Einführung Softwaretechnologie

Holger Kreissl, Januar 2003

CHEMNITZ UNIVERSITY
OF TECHNOLOGY

Basierend auf dem Skript von Prof. Kroha

Ausgearbeitet von Holger Kreissl
*Fehler bitte an **holger@kreissl.info** senden.*

1. Inhaltsverzeichnis

2. Softwaretechnologie I

***Softwaretechnik**: „multi-person construction of multi-version software“ (Parnas (1987)*

2.1. Einführung in Software Engineering

1. Software Engineering - charakteristische Eigenschaften und Ziele

- Fehler beheben, Funktionalität erweitern, Altes Entfernen oder an neue Umgebungen anpassen
- Gemeinsame Prinzipien sind
    1. Separation der Belange / Anforderungen
    2. Modulatität (Dekomposition komplexer Probleme, d.h. Top Down oder Zusammensetzen eines komplexen Systems aus bestehenden Fabrikteilen, d.h. Buttom-up)
    3. Abstraktion (wichtige Dinge herausheben und Details ignorieren und gegebenfalls verschiedene Abstraktionen der gleichen Realität darstellen)
    4. Voraussicht auf Änderungen (d.h. bei Entwicklung bedenken, daß sich Anforderungen im Laufe der Zeit ändern werden. Somit Version Managment notwendig)
    5. Generalisierung (d.h. Probleme versuchen zu generalisieren, um diese auf einer höheren Ebene anzugehen)
    6. Inkrementelles Vorgehen (Incrementality / schrittweises Vorgehen ermöglicht einen evulotionären Entwicklungsprozess bei dem ein frühes Feedback die initialen Anforderungen erreicht)

2. Programmieren im Kleinen und im Großen, Unterschiede

- Anforderungen im Kleinen bekannt
- Programme können von Einzelnen geschrieben und entworfen werden
- Bei großen Anwendungen ist Anforderung nicht mehr trivial
- Präzise Spezifikation wird notwendig und ein Modell der Andwendung wird ausgearbeitet
- Es wird nun im Team gearbeitet

3. Engineering - systematischer Ansatz zum Problemlösen

- Sytematisches Herangehen
- Genaues Definieren des Problemes
- Von bestehenden Lösungen Lernen
- Wiederholende Muster muss man erkennen können
- Definieren von formalen Modellen, da diese Automatisierung fördern
- Entwicklung erfolgt mit standart Werkzeugen
- Techniken entwickeln mit denen Klassen von Problemen gelöst werden können

4. Warum ist Software Engineering so kompliziert?

- Nur unklare oder uneindeutige Spezifikationen möglich
- Semantik ist sehr komplex und es gibt keine mathematische Unterstützung

Teildisziplinen der Softwaretechnik:

SW-Entwicklung, SW-Management, SW-Qualitätssicherung und Wartung und Pflege

5. Geschichte des Software Engineering

- Früher Probleme zwischen nur einem User und Computer und keinen anderen Personen
- Somit waren Probleme einfach und überschaubar
- Benutzer spezifizierten das Problem
- Programmierer interpretierten die Spezifikation und programmierten danach

6. Probleme der Entwicklung von großen Softwaresystemen, Ursachen

- Nur unklare oder uneindeutige Spezifikationen möglich
- Semantik ist sehr komplex und es gibt keine mathematische Unterstützung
- 67 % allein an Wartungskosten

7. Lebenszyklus eines Softwaresystems

- Entwicklung und Evolution eines Software Systems wird systematisch nach einer bestimmten Methode umgesetzt
- Entwicklung und Evolution eines Software Systems besteht aus wohldefinierten Phasen
- Jede Phase hat wohldefinierte Anfangs- und Endpunkte, welche klar den Übergang zur nächsten Phase definieren

8. Wasserfallmodell des Lebenszyklus eines Softwaresystems

- Analyse – Design – Implementation und Modultest - Inegration und Systemtest – Inbetriebnahme und Wartung

9. Phasen des Wasserfallmodells

- Anforderungsanalyse und Spezifikation (Was ist das Problem?)
- Design und Spezifikation (Wie kann man es lösen?)
- Coding und Modultests (Funktionsrumpfe werden programmiert)
- Integration und Systemtest (Alle module werden zusammengesetzt)
- Übergabe (GoLive) und Wartung (Änderungen nach der Kundenübergabe)

10. Programmiersprachen und SE (Einfluss)

- Zentrale Werkzeuge bei Entwicklung
- Sollten Modularität, Software Architektur, seprarate und unabhängige Kompilierung von Modulen, GUI Design und von Implementierung unabhängige Spezifikation ermöglichen

Grund für Datenbanken:

Datenunabhängigkeit, d.h. Verwendung von Daten, ohne die zugrundeliegende Datenrepräsentation kennen zu müssen

➔ *Trennung von Spezifikation und Implementierung*

11. Datenbanken und SE (Einfluss)

- Große strukturierte Objekte speicherbar (Quellen)
- Große unstrukturierte Objekte speicherbar (Oject Programs)
- Configuration und version managment einfach möglich
- Datenbankschema's sind gleichzeitig Dokumentation

12. Management und SE (Ziele, Struktur)

- Technisches Managment (Kostenschätzung,Projektplanung, Ressourcenplanung, Arbeitseilung, Projekt Überwachung)
- Personalmanagment (Motivation, Anstellung, Auswahl der richtigen Subjekte)

2.2. Eigenschaften Software

Was ist Software ?

"Sammelbezeichnung für Programme, die für den Betrieb von Rechensystemen zur Verfügung stehen, einschließlich der zugehörigen Dokumentation" (Brockhaus-Enzyklopädie)

13. Besondere Eigenschaften der Software als Produkt

- Leute denken Software wäre leicht anpassbar, was falsch ist
- Bei Entwicklung liegt Fokus auf SE, bei Produktion in anderen Branchen
- Ein Produkt ist nicht nur das was der Kunde erhält, sondern auch die Anforderungen, Design, Sourcen und Testdaten
- Eigenschaften von Software
    1. Korrektheit
    2. Wiederverwendbarkeit
    3. Zuverlässigkeit
    4. Wachstum/Evolution
    5. Robustheit
    6. Portabilität
    7. Effizienz / Leistung
    8. Verständlichkeit
    9. Benutzerfreundlichkeit
    10. Interoperabilität
    11. Verifizierbarkeit
    12. Produktivität
    13. Wartbarkeit
    14. Pünktlichkeit/Aktualität
    15. Korrigierbarkeit
    16. Sichtbarkeit

14. Korrektheit von Software

- Ein Programm hält sich an die Spezifikation und deren Funktionen
- Daher muss Spezifikation vorhanden und korrekt sein
- Korrektheit ist also eine mathematische Eigenschaft, welche die Beziehung zwischen Spezifikation und software kennzeichnet

15. Zuverlässigkeit von Software

- Das Programm macht das was es soll
- Zuverlässigkeit: relatives Qualitätsmerkmal -> inkorrekte Software kann trotzdem zuverlässig sein, geringes inkorrektes Verhalten kann toleriert werden

16. Robustheit von Software

- Ein Programm ist robust, wenn es sich selbst unter Bedingungen„vernünftig“ verhält, mit denen nicht in der Anforderungsspezifikation gerechnet wurde (z.B. inkorrekte Eingabedaten, Fehlfunktion der Hardware).
- Robust gegen erwartete Ereignisse, wie Fehleingaben gehört zur Korrektheit
- Rubustheit hier heißt robust sein gegen unerwartete Ergeignisse, außerhalb der Spezifikation

17. Leistung von Software

- Ökonomisches Verhalten heißt das ein System effizient arbeitet
- effizient, wenn es die Ressourcen des Computers wirtschaftlich ausnutzt.
- In Bezug auf Geschwindigkeit und Speicherbedarf
- Über Monitoring messen um Bottlenecks zu entdecken
- Modell analysieren ist zwar billig aber unsauber
- Modelle simulieren ist teuer aber sauber
- Leistungstests müssen nach der Herausgabe der Erstversion abgeschlossen werden, nicht vorher, da sich sonst einiges ändern kann

18. Benutzerfreundlichkeit von Software

- Ein Programm ist … wenn seine Nutzer es einfach zu bedienend empfinden (subjektiv)
- Dabei muss auf unerfahrene (Menüs) und erfahrene Benutzer (commandline) geachtet werden
- Eingebettete Systeme haben kein UI
- Wichtig ist eine Standartisierung des User Interfaces, um Wiedererkennung zu fördern

19. Verifizierbarkeit von Software

- Ein Softwaresystem ist verifizierbar, wenn seine Eigenschaften einfach geprüft und verifiziert werden können
- Formale Analyse, Tests, Tracing und Debugging…

20. Wartbarkeit von Software

- Wartung nicht als Verschleißbeseitigung zu sehen, vielmehr als Änderungen
- Änderungen an Software ist extrem teuer und macht 60 % aller Kosten aus
- Software Evulotion ist die Folge
- Vorausdenken ist erforderlich! (Anticipation of changes)
- Kategorien:
    1. korrigierende Wartung
    2. Anpassende Wartung
    3. vervollkommende Wartung

21. Korrigierbarkeit von Software

- Fehler müssen leicht behebbar sein ohne großen Aufwand betreiben zu müssen
- Stellt auch ein Hauptdesignziehl dar
- Anzahl an Teilen verringern und möglichsts Standartteile verwenden, da diese wenig Fehler enthalten

22. Erweiterbarkeit von Software (Wachstum, Evolution)

- Software Evolution, um neue Funktionen hinzuzufügen und alte zu ändern
- Erweiterbarkeit begünstigt durch Modularisierung

