_id,
      ..
```

7.12 Export Anpassung

```
class
  B
inherit
  A
    export
      {ANY} all
      {NONE} h
      {A,B,C,D} i,j,k
    end
feature
  ..
```

8 Agents

8.1 Anwendungen von Agents

- Iteration
- High-level contracts
- Numerische Programmierung

8.2 Offene und geschlossene Argumente

agent your_function($\underbrace{?}_{\text{Open}}$, $\underbrace{u,\ v}_{\text{Closed}}$)

Closed: Wird zur Zeit der Definition des Agents gesetzt.
Open: Wird bei jedem Aufruf des Agents gesetzt.

8.3 Beispiele

Agent-Funktion ist in gleicher Klasse

```
..
my_integer_list: LIST[INTEGER]
..
is_positive(x: INTEGER):BOOLEAN is
  do
    Result := (x > 0)
```

```
  end
```

Um nun zu Testen, ob alle Zahlen in Liste positiv sind:

```
all_positive :=
 my_integer_list.for_all(agent is_positive)
```

Agent-Funktion ist in externer Klasse

```
..
my_employee_list: LIST[EMPLOYEE]
..
is_married: BOOLEAN
```

Um nun zu Testen, ob alle Angestellten in der Liste verheiratet sind

```
all_married :=
 my_employee_list.for_all(agent {EMPLOYEE}.is_married)
```

8.4 Potential der Agents für Contracts

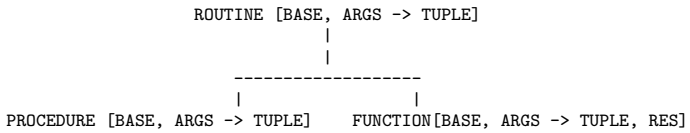
Agents können benutzt werden um generelle Eigenschaften, wie z.B. alle Elemente von 1 bis n haben sich nicht geändert auszudrücken.

8.5 Tuple Klasse

`TUPLE[X, Y, ...]`

`TUPLE[A,B]` ist ein Child von `TUPLE[A]` welches wiederum ein Child von `TUPLE` ist. Es bedeutet also: Tuple von mindestens n Argumenten.

Kernel library classes



Mathematisches Modell für Tupel

- Erste Intuition: `TUPLE[A, B, C]` repräsentiert das kartesische Produkt $A \times B \times C$
- Aber: $A \times B \times C$ kann nicht auf die Untermenge $A \times B$ abgebildet werden
- Besseres Modell: `TUPLE` stellt die Menge der partiellen Funktionen von $\mathbb{N}$ zur Menge der möglichen Werte, welche die Domain das Intervall $[1 \dots n]$ beinhaltet, dar.

Features der Routine Klasse

```
call(values: ARGS)
item(values: ARGS): RES -- In FUNCTION only
```

8.6 Argumente offen lassen

Nachfolgend finden sich Beispiele für die Typen von Agents.

```
a0: C; a1: T1; a2: T2; a3: T3;

u := agent a0.f(a1, a2, a3)
-- hat Typ PROCEDURE[C, TUPLE]

-- Beispielaufruf
u.call([])

v := agent a0.f(a1, a2, ?)
-- hat Typ PROCEDURE[T3]

-- Beispielaufruf
y.call([a3])
```

8.7 Ziel offen lassen

```
r := agent {T0}.f(a1, a2, a3)
-- hat Typ PROCEDURE[T0, TUPLE[T0]]

-- Beispielaufruf
r.call([a0])
```

8.8 Iteratoren

In `LIST[G]` findet sich:

```
for_all(test: FUNCTION[ANY, TUPLE[G], BOOLEAN]) is
   -- Is there no item in structure that doesn't
   -- satisfy test?
  do
    ..
  end
```

Weitere Iteratoren

```
for_all
there_exists
do_all
do_if
do_while
do_until
```

9 Event-driven programming

Es gibt Publishers, welche Events triggern und Subscribers welche die Events behandeln.

9.1 Event Library (EVENT_TYPE)

Publisher Seite