23. Wiederverwendbarkeit von Software

- Gibt an, wie einfach es ist, Teile des SW-Systems (evtl. mit geringen Änderungen) in anderen SW-Systemen wiederverwenden zu können
- Libs schaffen, welche man wiederverwenden kann
- Software ReUse ist trotzdem selten
- Wiederverwendbare Komponenten sind auch viel teurer als die die es nicht sind

24. Portabilität von Software

- Software kann in verschiedenen Umgebungen laufen (Hardware, Software, OS)
- Es sollte daher eine minimale Konfiguration erfordern
- Strafe für Portabilität ist schlechteres Laufzeitverhalten, da nicht mehr so optimiert

25. Verständlichkeit von Software

- Ein Programm ist verständlich, wenn sein Verhalten vorhersehbar ist
- Intern:
    1. Wie verständlich ist das SW-System?
    2. Einfluss auf Erweiterbarkeit und Verifizierbarkeit
- Extern:
    1. SW-System ist verständlich, wenn es vorhersagbares Verhalten aufweist
    2. Verständlichkeit ist Teil der Benutzerfreundlichkeit

26. Interoperabilität von Software

- Möglichkeit eines Systems mit andern Systemen zu koexistieren und zu kooperieren
- Offene Systeme haben standartisierte Schnittstellen, die Kommunikation von Systemem verschiedener Hersteller ermöglichen

27. Produktivität der Software-Entwicklung

- Zeigt an, wieviel Qualität in welcher Zeitspanne hergestellt werden kann
- Effizienz heißt, das ein gutes hochqualitatives Produkt in kurzer Zeit hergestellt werden kann
- Produktion wiederverwendbarer Systeme senkt die Effizienz, da dies aufwendiger ist
- Produktivität ist schwer zu messen

28. Möglichkeit der rechtzeitigen Lieferung des Produktes (Pünktlichkeit / Aktualität)

- Notwendig sind dafür Vorsichtige Kostenberechnung, Aufwandsabschätzung und Planung, Milestones, Projektmanagment mit computergestützten Projektmanagmenttools

- Probleme sind das einschätzen von Zeitaufwand, Produktivität und setzen sinnvoller Milestones
- Anforderung vom Kunden wächst schneller, als diese Umsetzbar ist
- Lösung der Probleme durch inkrementelle Entwicklungsmethoden
- SW-Krise: SW-Produkte verspätet und fehlerhaft ausgeliefert

29. Sichtbarkeit

- Jeder Schritt wird klar dokumentiert
- Anforderung und Spezifikation
- Einzelne Schritte und der Status des Projektes sind jederzeit verfügbar
- Externe Qualität (Präsentation des Status)
- Interne Qualität (ermöglicht den Programmieren eine genauere Abschätzung und Aufteilung ihrer Handlungen)
- = Pflege der Anforderungs-, Entwurfsspezifikation

2.2.1. Prinzipien der Softwaretechnik

30. Prinzip des Filtern und Trennen – Zerlegen von Komplexität

- Filtern und Trennen der relevantesten Kernpunkte
- Weglassen nicht benötigter Informationen
- Nur so kann auch Problem dekompositioniert werden (z.B. in Module)es ist oft leichter die Teilprobleme zu erfassen, als das Ganze
- Zerlegung auf Basis von:
  1. Zeit -> Zeitplan von Aktivitäten
  2. Qualitätsmerkmalen -> z.B. erst Korrektheit, dann Effizienz
  3. Sichten -> z.B. einerseits Datenfluss, andererseits Kontrollfluss
  4. Größe -> *Modularisierung*
  5. Verantwortlichkeiten -> Arbeitsverteilung auf verschiedene
  6. Personen mit verschiedenen Fähigkeiten

31. Prinzip der Modularisierung

- Dekomposition des Problems in einzelne Module
- komplexe Systeme können in kleine Stücke oder Bausteine = *Module* aufgeteilt werden -> System ist modular
- Parallel bearbeitbar und testbar
- Kommunikation über eindeutige Schnittstelle
- Vertikale Modularität (jedes Modul in verschiedenen Abstraktionslevels beschrieben)
- Horizontale Modularität (Systembeschreibung auf selben Abstraktionsniveau)
- Trennung von Funktionsspezifikation (ER-Diagramme für Daten, DF-Diagramme für Funktionen, Petri-Netze für Steuerung)

Modularisierung

Zerlegbarkeit eines Systems:
Unterteilen des ursprünglichen Problems in Teilprobleme, daraufhin rekursives Unterteilen der Teilprobleme (Top-Down) -> divide et impera

Zusammensetzbarkeit eines Systems:
Zusammensetzen des Systems aus elementaren, vorgefertigten Komponenten (Bottom-Up)

32. Prinzip der Abstraktion

- wichtige Dinge herausheben und Details ignorieren
- Einteilung in relevante, logische Teile
- möglicherweise viele verschiedene Abstraktionen ein- und derselben Realität:
  1. verschiedene Sichtweisen
  2. verschiedene Absichten
- *Modelle*: Abstraktionen der Realität

33. Prinzip des Vorgriff von Änderungen

- ein Prinzip, das Software stark von anderen Typen von Industrieprodukten unterscheidet
  1. neue Anforderungen werden gefunden
  2. alte Anforderungen werden aktualisiert

  -> Vorsorge für Veränderungen treffen (z.B. durch entspr. Entwurf)

- Werkzeuge zur Verwaltung der verschiedenen Versionen und Revisionen der Software und ihrer Dokumentation nötig
  - Problem des Konfigurationsmanagements

- Anforderungen die der Kunde irgendwann mit hoher wahrscheinlichkeit mal haben möchte oder Anforderungen, welche aus anderen Gründen, wie Änderungen im Umfeld etc. entstehen
- das System muss so programmiert werden, dass Änderungen leicht durchzuführen sind und nicht etwa das halbe System neu entworfen werden muss

34. Prinzip der Allgemeinheit / Allgemeingültigkeit

- wenn ein bestimmtes Problem gelöst werden soll, sollte man versuchen, ein allgemeineres Problem als dieses zu identifizieren und das allgemeinere Problem zu lösen (denn das wahrscheinlich eher wiederverwendbar)
- Ähnlichkeiten entdecken und Generalisieren, d.h. Standards nutzen können
- Erhöht Wiederverwendtbarkeit und senkt kosten
- Ohne Generality würden ähnliche Module immer wieder neu entwickelt werden müssen

35. Prinzip der Inkrementalität

- Schrittweises Vorgehen, um Überblick zu behalten und Fehler zu verringern
- Ist ein evolutionärer Prozess
- „Alles auf einmal“-Methode bei großen Projekten nicht durchführbar
- deshalb schrittweises Vorgehen
- frühzeitiges Feedback vom Kunden: wichtig, da anfängliche Anforderungen evtl. nicht stabil sind oder nicht völlig verstanden wurden

36. Warum wird Software so oft geändert?

- Anforderungen des Kunden wachsen
- Es werden Fehler gemacht
- Bei Spezifikation wurden Teile vergessen

2.3. Analyse

SE als Ingenieurarbeit (ingenieurgemäßes Herangehen):

1. Definiere klar das Problem, das es zu lösen gilt.
2. Entwickle Standardwerkzeuge, -technologien und -methodologien, um das Problem lösen zu können

- *Funktionsmodellierung: Datenflussdiagramme, Data Dictionary, Prozessspezifikationen*
- *Datenmodellierung: ER-Modell*
- *Ereignismodellierung: Endliche Automaten, Petri-Netze*

37. Ziele, Struktur und Prozess der Analyse

- Ziel ist es herauszufinden, was das eigentliche Problem ist
- Strukturierte (top-down Analyse eines Prozesses) oder OO-Analyse (buttom-up Analyse eines Objektes)
- Anforderungsanalyse (Machbar?, Kosten, Aufwand etc)
  1. Spezifikation der Hauptziele
  2. Informationsquellen beschaffen
  3. Anforderungsanalyse
  4. Abgrenzung des Problembereiches
  5. Festlegung der Beteiligten
  6. Inhaltsbeschreibung
  7. Use Case
  8. Prioritäten festlegen
  9. alternative Lösungen suchen
  10. Empfehlungen vergeben

38. Erfassung der Anforderungen

- Anforderungen werden in natürlicher Sprache festgehalten
- User Manuals gehen aus Spezifikation hervor
- System Testpläne sollten von Spezifikation abgeleitet werden
- Blue Box Tests (Test auf Spezifikationen, da noch kein Code)
- Kategorien von Anfroderungen
  1. *funktionale*: Was soll das neue System leisten?
  2. *qualitative*: Qualitätsfaktoren, die durch zu erfüllen sind
  3. *operationelle*: zu erwartende Bedingungen beim praktischen Einsatz