```
..
-- Deklarierung
click: EVENT_TYPE[TUPLE[INTEGER, INTEGER]]
..
-- Instanzierung
create click
```

```
..
-- Jedes Mal wenn Event eintrifft
click.publish([x_coord, y_coord])
```

Subscriber Seite

```
click.subscribe(agent my_procedure)
```

10 Concurrent Programming

A simple, general and powerful concurrency and distribution model (SCOOP).

10.1 Was macht eine Applikation nebenläufig?

Definition (Prozessor). Unabhäniger Kontroll-Thread welcher sequentielle Ausführung von Instruktionen auf einem oder mehreren Objekten unterstützt.

Kann implementiert sein als:

- Computer CPU
- Prozess
- Thread
- AppDomain (.NET)

Die Zuordnung von Prozessor zu Rechnerressourcen erfolgt durch Concurrency Control File.

10.2 Feature call

synchroner Aufruf:

```
x: CLASS_X
```

asynchroner Aufruf:

```
x: seperate CLASS_X;
```

Ziel eines separaten Aufruf muss ein formales Argument von umgebender Routine sein:

```
store (b: seperate BUFFER[G]; value: G) is
 -- Store value into b
do
  b.put(value)
end
```

Um exklusiven Zugriff auf ein seperates Objekt zu erhalten benutzt man es als Argument im Aufruf:

```
buffer: seperate BUFFER[INTEGER]
create buffer
store(buffer,10)
```

10.3 Von Preconditions zu Wait Condition

```
store(b: separate BUFFER[G]; value: G) is
require
 not b.is_full
 value /= void
..
```

Wenn b separate ist, so wird die Precondition zu einer wait Condition. Es wird also gewartet, bis b Platz hat und erst dann wird weiter "gearbeitet".

Der Client wartet nur, falls es nötig, also falls z.B. ein Query auf ein separates Objekt ausgeführt wird, bzw. erst wenn dieser Wert dann wirklich gebraucht wird.

11 Designprinzipien, Design für Wiederverwendbarkeit

- Command / Query Seperation Prinzip
- Systematische Namenskonventionen
- Operanden / Optionen Seperations Prinzip

11.1 Command / Query Seperations Prinzip

Ein Command tut etwas, aber gibt kein Resultat zurück. Ein Query gibt ein Resultat zurück, verändert den Zustand aber nicht. Ein Gegenbeispiel wäre das C Konsrukt `getint()`.

Referential transparency

Eine Funktion gibt immer das selbe zurück, also falls $a = b$, so ist auch $f(a) = f(b)$.

```
io.read_integer
n := io.last_integer
```

guter Stil: Lösen von $\mathbf{Ax} = \mathbf{b}$

```
A.try_to_solve(b)