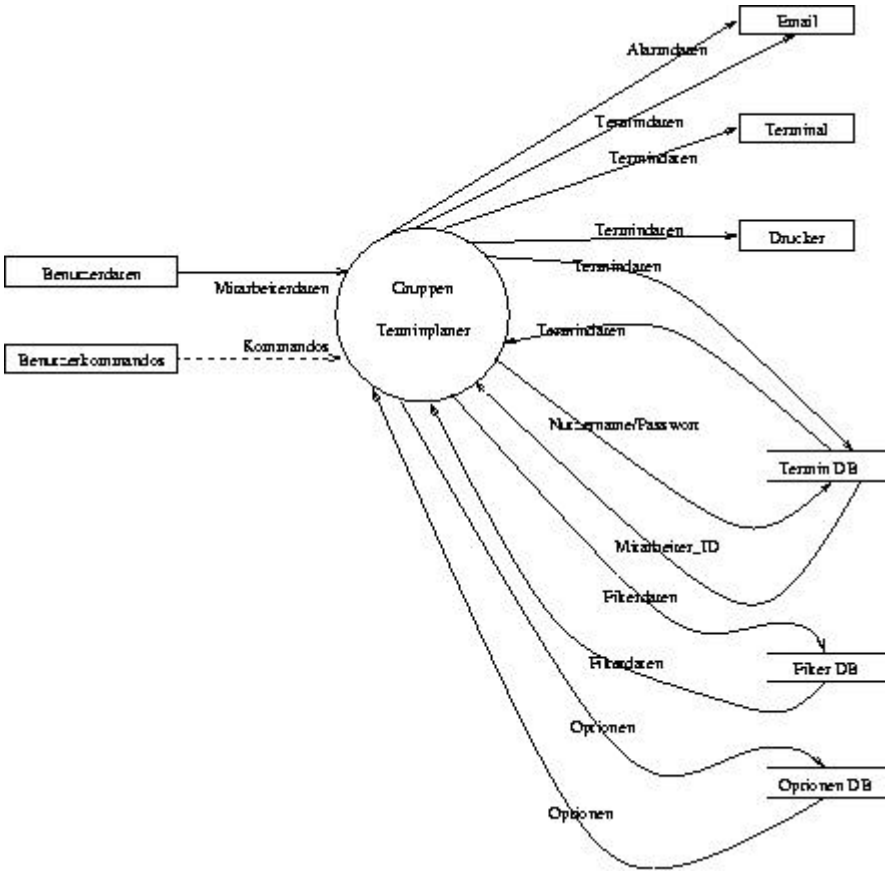
Phasen der Analyse
1. Ist-Analyse:
Finden der Anforderungen
Kenntnisse des existierenden Systems sammeln
Kundenwünsche präzisieren

2. Soll-Konzept:
Analysieren der Anforderungen
Kenntnisse analysieren
2 Verfahren: Strukturierte und Objektorientierte Analyse

2.3.1. Strukturierte Analyse

39. Strukturierte Analyse

- Strukturierung durch wiederholtes Zerlegen in Bestandteile geringerer Komplexität, bis elementare Bestandteile gefunden wurden, welche eindeutig und vollständig beschrieben werden können (Top-Down Analyse)
- Startet mit Verhaltensanalyse der Prozesse
- Dabei werden Funktionen in Gruppen aufgeteilt
- Resultat ist Hierarchie von Funktionen
- Datenflussdiagramme, Data Dictionary, Prozessbeschreibung
- modifizierte SA:
    - *Funktionsmodellierung*: statische Eigenschaften des Systemverhaltens
        - Datenflussdiagrammen
        - Data Dictionary
        - Prozessspezifikationen
    - *Datenmodellierung*: Eigenschaften der Daten und ihre semantischen Beziehungen
    - *Ereignismodellierung*: dynamische Eigenschaften des Systemverhaltens



40. Datenflussdiagramm - Konstruktion, Vorteile, Nachteile

- Zeigt wie Daten sich im System bewegen und wo sie Änderungen hervorrufen
- Top-Down Modell (iterative Dekomposition)
- ist Modell der Systemfunktionalität als gerichteter Graph
- Steuerung des Ablaufs nicht ersichtlich

41. Konsistenz eines Datenflussdiagramms

- Prozessbeschreibung (jede Funktion ist beschrieben)
- Data Dictonary (alle Datentypen und Datenflüsse sind definiert)
- Balanciertes DFD (I/O Datenflüsse des Vaterprozesses sind dem Kind-DFD bekannt)

42. Was steht alles in einem Datenkatalog?

- Zentraler Ort, wo alle Informationen über Datentypen und ihre Transformationen gespeichert sind
- Case Tools prüfen ob diese identisch sind
- Alle Eigenschaften der Daten sind im DD aufgelistet
- Data Elements sind gruppiert in Data Flow Records (Name, Record, Source, Destination, Description)
- Enthält Infos über Prozesse, externe Entities und alle Records

```
Termine       = Datum + Anfang + Ende + Status + Zweck
Termindaten   = Datum + Anfang + Ende + Alarmdaten + Status + Periode + Zweck
Anfang        = Zeit
Zeit          = Stunde + Minute
Datum         = Tag + Monat + Jahr
Adresse       = PLZ + Ort + Strasse
Gruppendaten  = Gruppenname
Optionen      = {Farbe + Terminart}
Terminart     = bestaetigt | geplant | {Gruppentermin_von_Gruppe_X} |
Minute        = Ziffer + Ziffer
Mitarbeiter_ID = Ziffer + Ziffer + Ziffer + Ziffer
Nutzername    = {Zeichen}
Passwort      = {Zeichen}
Nachricht     = {Zeichen}
PLZ           = {Ziffer}
```

2.3.2. Prozessspezifikation

43. Wie beschreibt man Prozesse in der strukturierten Analyse?

- für alle elementaren Prozesse werden Prozessspezifikationen angefertigt

> *elementarer Prozess: Prozess auf der untersten Ebene der Prozesshierarchie, nicht mehr weiter verfeinert (Blatt des Hierarchiebaums)*

- vier Möglichkeiten der Beschreibung
  1. strukturierte Sprache (Pseudo-Code)
  2. Entscheidungstabellen
  3. endliche Automaten
  4. logische Spezifikation

44. Was ist eine Entscheidungstabelle?

- Falls Condition wahr dann mache dies… (quadratische Möglichkeit der Belegung)
- kompakte und übersichtliche Darstellung von vorzunehmenden Aktionen oder Handlungen, die von der Erfüllung oder Nichterfüllung mehrerer Bedingungen abhängen
- besteht aus vier Komponenten:
  1. *Bedingungen*: logische Ausdrücke
  2. *Aktionen*: semantische Einheiten, werden später durch entsprechende Programmabschnitte implementiert

3. *Regeln*: Eine Regel bezeichnet eine Kombination von Bedingungen
4. *Aktionsanzeiger*: Ein Aktionsanzeiger repräsentiert eine Zuordnung zwischen einer Regel und einer Auswahl von Aktionen, die bei Gültigkeit der Regel durchgeführt werden

45. Was modelliert man mit einem endlichen Automaten, wie und warum?

- Kontrollaspekte, bei denen ein System verschiedene Zustände aufweist
- für Modellierung eines Kontrollflusses geeignete hypothetische Maschine, die Reaktionen (*Aktionen*) eines Prozesses auf eintretende Ereignisse beschreibt
- Darstellungsmöglichkeiten
  1. Zustandsdiagramm
  2. Zustandstabelle
  3. Zustandsmatrix

46. Endliche Automaten - Grenzen des Einsatzes, Vorteile, Nachteile

- Wird schnell zu komplex wenn zu viele Zustände
- Bei 20 Zuständen schon 1 Million verschiedene Belegungen möglich
- Manchmal hängt eine Aktion von mehreren Zuständen ab und nicht nur von einem
- Synchronisation nur schwer möglich (z.B. Erzeuger-Verbraucher Problem)

2.3.3. Petri Netze

Ist ein für Modellierung eines Systems konkurrierender/kooperierender Prozesse geeignetes Werkzeug.

47. Problem der Synchronisation und Petri-Netze

- Marken repräsentieren den Kontrollfluss, sind jedoch anonym, deshalb Inhalt der Marke unbekannt
- Bsp.: Marke in einem Puffer beschreibt das Vorhandensein einer Nachricht, aber nicht, ob diese wohl formuliert ist oder nicht -> eine Auswahl auf Basis des Inhalts der Marke kann nicht durch das Petri-Netz beschrieben werden
- wenn mehrere Transitionen geschalten werden können, keine Beschreibung von Prioritäten möglich
- keine Beschreibung von Zeitbeschränkungen möglich
  Bsp.: Wenn länger als eine Sekunde gewartet wird, soll eine Meldung generiert werden

48. Konstruktion eines Modells mit einem Petri-Netz

- Input Place -> Übergang (Transition) -> Output Place
- Marken werden zur Synchronisation verwendet
- Darstellungselemente:
  1. *Stelle*: Zustand eines Prozesses; repräsentiert eine Bedingung
  2. *Transition*: gesteuerter Zustandsübergang; modelliert ein Ereignis, welches durch das Schalten („Feuern“) der Transition ausgelöst wird
  3. *Pfeil*: zwischen Stelle und Transition und umgekehrt
  4. *Marke*: ihre Anwesenheit in allen vor einer Transition liegenden Stellen ist die notwendige Bedingung, dass die Transition schalten (feuern) kann
- Mutual Exclusion einfach umsetzbar, dafür Deadlocks möglich

49. Warum und wie werden Petri-Netzen erweitert?

- Marken werden durch Werte repräsentiert
- Transitionen sind Prädikate zugeordnet, die sich auf die Werte der Marken der Eingabestellen beziehen und den Übergang mitbestimmen (Transitionen können nur noch beim Vorliegen bestimmter Marken schalten)
- Transitionen sind Funktionen zugeordnet, die aus den Werten der Marken der Eingabestellen die Werte der Marken der Ausgabestellen berechnen
- Spezifikation von Prioritäten, wenn mehrere Transitionen geschalten werden können

2.3.4. OO Analyse

"Im Mittelpunkt stehen Objekte, welche Abbilder von Real-Welt-Objekten des zu analysierenden Problembereichs darstellen. Jedes Objekt besitzt eine eindeutige Identität (Objektidentität), bestimmte Eigenschaften (Attribute) sowie ein bestimmtes Verhalten (Methoden). Die Objekte kommunizieren miteinander durch das Senden von Botschaften. Wie ein Objekt auf erhaltene Botschaften reagiert, wird jeweils durch die in ihm vorhandenen Methoden definiert. Weitere wichtige Konzepte der OOA sind u.a.: Klasse, Datenkapselung, Vererbung und Polymorphismus (überladen)."

50. Objektorientierte Analyse

- Startet mit der Analyse der Objektstruktur
- Funktionalitäten wird in Klassen gruppiert und gekapselt
- Buttom-Up Analyse
- Objekt-Orientiertes Design benutzt ein konkretes HW und SW System, Architektur, Sprache und OO Lib

51. Konzepte der objektorientierten Programmierung

- Kapselung, Klassen, Nachrichten, Methoden, Vererbung, Überschreiben,Überladen von Funktionen
- Assoziation modelliert Beziehungen zwischen Objekten einer oder mehrerer Klassen
- Aggregation ist Sonderform der Assoziation, bei der die Objekte der beteiligten Klassen keine gleichwertige Beziehung führen, sondern eine Ganzes-Teile-Hierarchie darstellen (beschreibt, wie sich etwas Ganzes aus seinen Teilen zusammensetzt = part_of)
- Klassen können abgeleitet werden und daraus Objekte erstellt werden
- Statisches Binden (Code verbunden über Zeiger, Adressen von Methoden zur compilierzeit festgelegt) oder dynamisches Binden (Adressen der Methoden erst zur Laufzeit ermittelt - virtual Ausdruck notwendig, um Compiler zu zeigen, welches Objekt gemeint ist wenn mehrere polymorphe vorhanden, da erst zur Laufzeit über Zeiger verbunden wird)

52. Schritte der objektorientierten Analyse

- Initial problem statement (Ziele, Interface, Technische Aspekte, Features)
- Objekt identifikation, Klassenidentifikation mit Attributen, Beziehungsmodellierung
- Klassenhierarchie (Aggregationen und Assoziationen) definieren und Beschränkungen (constraints) festlegen
- Methoden und Dienste definieren, Funktionalität und OO-Schema definieren

53. Grammatische Inspektion

- Die Dokumente der Spezifikation werden grammatikalisch geparst
- Substantive werden Objekte, also Klassen (Objekte mit gemeinsamen Eigenschaften)
- Verben stellen später Methoden der Klassen dar oder auch Beziehnungen
- Und Adjektive sind Attribute von Objekten

54. Modellierung von Beziehungen

- Stellen semantische Relationen zwischen Objekten dar
- Aggregation (Is Teil von) um Hierachien zu erzeugen
- Spezialisierung/Generalisierung (Is-A Beziehung,d.h. Vererbung)
- Message-Flow-Structur (Sender und Empfänger für Nachrichten)

55. Modellierung von Restriktionen (Constraints)

- Einschränkungen die zu den Klassen definiert werden (Max-Werte oder Min-Werte…)

2.3.5. Risikoanalyse

56. Wozu gibt es die Risikoanalyse und wie wird sie durchgeführt?

- Gibt es Konkurrenz?
- Welche Technologien verwenden? (web, DMBS, Languages)
- Welche Markteinflüsse beeinflussen das Projekt?
- Anzahl zu erwartender Benutzer
- Zukunftstrends
- Problem vom Kunden verstanden?
- Problemdimensionierung
    1. Transactions / time
    2. angenommen Laufzeit für verschiedene Funktionen
- Welche Altsysteme müssen angebunden werden?

57. Wie identifiziert man Systemgrenzen und Akteure?

- Interne Systemgrenzen und externe (nicht unsere Aufgabe, aber evt. Schnittstellen dazu)
- Akteure ist all das, was mit dem System interagieren muss
    1. Wer benutzt das System (Mensch oder System)?
    2. Wer wartet es?
    3. Wer fährt es herunter?
    4. Wer bekommt Infos aus dem System?
    5. Wer füttert es mit Daten?
    6. Was passiert automatisch im System?

58. Wie identifiziert man Anwendungsfälle? (Use Cases)

- Beschreibt was ein Akteur machen möchte
- Was passiert, was wird ausgetauscht und welche Ereignisse gibt es dabei?

2.3.6. Modellierungshilfen

59. Was ist ein Szenario?

- Ist ein Spezieller Pfad durch die Use-Cases aus sicht des Users
- Primäre (Basic Path) und sekundäre Pfade (anderer nebenläufige Pfade)
- Kombiniert man alle Pfade erhält man einen Kompletten Use Case

60. Was ist der Umfang eines Projektes und warum ist er so wichtig?

- Szenarien sollten vollständig sein, um Fehler schnell erkennen zu können

61. Modellierung in UML - grundlegende Fragen

- **Basiert auf OO**
    1. data-centered Methoden wie ERD, Data-Flow-Diagramme. Statusdiagramme
    2. strukturelle Methoden
    3. scenario basierte Methoden (Verhaltensanalyse)
    4. UML (Objektmodelle aus den Use Cases)
- UML – Use Case Diagramm zeigt Akteure, Use Cases und die Beziehungen
- Strukurdiagramme für Klassen und Pakete (Gruppen von Klassen)
- Verhaltensdiagramme für dynamisches Verhalten und Systemaktivitäten
- Implementationsdiagramm
    1. Komponentendiagramm (relationen zwischen Programmeinheiten)
    2. Deploymentdiagramm (Kommunikation zwischen Komponenten)
- **Fragen**
    1. Wer sind die User des Systems?
    2. Welche sind die Hauptobjekte?
    3. Welche Objekte werden für welchen Use Case gebraucht?

62. Was beschreibt man alles in einem Klassendiagramm?

- Domain Class Modellierung
- Business Modell (definiere die wichtigsten Einheiten des Systems und ihre Relationen)
- Statische Semantik (Klassenhierarchie beschreibt technische Struktur der Problemdomäne) kann auf Business Modell basieren
- Ein Klassendiagramm enthält alle Klassen und Beziehungen (Generalisierung, Assoziierung, Aggregation, Komposition) und stellt somit auch die Hierarchie dar

63. Wie modelliert man ein GUI (Graphical User Interface)?

- Erst wird Handbuch geschrieben, dann der Code
- Dabei werden parallel GUI's entworfen, die bei Use-Cases verwendet werden können
- Window Navigation Diagramme zeigen Relationen zwischen Fenstern

64. Wozu gibt es die Robustness-Analyse? !!!

- Hier werden drei Arten von Objekte eingeführt
    1. Boundary Objects (Akteure welche mit System kommunizieren)
    2. Entity Objects (Objekte des Domänenmodells – Hauptentities)
    3. Control Objects (Verbinden Boundary und Entity Objekte)
    4. Das Robustness-Diagramm ist die Schnittstelle zwischen dynamischem und Statischem Teil eines Projektes
    5. Dabei gehört zum dynamischen Teil die Use Cases und Sequenzdiagramme und zum statischen das Domain Modell und das Klassendiagramm

6. Durch das Robustness-Diagramm wird es einfacher den Überblick zu halten

65. Warum und wie konstruiert man Sequenzdiagramme? !!!

- Visuallisiert die zeitlichen Abläufe und zeigt, was parallel passiert und in welcher Reihenfolge

66. Was modellieren die Kollaborations- und Zustandsdiagramme? !!!

- Kollaborationsdiagramme zeigen wie einzelne Module zusammenarbeiten
- Statusdiagramme dokumentieren die jeweiligen internen Stati des Systems

2.4. Spezifikatioen

Softwaretechnik, Software Engineering:
"Bezeichnet das geplante, systematische Vorgehen bei der Softwareentwicklung unter Anwendung von Methoden, Verfahren und Softwarewerkzeugen mit dem Ziel, qualitativ hochwertige Softwareprodukte wirtschaftlich herzustellen und zu nutzen."

67. Welche Rolle spielen Spezifikationen in der Software-Entwicklung?

- Ist eine Referenz bei der Implementation
- Ein Produkt ist nur gelungen, wenn es sich an der Spezifikation hält

68. Welche Eigenschaften sollte eine Spezifikation haben?

- Enthält eine Beschreibung von dem, was die Implementation enthalten muss
- Spezifikation, Anforderungsspezifikation, Design Spezifikation und Modulspezifikation
- Spezifikation ist Referenzpunkt wärend der Produktwartung
- Qualitätsmerkmale einer Spezifikation
    1. Konsistenz (keine Widersprüche)
    2. Komplettheit (alle benötigten Anforderungen)
    3. Klar und deutlich (alle benutzten Wörter erklären und definieren)
    4. eindeutig und verständlich

69. Wie unterscheidet sich eine operationelle von einer deskriptiven Spezifikation?

- Operationelle Spezifikationen beschreiben die erwünschten Verhaltensweisen
- Deskriptive Spezifikationen beschreiben die erwünschten Eigenschaften (beschreibt also das Was aber nicht das Wie)
    1. Data Flow Diagramme beschreiben Operationen aber nicht die benutzten Datenstrukturen oder Relationen
- Ein Prototyp ist ein operationales Modell

70. Wie formal werden Spezifikationen formuliert?

- Informal (normale Sprache), semiformal (präzise Syntax aber unpräzise Semantik) oder Formal präzise Syntax und Semantik)

71. Was ist die Verifikation einer Spezifikation?

- Funktionalität, Komplettheit und Konsistenz einer Spezifikation sind zu prüfen
- Überwachung des dynamischen Verhaltens des spezifizierten Systems
- Analyse der Eigenschaften des Systems

72. ER-Modell

- Entities (Objekttypen), Beziehungen und Attribute (Objekteigenschaften)

73. Logische Spezifikation

- = FOT (first-order-theory)
- Beschreiben in Form von Formeln und Logischen Ausdrücken
- Variablen, Konstanten, Funktionsprädikate,Quantifizierungen u.s.w.
- Immer Boolsches Ergebnis
- Pre- und Postconditions
- Vorteil ist, wenn Logische Spezifikation korrekt sind auch alle Ableitungen davon korrekt