if not A.singular then
  x := A.solution
end
```

11.2 Generelle Prinzipien

- Konsistenz.
  Wie kann man neue Designentscheidung mit bisherigen vereinen?
  Wie mache ich Designentscheidung, so dass zukünftige Entscheidungen einfach damit kompatibel sind?
- Benute Assertions.
- Symmetrien bevorzugen. Z.B. before und after selbes Verhalten.
- Limit-Fälle. Full oder Empty?

11.3 Operanden und Optionen

Zwei mögliche Arten von Argumenten von Features:

Operanden Auf was Feature operiert.

Optionen Einstellungen, wie Feature operiert.
Sind meist erkennbar daran, dass Default-Werte Sinn machen. Die Optionen können sich im Verlauf der Entwicklung der Klasse ändern, Operanden sollten gleich bleiben.

Die Argumente eines Features sollten nur Operanden sein. Optionen sollten Default-Werte haben, und es sollte Prozeduren geben, mit denen man die einzelnen Optionen setzen kann, falls dies gewünscht ist.

11.4 Grammatikalische Regeln

- Prozeduren: Verben, infinitiv.
- Boolesche Queries: Adjektive, z.B. full oder is_some_property
- Andere Queries: Nomen oder Adjektive
- Keine Verben für Queries.

12 Software lifecyle Modelle

Prozess-Qualität

- Pünktlichkeit
- Kosteneffektiv

12.1 Lifecyle Modelle

Lifecylce Modelle haben zum Ziel die Menge von Prozessen, welche bei der Produktion eines Softwaresystems involviert sind zu beschreiben und zu sequenzieren. Es gibt versch. Modelle und Standards, z.B. das Capability Maturity Model (CMM) wobei dies v.a. auf die Dokumentation ausgerichtet ist.

Wasserfall Modell

Jeweils Pfeile zurück und vor.

1. Feasibility study
2. Requirements analysis
3. Specification
4. Global design
5. Detailed design
6. Implementation
7. Validation & Verification
8. Distribution

Probleme mit dem Wasserfall Modell:

- Spätes Erscheinen von wirklichem Code.
- Es fehlt der Support von Requirements Änderungen und allgemeiner für extendibility und reusability.
- Es fehlt der Support von Maintance Aktivitäten.
- Sehr synchrones Modell.

Seamless (= nahtlos) Development

Spezifikation (TRANSACTION, PLANE, CUSTOMER, ENGINE,...), Design (STATE, USER_COMMAND,...), Implementation (HASH_TABLE, LINKED_LIST,...), Verification & Validation (TEST_DRIVER,...), Generalization (AIRCRAFT,...).

Cluster Model

Verschiedene Seamless Development (= Cluster i) beliebig parallel seriell oder verschoben entwickeln.

Bottom-Up development: von den allgemeinsten clustern zu den Applikationsspezifischten. Flexibles schedulen von Clustern, Wasserfall vs. Trickle als Extrembeispiele.

13 Configuration Management

Definition. Configuration Management ist die eindeutige Identifizierung, kontrollierte Speicherung, Änderungskontrolle, und Statusreporting von ausgewählter zwischenzeitlicher produzierter Arbeit, Produkt Komponenten, und Produkte während dem Leben eines Systems.

Oder kurz: Configuration Management ist die Rolle der ZEIT in der Software Entwicklung.

Ein System das Dokumente speichert und organisiert über einem Raum und Zeit wird als Repository bezeichnet.

13.1 Begriffe

Versionisierung

Versionen geben eine eindeutige zeitabhängige Identifikation von jedem Dokument.

Views

Man kann all die verschiedenen Dokumente zu einem bestimmten Zeitpunkt in der Vergangenheit anschauen, es wird dann von jedem Dokument das damals aktuellste genommen.

Head

Die aktuellste Version von jedem Dokument.

Branching

Erstellen von mehreren Versionen von einer existierenden Menge von Dokumenten wird als BRANCHING (oder FORKING) bezeichnet.

MERGING

Zusammenführen von Versionen die unabhängig über die Zeit entwickelt wurden.

13.2 Regression Testing

Testen von Features, die in *älteren Versionen schon funktionierten*, wobei man testet, ob sie in neueren Versionen der Software auch funktionieren. Regression Testing ist im Normalfall Teil des Commit, oder des Daily Build.

13.3 Bug-Tracking

Infrastruktur, die Bug-Reports festhält und managed in einem Projekt. Die Anzahl und Qualität der Bugs ist im Normalfall ein Kriterium für das Release der Software.

14 Projektmanagement

14.1 Grundlagen

Definition (Projekt). Ein Projekt ist eine temporäre Unternehmung unternommen um ein einmaliges Produkt oder Service zu erstellen.