74. Logisches Prototyping - Methoden, Probleme

- Interpreter für Operationale Spezifikation
- Damit aber nur das Verhalten prüfbar und nicht die Eigenschaften
- Besser logische Sprache wie Prolog umd FOT zu approximieren

75. Algebraische Spezifikation - wie beschreibt man die Syntax?

- Elemente (Char, Nat, Bool) und Operationen (Funktionen der Algebra wie new, append, add, lenght, isEmpty…)

76. Algebraische Spezifikation - wie beschreibt man die Semantik?

- Axiome der Algebra, welche immer Wahr sein müssen

77. Algebraische Spezifikation - Probleme

- Unvollständig, Überspezifizierung, Widersprüchlich, Redundanz

2.5. Entwurf

„Ziel: Umwandlung der während der Analysephase gewonnenen Anforderungsspezifikation in die Architektur des künftigen SWProdukts.“
Dient später als Vorlage für Implementierung…

- *Strukturdiagramme*
- *allgemeine Transformationsstruktur*
- *Transaktionsstruktur*

78. Entwurf - Ziele, Struktur und Prozess

- Ziel ist Spezifikation des Wie (Also How to Solve?)
- Das Design bringt die Anforderungsspezifikation in Vorlagen für Implementierung
- Entscheidungen über HW und SW Plattform

- Softwarearchitektonische Entwurfsphase (grundlegende Komponenten und Muster)
- Detailierter Entwurf der Module (verfeinern der Komponenten – nur Spezifikation)

79. Auf welche möglichen künftigen Änderungen soll im Entwurf vorgegriffen werden?

- Änderungen im Algorithmus, Datenrepräsentation, Perepheriegeräte oder sozialem Umfeld

80. Was sind Patterns?

- Sich wiederholende Muster, welche erkannt werden müssen, um Templates einsetzen zu können
- Pattern-Beschreibung enthält immer Kontext, Problem und Lösung

81. Was ist eine Softwarearchitektur?

- beschreibt die Struktur des SW-Systems durch Systemkomponenten (z.B. Unterprogramme, abstrakte Datentypen, Klassen) und ihre Beziehungen untereinander
- Besteht aus Komponenten, der Struktur des Systems und der Kommunikation zwischen den Komponenten
- Entwurf der Architektur setzt viel Intuition und Erfahrung voraus
- Motivation:
  - Beherrschung der Komplexität des SW-Systems
  - Arbeitsteilung im Rahmen eines Teams
  - Wiederverwendbarkeit
  - einfache Wartbarkeit

82. Welche Patterns werden in der Softwarearchitektur benutzt (Beispiele)?

- Schichtenarchitektur (wie Schichtenmodell von TCP/IP)
- Pipes und Filters Architektur (Scanners, Parser, Semantikanalyse, Optimierung)
- Broker Architektur (Kommunikationsbasiert für verteilte Systeme)
- Model-View-Controller Architektur (interaktive Applikation mit Modell, View und Events
- Presentation-Abstraction-Control (Präsentation,Abstraktion und Steuerung der Agenten)

2.5.1. Modularisierung

Modul (Programmbaustein)
- ist logische Einheit mit klar abgegrenztem Aufgabenbereich
- Kommunikation mit anderen Modulen nur über Ex-/Importschnittstelle
- Geheimnisprinzip – Innere Funkionen und ADT's für Umwelt verborgen
- austauschbar durch anderes Modul mit gleicher Exportschnittstelle
- kann getrennt von anderen Programmteilen entwickelt werden
- positiv für Wartbarkeit, Wiederverwendbarkeit ,Testbarkeit und Portabilität

83. Modularisierung - USES, IS_USED, IS_COMPONENT_OF

- Modul ist wohl definierte Komponente eines Softwaresystems

- Funktionale Module (Leistung) oder ADT-Module / Datenobjektmodule
- USES Relation für ausgehende Kanten
- IS_USED für eingehende Kanten
- Mit IS_COMPONENT_OF oder CONSISTS_OF wird Hierarchie gebaut

84. Wie kommunizieren Module (Klassifikation von Moduln)?

- Über Schnittstellen (Variablen, Funktionen etc), welche explizit als Schnittstelle gekennzeichnet werden (exportiert)
- Export (Dienst anbieten) und Import (Dienst anfordern bzw. einfügen)
- Vorteil ist die Abstraction der Module, da Details verschlossen bleiben
- Solange das Interface nicht geändert wird, verursachen Änderungen im Modul keinen Mehraufwandt

85. Schnittstelle - EXPORTED, IMPORTED, Entwurf

- Eine Schnittstelle sollte nur das nötigste so kurz wie möglich anbieten
    1. sonst zu komplex und unverständlich
    2. sonst Fehleranfälligkeit in Implementierung
    3. zu viele Informationen könnten missbraucht werden

86. Notation des Modulentwurfs

- Keine Standartisierte Notation bis jetzt
- Jede Notation ist so Formal wie es die Syntax ist
- die Semantik der Exports wird aber nicht formal spezifiziert
- Textnotation beinhaltet Funktionsrümpfe und Beschreibung in natürlicher Spache
- Graphische Design-Notation zeigt ein Diagramm mit Relationen (USES) und CONSIST_OF Beziehungen

87. Was ist ein generisches Modul oder Unterprogramm? !!!

- Generische Module enthalten nur Strukturinformationen
- Anwendungsspezifische Datenfelder werden vom Anwender dazugefügt

88. Schrittweises Vorgehen und Verfeinern - Vorteile, Nachteile

- Schrittweise Top-Down Verfeinerung beliebteste Methode
- In jedem Schritt wird Problem in Unterprobleme zerlegt
- Subsolutions werden einfach zusammengelinkt über Kontrollstrukturen
- Decomposition Tree zur Darstellung
- **Probleme**
    1. bei großen Programmen schlecht umsetzbar
    2. führt nicht immer zur besten Lösung
    3. Entwurf ist ein kreativer Prozess, der so nicht umgangen werden kann
    4. entdecken anderer Lösungen bleibt so oft auf der Strecke
- **Nachteile:**
    1. Unterprobleme werden isoliert betrachtet, deshalb
        1. keine Generalisierung möglich
        2. kaum eine Wiederverwendbarkeit
        3. keine Vereinigung mehrerer Subproblems
    2. kein Information Hiding, da keine Kapselung
    3. falls sich was in Struktur ändern, muss alles überarbeitet werden

89. Bearbeitung von Anomalien - Ausnahmen

- Unerwartete und unvorhergesehene Umstände müssen abgeschirmt oder abgefangen werden (Exeptions)
- Ein Modul sollte bei einer Anomalie ein Exeption-Signal an den Client senden, der einen Exception Handler aufruft (z.B. für Overflow, Div by Zero etc.)
- Try {…} catch (EXEPTION) { //Exeption bearbeiten}

2.6. Verifikation und Tests

- *Motivation*
- *Testfälle und Testmethoden*
- *black-box-Testen*
- *white-box-Testen*

90. Verifikation, kleine Programme, große Systeme, Ansätze

- Kleine Programme durch mehrere Beispielanwendungsfälle
- Mit dem Verhalten des Programmes experimentieren = Testen
- Das Programm analysieren = Korrektheit herleiten
- Problem ist das die Verifikation auch zu verifizieren ist

91. Testen, Problem von Kontinuität

- Das System wird in repräsentativen Fällen getestet
- Es ist nicht mögliche alle Zustände zu testen
- Problem der Kontinuität ist, dass Software nicht weiß, wie Ordnung der Werte bei Vergleichen semantisch korrekt ist

92. Theoretische Probleme des Testens, vollständiges Testauswahlkriterium

- Konsistenz und Vollständigkeit der Test sind nicht machbar

93. Empirisches Testen

- Es wird mit verschiedenen Testreihen (Gruppen) verschiedene Use Cases getestet
- Da ein vollständiger Test nicht möglich, wird zum idealen Test Set approximiert
- Die Schwierigkeit besteht darin alle wichtigen Gruppen einer bestimmten Problemklasse zu finden

94. Unterschiede zwischen White-Box-Testen und Black-Box-Testen

- **White-Box** Tests benutzen interne Struktur der Software und ignorieren ggf. die Spezifikation
- **Black-Bock-**Tests basieren ausschließlich auf der Spezifikation, da kein Wissen über Code oder Entwurf vorhanden ist

95. Abdeckung von Anweisungen, Kanten, Bedingungen, Pfaden

- Abdeckung von Anweisungen (Ein Fehler kann nicht entdeckt werden, wenn der Teil des Programms den Fehler enthält, der nicht ausgeführt wird)

- Deshalb sollte jede elementare Anweisung überprüft werden

- Abdecken von Kanten (jede Kante des Kontroll-Fluss-Graphen muss einmal traversiert werden, was nicht immer geht.. z.B. Loop Condition besteht aus mehreren Konjunktionen)
- Abdecken von Bedingungen (Jede Kante des Kontroll-Fluss-Graphen wird traversiert und dabei jeder mögliche Wert für alle Bedingungen eingesetzt)
- Pfadabdeckung (alle Pfade von Anfangszustand zum Endzustand im Controll-Fluss-Graph werden traversiert. Meistens sind dort zu viele Pfade aufgrund vieler Loops im Programm. Jede Iteration ist ein extra Pfad!)

96. Syntaxgesteuertes Testen, Testen von Grenzwerten

- Syntaxgesteuertes Testen heißt, das für jede Produktion der BNF ein Testfall bearbeitet wird, d.h. alle Grammatikgesetze werden überprüft

97. Testen im Großen, Bottom-Up-Integration und Top-Down-Integration, Vorteile, Nachteile

- Module Tests zum verifizieren der korrekten Implementierung
- Integration Tests verifizieren die Zusammenarbeit der beteiligten Module
    1. Big-bang – gar kein integriertes Testen
    2. Incremental – einfacher Fehler zu finden
- System Tests prüfen das gesamte System mit allen Modulen
    1. bottom-up (USES Hierarchie)
    2. top-down (lower level Simulation)

98. Was ist Debugging?

- Aktivität des Suchen und Behebens von Fehlern, nachdem er entdeckt wurde

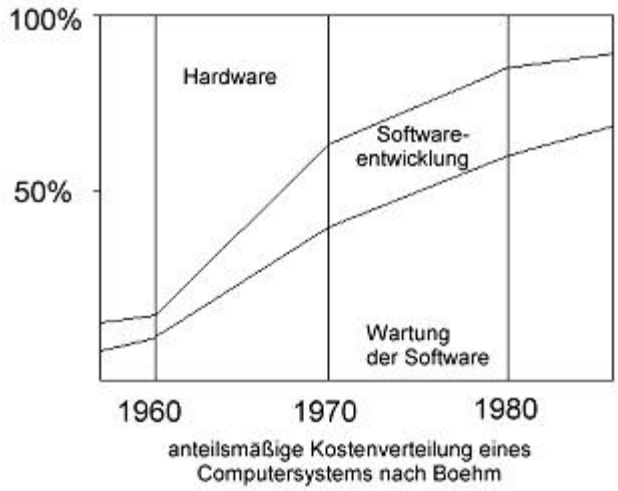
3. Softwaretechnologie II

1. Probleme der industriellen Softwareproduktion

- Softwareentwicklung ist ein Prozess von Planung und Management
- Implementierung wird begonnen, nachdem das Problem verstanden wurde
- Problem ist Softwareentwicklung vergeht nicht linear (Feedback Loops notwendig)

2. Lebenszyklus eines Softwareprodukts

- Aufgabenstellung -> Entwicklung -> Nutzung -> Aufgabenstellung -> …
- Deailiert z.B. durch Phasen des Wasserfallmodells
  - Analyse
  - Entwurf
  - Implementierung
  - Integration
  - Installation
  - Wartung



anteilsmäßige Kostenverteilung eines Computersystems nach Boehm

3.1. Software Produktionsprozess – Prozessmodelle

3. Code-and-Fix-Modell und Softwarekrise

- Früher ein Person pro Softwareprojekt
- Problem was gut verständlich und einfach
- Softwareentwicklung bestand nur im Coding
- 2 Phasen:
  - Programmierung
  - Fehler entfernen
- Probleme des Code and Fix Modells:
  - Nach mehreren Änderungen am Code wird Struktur unübersichtlich und schlecht
  - Es wird immer schwerer neue Änderungen einzubauen

Software Krise

- 1960 startet Entwicklung großer Software Systeme
- Grundproblem überall: overbuget und Zeitverzug
  - Zu lösende Probleme wurden nicht gut erkannt und verstanden
  - Mitarbeiter verbrachten mehr Zeit mit gegenseitiger Kommunikation als mit Codieren
  - Änderungen von ursprünglichen Systemanforderungen
  - Entwickler verliesen Projekte
- Original System Anforderungen wurden nachträglich geändert
  - Software Engineering wurde geboren, um nach einem Ausweg aus diesen Problemen zu suchen

Softwarekrise:
"In verschiedenen Erscheinungsformen uns ständig begleitende Erscheinung, die ausdrückt, dass der zu leistende Aufwand für Softwareherstellung und -betrieb die dafür zur Verfügung stehenden Kräfte übersteigt oder in Kürze übersteigen wird."

3.1.1. Wasserfallmodell

4. Wasserfallmodell

1. Machbarkeitsstudie
2. Anforderungsspezifikation und –analyse
3. Entwurf
4. Coding und Modultests
5. Implementation und System Tests
6. Übergabe und Wartung

- Sequentielle Abfolge der Phasen

5. Machbarkeitstudie

- Kosten abschätzen und alternative Lösungen suchen
- Ziel:
  - Problemdefinition
  - Alternative Lösungsvorschläge
  - Benötigte Ressourcen und Kostne

6. Spezifikation und Analyse der Anforderungen

- Notwendige Qualitätsmerkmale spezifizieren
  - Funktionalität
  - Performance
  - Benutzbarkeit
  - Portabilität …
- Beschreibt das „Was" umzusetzen ist, nicht das „Wie"
- Ziel: Anforderungsspezifikation (Pflichtenheft)

7. Entwurf

- Dekomposition des Systems in Module

- Entwurfsspezifikation beinhaltet
  - Beschreibung der Softwarearchitktur (Module, Beziehungen)
  - Abstraktionslevel (Uses, Is-Component Of…)
  - Detailierte Modulschnittstellen

8. Codieren und Testen

- Programmierung der Module
- Module Testen und Debuggen
- System Tests nach Implementation
  - Alpha-Tests (reelle Umstände aber vorgegebene Benutzer)
  - Beta-Tests (reelle Umstände und ausgewählte Kunden als Nutzer)

9. Lieferung und Wartung

- Wartung macht über 60% der Produktkosten aus
  - Korrigierende Wartung (errors)
  - Perfektionierende Wartung (Performance)
  - Adaptive Wartung (Erweiterungen oder Änderungen) -> Evolution

10. Vor- und Nachteile des Wasserfallmodels

- Kein Feedback zu Vorgängerphasen
- Starrheit der Phasen
  - Frühe Fehler haben schwere Folgen
  - Anforderungsspezifikation meistens unvollständig
  - Nur limitierte Informationen verfügbar
  - Kostenabschätzung und Planung kann nur nach einer gewissen Analyse stattfinden
  - Benutzer wissen oft nicht die genauen Anforderungen einer Applikation
  - Wasserfallmodell bezieht nicht Anticipation Of Changes ein
  - Dokumentbehafteter Prozess, da jede Phase tipparbeit voraussetzt

3.1.2. Weitere Modelle

11. Evolutionäres Modell

- „Machs doppelt“ Prinzip
- Erstversion eines Produktes ist ein Wegwerfprototyp
- Wird als Versuch betrachtet, um Machbarkeit und Anforderungen zu analysieren und zu verifizieren
- Problem:
  - Zeitloch zwischen Anforderungsdefintion und finaler Delivery des Produktes bleibt
  - Lösung des Problems: Inkrementelles Vorgehen

12. Inkrementelles Modell für die Implementierung

- Wasserfallmodell wird bis zum Entwurf angewendet
- Dann wird schrittweise vorgangen
- Schnittstellen, welche späteres Hinzufügen von Subsystemen erlauben
- Code-And-Tests + Integrate+Tests
- Jeder inkrementelle Schritt wird separat entworfen, codiert, getestet und integriert
- Die Schritte werden einer nach dem anderen entworfen
- Erst nach Feedback mit Kunden implementiert

13. Prototyping

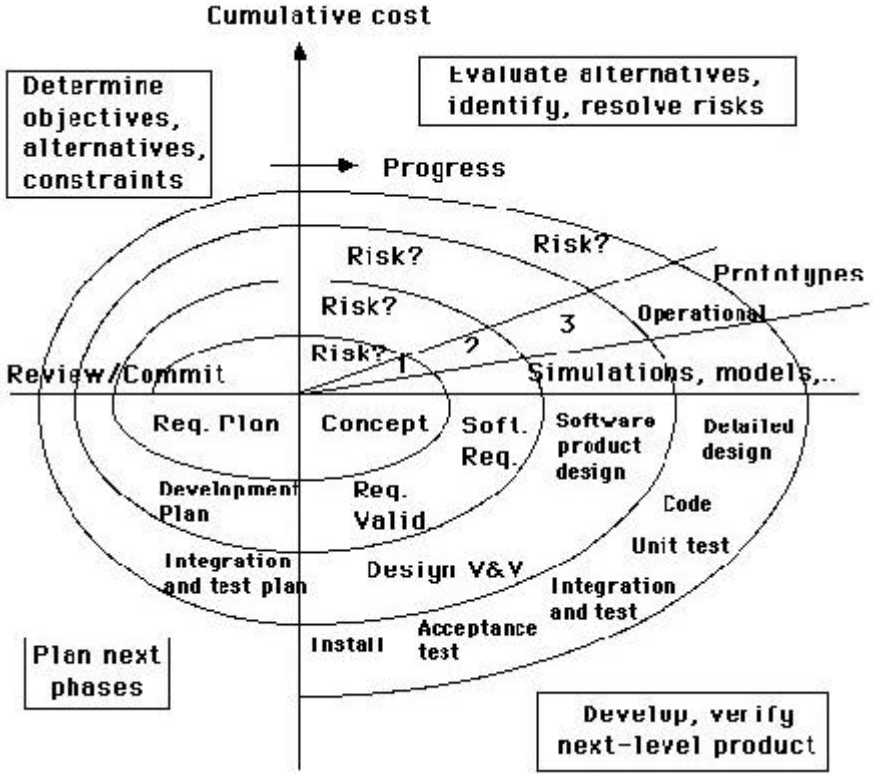
- Evolutionäres Prinzip um den Lebenszyklus zu strukturieren
- Display bzw. GUI enthalten
- Dummy Funktionen
- Gutes Werkzeug, um Anforderungen des Kunden zu verfeinern

14. Transformationsmodell

- Softwareentwicklung hier eine Folge von Schritten, welche Spezfifikation in Implementation überführt
- Transformationenen
  - Manuell
  - Semi-automatisch
  - Zwei Hauptschritte:
    - Anforderungsanalyse (und Verifikation dieser)
    - Optimierung (und Tuning an dieser)