Operationen sind dagegen laufend und wiederholend.

Definition (IT-Projekt). Ein IT-Projekt ist ein Projekt um ein Produkt oder Service zu erstellen, bei welchem die Benutzung von Informationstechnologie die entscheidende Charakteristik ist.

Definition (Projekt Management). Projekt Management ist die Anwendung von Wissen, Fähigkeiten, Tools, und Techniken so, dass die Projektaktivitäten die Projektanforderungen treffen.

PM Wissensgebiete

- Projekt Integrations Management
- Projekt Kosten Management
- Projekt Kommunikations Management
- Projekt Umfang Management
- Projekt Qualitäts Management
- Projekt Risiken Management
- Projekt Zeit Management
- Projekt Human Resource Management
- Projekt Zulieferer Management

14.2 Integrations Management

Zusichern, dass verschiedene Elemente vom Projekt geeignet koordiniert sind. Ist Sache des Projekt Managers.

Triple Constraint

Zeit, Umfang und Kosten sind eng miteinander verknüpft, wenn man eines ändert, so ändern sich auch die anderen. Wenn man also sparen will, so kann man dies nur unter Beschneidung der anderen beiden tun.

Die Prioritäten sind durch die Kunden definiert, der Job des PM ist es aber einen geeigneten Tradeoff zu suchen.

Mehr konkurrenzierende Objekte

Qualität, Zeit, Kundenzufriedenheit, Kosten, Risiken, Umfang.

Projekt Erfolg

Definition. Ein Projekt ist erfolgreich, wenn die spezifizierten Resultate in der geforderten Qualtiät und in der ausgemachten Zeit und Resourcen Limiten geliefert wird.

14.3 Projekt Lebenszyklus

Aufteilen von Projekten in Unterprojekte. Diese teilt man wiederum auf in Phasen.

Unterprojekte

Unterteilung folgt der Struktur des Produkts.

Fortschreitende Ausarbeitung

Charakteristiken von einem einmaligen Produkt oder Service müssen laufend ausgearbeitet werden.

Deliverables (= Ergebnis)

Definition. Irgendein messbares, greifbares und überprüfbares Ergebnis, Resultat oder Element welches produziert werden muss um ein Projekt oder ein Teil eines Projekts fertigzustellen.

Projekt Phasen

Eine Sammlung von logisch verknüpften Projektaktivitäten im allgemeinen gipfelnd in der Vollendung von einem grossen Deliverable.

Projektlebenszyklus

Der Einfluss der Aktionäre auf die Produktcharakteristiken und finale Kosten ist zu Projektbeginn am Grössten und nimmt laufend ab. Die Kosten sind zu Beginn und Ende nicht so hoch wie dazwischen.

Von Projekten zu Operationen

- Ideen, Studien (gehört nicht zum Projekt)
- Analysephase
- Designphase

- Implementationsphase
- Testphase
- Deployment (=Aufstellen) Phase
- Operation, Produktion (gehört nicht zum Projekt)

14.4 Projekt Management Life Cycle

Wie die Core Processes werden auch die Projekt Management Prozesse unterteilt, hier aber in sogenannte Gruppen.

- Initiating Processes
- Planning Processes
- Controlling Processes
- Executing Processes
- Closing Processes

Diese Gruppen überlappen und sind nicht diskrete, einmalige Events.

15 Exception Handling in OO

15.1 Exception Vokabular

- Raise, trigger oder wirf eine Exception
- Handle, oder fange eine Exception

15.2 In Java

Raise

```
my_routine(..) throws my_exception {
  ..
  if abnormal_condition
    throw my_exception;
}
```

Handle

```
try {
  instruction_1;
  instruction_2;
  ..
  instruction_n;
}
catch (Expected_exc_type e){
  handling_code;
}
finally {
 // for both
}
```

15.3 Exception handling

- Failure: Einer Routine oder anderen Operation ist es unmöglich seinen Contract einzuhalten.
- Exception: Ein nichtgewünschtes Event tritt während der Ausführung einer Routine ein, als ein Resultat eines Failure einer Operation welche von der Routine aufgerufen wird.

15.4 Exception Korrektheit

$$\{True\} rescue_r \{INV\}$$

16 Quality Assurance

16.1 Validation und Verifikation

Ein Software Element muss:

- Die richtige Sache machen (Validation)
- Die Sache richtig machen (Verifikation)

16.2 Terminologie

Ein *Error* im Softwarekonstruktionsprozess löst einen *Fault* im Produkt aus, welches während einer Operation ein *Failure* der Programmausführung bedeutet.

16.3 Verifikationstechniken

Statisch (keine Ausführung) vs Dynamisch (Ausführung, aktuelle oder simulierte)