15. Spiralmodell

- Ist ein Metamodell welches auf alle Modelle angewandt werden kann
- Identifizieren und Entfernen von Risiken schon beim Entwurf
- Es ist zyklisch und nicht linear wie das Wasserfallmodell
  - Stage 1: Ziele, Anforderungen und Alternativen identifizieren
  - Stage 2: Alterativen untersuchen, Prototyping und Simulationen
  - Stage 3: Entwicklung und Verifikation
  - Stage 4: Ergebnisse untersuchen und nächste Iteration planen
- Es findet also in jedem Druchlauf erneut eine Risikoanalyse statt
- Bei jedem Durchlauf entsteht neuer Prototyp



16. Bewertung von Prozessmodellen

- Wasserfallmodell: Dokumentationsbasiert
- Evolutionäres Modell: Inkrementell
- Transformationsmodell: Spezifikationsbasierend
- Spiralmodell: Risikobasierend
- Prototyping: Endnutzerbezogen
    - Unterstützt GUI Design
    - Vermindert Risiken wie Aufhalt an unnötigen Aspekten
    - Hift sich auf relevanten Probleme zu konzentrieren
    - 40 % weniger Entwicklungszeit und Quellinstruktionen

17. Softwaremethodologien - Vorteile, Nachteile

- Methode = ein Weg etwas zu tun
- Methologie = ein System von Methoden, unterstützt von einem Werkzeug
- Standarts: Methologien werden zu firmenweiten wiederverwendbaren Packeten zusammengefaßt
- Vorteile:
    - Führt den Programmierer durch alle Phasen
    - Lernt Unerfahrene an (Wie Problem systematisch Lösen?)
    - Standartisierte Problemlösungsstrategien
- Nachteil:
    - Fehlende formelle Untersuchungen
    - Verbrauchen viel Personal – aufwendig
    - Es braucht manchmal mehr zeit die Methologie zu verstehen, als das Problem

Comparison of Life Cycle Models		
Build-and-Fix	Fine for small programs that do not require much maintenance	Totally unsatisfactorily for nontrivial programs
Waterfall	Disciplined approach Document driven	Delivered product may not meet client's needs
Rapid Prototyping	Ensures that delivered product meets client's needs	A need to build twice. Cannot always be used
Incremental	Maximizes early return on investment. Promotes maintainability	Requires open architecture. May degenerate into build-and-fix.
Synchronize and-stabilize	Future user's needs are met. Ensures components can be	Has not been widely used other than in Microsoft.

	successfully integrated	
Spiral	Incorporates features of all the above models	Can be used only for large-scale products. Developers have to be competent at riskanalysis

Quelle: University Of California

18. Strukturierte Analyse und Entwurf

- Funktionsmodellierung: Datenflussdiagramme, Data Dictionary, Prozessspezifikationen
- Datenmodellierung: ER-Modell
- Ereignismodellierung: Endliche Automaten, Petri-Netze

Qualitätskriterien für Entwicklung:

Integration, Einsetzbarkeit, Adäquatheit, Erlernbarkeit Wiederverwendbarkeit , Werkzeugunterstützung und Wirtschaftlichkeit

19. Jacksons strukturierte Programmierung

- Jackson Program Design Methodology oder Jackson Structured Programming (JSP)
- Methode für Entwurf und Modellierung „im Kleinen“
- Entwurf beginnt mit Untersuchung von dem was bekannt ist
- In mehreren Iterationen wird ein Modell basierend auf Data-Structured Methoden erstellt
- aus der Beschreibung der Ein-/Ausgabe-Datenstrukturen und ihren Beziehungen zueinander wird direkt die Feinstruktur des Kontrollflusses abgeleitet

3.2. Konfigurationsmanagement

20. Konfigurationsmanagement

- Meiste viele Ausführungen eines Produktes erstellt
- Produkte unterliegen vielen Änderungen bei Wartung
- Softwarehaus muss jede Änderung nachvollziehen können
- Nur so Gewissheit, wie Fehler zu korrigieren sind und Entscheidungen zu treffen sind
- Release = Gruppe von Komponenten die zusammen erstellt werden
- Wie soll dies kontrolliert werden? Configurationmanagement
    - Version = Instanz eines Objektes
    - Configuration item = spezielle version einer Komponentengruppe
    - Baseline = eingefrorene Release, welche spezifischen Status repräsentiert
    - Variante = configuration item, was sich leicht unterscheidet
    - Change request = online Form, welche alle Änderungsnotizen festhält
- CM kontrolliert Änderungen und die Evulotion eines Produktes
- Probleme sind das Teilen von Komponenten und das Handhaben von Produktfamilien
-

21. Teilung von Komponenten

- Problem des mehrfachzugriffes auf ein gemeinsamen Pool von Komponenten
    - Mehrfachzugriffe müssen abgeglichen und synchronisiert werden
    - Andere müssen über Änderungen informiert werden
    - Es darf nicht zu Datenverlusten oder Inkonsistenten im Code kommen

22. Verwaltung von Produktfamilien

- CM verwaltet nicht nur Source Codes, sondern auch Doks, Testdaten und Manuals
- Problem ist, daß eine Komponente in mehreren Versionen exisitieren kann
- Lösungsmöglichkeiten
    - Jedes Familienmitglied besteht aus verschiedenen Versionen der Komponenten
    - Jede Familienmitglied beinhaltet eine eigene Kopie aller notwendigen Komponenten

23. Versionierung

- Zentrale shared database welche mit Projekt assoziiert wird
- Jeder Programmierer hat seinen privaten Entwicklerplatz, wo seine Zwischenversionen abgelegt werden, an denen er gerade arbeitet
- Das Ausgeben von Modulen nur über CheckIn / CheckOut Funktionalität

3.3. CASE Werkzeuge

24. CASE-Werkzeuge - Vorteile, Nachteile

- Computer Aided Software Engineering
- Unabhängig von Hardware, OS und Sprache
- Unterstützt Analyse und das Definieren von Datenflüssen
- Hift beim Entwurf bei Erstellung detaillierter Spezifikationen
- Hilft beim Programmieren und vereinfacht und unterstützt die Dokumentation in jeder Phase
- Hilft beim Projektmanagement und kontrolliert das Projekt
- CASE Tools garantieren, daß nichts vergessen wird
    - Jedes Datum mit Attributen
    - Jeder Prozess
    - Jede Beziehung zwischen Daten
- Datenanalyse ist in CASE Tools integriert
- Nachteile:
    - CASE Tools sidn Komplex – viel Einarbeitungszeit notwendig
    - CASE Tools sind Hilfsmittel, keine Lösungsgeneratoren

25. Reverse-Engineering

- Einige CASE-Tools unterstützen dies
- Es wird versucht, von Umsetzung zurück zu Spezifikation zu kommen
- Alles genau umgekehrt…

26. Auswahl eines CASE-Werkzeugs

- Komplette Integration von Analyse, Entwurf, Coding und Tests?
- Welche Standarts und Methologien werden unterstützt?
- Teamwork möglich?
- Client / Server Konzept?
- Wie benutzerfreundlich? (einfach zu nutzen mit GUI, Icons, Hilfesystem)
    - Arbeit sollte mit CASE nicht länger dauern als ohne

27. Architektur eines CASE-Werkzeugs

- Für jede Phase gibt es andere Editoren, welche für jede Phase Dokumente erzeugen
- Datenbank als DS sinnvoll da gegen inkonsistenten und Mehrbenutzerbetrieb unterstützt
- Relationales DBMS nicht effizient, da zu viele relationen zwischen Objekten und Objekte zu komplex (zu viele Joins notwendig, wenn ein Objekt geholt werden soll)
- Zentrales Data Dictonary welches Definitonen aller Daten und Attribute enthält
- OODB hat viele Vorteile (hebt Nachteile von relationalen DBMS auf)
    - Gespeicherte Objekte werden nicht durch Normalisierung gesplittet

3.4. Projektmanagement

Technisches Management:
Projektabschätzung, Projektplanung, Planung des Personalbedarfs, Aufteilung und Zuweisung von Aufgaben, Projektsteuerung und –überwachung

Personalmanagement:
Personal einstellen, Mitarbeiter motivieren, die richtigen Leute den richtigen Aufgaben zuweisen

28. Projektmanagement - Ziele und Probleme

- Planen, Abschätzen, Einteilen, Überwachen, Berichten -> Kontrolle
- Nutzen ovn PM Software
  - Prozess des Koordinierens aller Schritte
  - Während des gesamten Software Lebenszyklus
  - Projektmanager
    - Senior Systemanalyst
    - Large IS => einzelner Manager
    - Small IS => Programmierer selbst

29. Allgemeine Aufgaben des Managements

- Planen aller Aktivitäten (Aufgaben identifizieren und Zeit/Kostenabschätzung)
- Aufgaben verteilen (Team zusammenstellen)
- Organisatorisches (Projektarbeit strukturieren und einteilen)
- Überwachen der Prozesse (Führen, Beraten, Koordinieren)
- Kontrollieren der Arbeiten (Ergebnisse überprüfen …)

30. Planung eines Projekts

- Planung am Anfang einer jeden Phase
- Eine Aktivität benötigt verschiedene Ressourcen (Personal, Zeit, Geld)
- Ereignisse festlegen (milestones etc.)
- Planung am Ende einer Phase, um Kosteneinschätzung zu verifizieren

31. Methoden der Projektabschätzung

- Schwierigste überhaupt am Projektmanagement
- Projektgröße und benötigte Ressourcen verlaufen nocht propoprtional!
  - Kommunikation, Änderungen, Schnittstellen etc….
  - Graph mit (1/2) n (n – 1) Kanten (also quadratischer Aufwand)
- 3 Methoden
  - Quantitative unterstützt
  - Erfahrungsbasierte Methode
  - Constraintmethode

32. Methode der quantitativen Abschätzung

- Tabellen und Formeln zur Abschätzung benutzt
- Tabellen: Nummern und Typen der Dateien, Funktionen etc. als Anhaltspunkt welcher mit Produktivität dividiert wird
- D.h. es werden Zahlengewichte für einzelne Probleme gegeben

- Work * Experience / Productity = person/days

33. Methode der Abschätzung auf Basis der Erfahrung

- Basiert auf Erfahrung vorangehender Projekte
- Funktioniert nicht bei Großprojekten da zu komplex

34. Methode der Beschränkungen

- Projekt Anforderungen dienen als Basis

3.5. Scheduling

35. Zeitplan eines Projekts (Scheduling)

- Reihenfolge der Aufgaben festlegen
- Welche Aufgaben hängen von welchen Ergebnissen ab?
- Abhängigkeiten werden festgestellt, indem Aktivitäten in logische Sequenz gelegt werden
- Grantt Diagramme und PERT/CPM

36. Gantt-Diagramme

- Können schnell zu groß werden
  - Dekomposition: pro Team ein Plan, Pro Aufgabe ein Plan
- Nachteile:
  - Keine Abhängigkeiten dargestellt
  - Keine Personalhinweise oder Personentage angegeben

37. PERT/CPM-Diagramme

- Kritischer Weg – Methode
- Gerichteter Graph mit
  - Aktivitäten / Aufgaben als Kanten
  - Ereignissen als Knoten
  - Für Synchronistation (z.B. Abhängigkeiten) Dummy-Knoten
- Im Knoten wird Earliest Completion Time und Latest Completion Time eingetragen
  - So entstehen Buffer, welche zur Reallokation bei Verschiebungen benutzt werden
- Nachteile:
  - Wird schon für kleine Projekte sehr kompliziert
  - Kein Personenscheduling

38. Kritischer Weg

- Kritischer Pfad ist ein Weg welcher aus Knoten besteht bei denen kein Buffer vorhanden (ECT == LCT)

39. Kosten-Nutzen-Analyse

- Es werden die Kosten mit dem Nutzen verglichen
- Ökonomischen Nutzen ermitteln und mit alternativen Lösungen vergleichen
- Stragtegien:

- Payback Analyse
- Return and Investment Analyse
- Present Value Analyse

40. Pay-back-Analyse

- Wieviel Zeit ist notwendig, das investierte Geld wieder reinzuholen?
- Nachteil: Lässt kosten nach Payback Periode außer acht

41. Return-and-Investment-Analyse

- ROI = (Totaler Nutzen – Totale Kosten) / Totale Kosten
- Prozentuale Angabe welche angibt, wie Profitabel das Geschäft wäre
- Projekte müssen ein Minimum unterschreiten
- Nachteile: Durchschnittswerte als Grundlage

42. Present-value-Analyse !!!!

- Money today = money yesterday + Interest
- PV (present value)

3.6. User Interface Entwurf

43. Entwurf der Benutzerschnittstelle

- UI ist Maßstab nach dem Systeme von Nutzern bewertet werden
- Falls UI schwer benutzbar
  - Software wird verworfen
  - Es können schnell Fehler gemacht werden
- Heute notwendige Komponenten
  - Highres Graphic Display, Maus und verschiedene Schnittstellen für verschiedene Nutzerklassen
- GUI's sollten Fähigkeiten der individuellen User unterstützen
- Konsistenz und Hilfesystem

44. Konsistenz der Benutzerschnittstelle

- System Kommandos, Menüs und andere Oberflächen sollten
  - Gleiche Format besitzen
  - Parameter immer gleich übergeben
  - Untersysteme sollten sich ähneln
  - Nutzer sollte beim Erlernen eines Kommandos rückschlüsse auf Bedienung aller Anderen ziehen können

45. Eingebaute HELP-Unterstützung

- Vom Benutzerterminal erreichbar
- „How to get started?"
- Volle Beschreibung des Systems und wie es zu Nutzen ist
- Strukturierte Hilfe

46. Schablonen für Benutzerschnittstelle

- Es werden Metapher verwendet, welche einfach assoziierbar sind
  - Control Panel
  - Desktop…
  - Trash

47. WIMP-Schnittstelle

- Windows – Icons – Menues – Pointing (Mausklick)
- Vorteile
  - Einfach zu lernen
  - Viele Oberflächen zur Interaktion
  - Überschaubar und unkompliziert
- Nachteile
  - Keine Standarts
  - Schwer möglich sinnvolle Icons für abstrakte Komponenten zu finden

48. Systeme mit Menü

- Benutzer müssen nicht den genauen Befehlnamen wissen
- Weniger Tipparbeit
- System kann nicht in Fehlerzustand geführt werden
- Contextsensitive Hilfe

- Pull-Down und Pop-Up Menüs
- Nachteile:
    - Logische Verknüpfungen unmöglich ausdrückbar
    - Für Experten Menü langsamer als Commandozeile
    - Menühierarchie schnell komplex wenn viele Auswahlmöglichkeiten

49. Graphische Schnittstelle

- Grafiken werden zu Darstellung von Informationen verwendet
- Bilder sagen mehr als Worte
- Trends sind sichtbar

50. Textuelle Schnittstelle

- Kommandozeileninterpreter billig und einfach
- Kombinierende Kommandos erweitern Funktionalität und Aufrufkomplexität
- Externe Prozeduren und Programme können eingebunden werden
- Erfahrene Benutzer sehr schnell
- Nachteile:
    - Lernen aufwendig
    - Fehler in Kommandos möglich
    - Interaktion durch Tastatur (langsam)
    - Nicht für unerfahrene Nutzer sinnvoll

51. Entwurf von Fehlermeldungen

- Ersteindruch an Nutzer
- Schwer für unerfahrene Anwender
- Eigenschaften
    - Konsisten, Konstruktiv, Eindeutig und Präzise
- Nachricht sollte Möglichkeit zur Fehlerbehebung enthalten
- On-Line Hilfe sollte auftufbar sein

52. Anwendung von Farbmonitoren

- Ist zwar neue Dimension, nützt aber blinden oder Farbblinden nichts und es gibt auch keine Standarts
- Tips:
    - Nicht zu viele Farben nutzen (4-5)
    - Konsistenz bei Farbwiederverwendung
    - Farbanpassung sollte durch Nutzer einstellbar sein

3.7. Messen und Software-Metriken

53. Software-Metriken

- Messen von verschiedenen Aspekten, um diese verständlicher darzustellen
- Metriken für Produktivität
  - Versucht Voraussagen zu erleichten
  - Wie war die Produktivität bei vorangegangenen Projekten?
  - Wie kann vergangene Produktivität auf aktuelle Projekte extrapoliert werden?
- Helfen den technischen Prozess der für Produktion verwendetet wird und das Produkt selbst, zu verstehen
- Messungen finden statt um das Produkt zu verbessern
- Notwendig für Planung, Kosten -> Abschätzungen möglich

54. Gründe fürs Messen

- Qualität eines Produktes bestimmen
- Produktivität der Angestellten überprüfen
- Den Nutzen neuer SE-Tools und Vorgehensweisen untersuchen
- Baseline für Abschätzungen schaffen

55. Direktes und indirektes Messen

- Direktes Messen
  - Lines Of Code (LOC or KLOC)
  - Ausführungsgeschwindigkeit
  - Speicherverbrauch
  - Gemeldete Defekte / Zeitperiode
- Indirektes Messen
  - Funktionalität
  - Qualität
  - Komplexität
  - Wartung

3.7.1. Kategorien / Klassifikation von Metriken

56. Kategorien von Software-Metriken

- Produktivitäts-Metriken (Fokus auf SE Prozess)
- Qualitäts-Metriken (Wie nah ist Produkt an Kundenanforderungen)
- Technische Metriken (Fokus auf Produkt selbst, z.B. Modularität)
- Größen-Orientierte Metriken(Direkte Messwerte)
- Funktionsorientierte Metriken (Indirektes Metriken)
- Meschenorientierte Metriken (Informationen über die Arbeitnehmer die es produzieren)

57. Größe-orientierte Metriken

- Direkte Messungen welche mit Entwicklung korrespondieren
- Produktivität = KLOC / person-month
- Qualität = Fehler / KLOC
- Kosten = $ / KLOCK
- Dokumentation = Seitenanzahl / KCLOC

- Vorteil: einfach zu messen
- Probleme:
  - Nicht als bester Weg zum Messen aktzeptiert
  - Sprachabhängig
  - Planer muss LOC abschätzen, lange vor Analyse abgeschlossen ist

58. Funktionsorientierte Metriken

- Indirekte Messungen der Software und des Prozesses mit Fokus auf Programmfunktionalität
- Empirische Beziehungen basieren auf zählbaren Messungen
- Beispiel: (pro Messung ein Count und 3 Wichtungen, wie simple, average und complex)
  - Number Of Inputs
  - Number Of Outputs
  - Number Of Files
  - Number Of External Interfaces
- Ergebnis: Total Count

59. Function-Points-Methode

- FP = Gesamtanzahl * [0.65 + 0.01 Sum(1..14)(F_i)]
- F_i sind Faktoren zwischen 0 und 5, welche die Semantik und Komplexität des Problems beschreiben
- 0.65 und 0.01 sind empirische Konstanten
- 1 – 14 sind 14 verschiedene Aspekte wie z.B
  - F_1 : Benötigt das System Backup und Revovery
  - F_2 : Ist die Performance kritisch?
  - F_3 : Soll der Code wiederverwendbar sein?
  - …
- Produktivität = FP / Person-Month (Analog zu Size-Orienten direktem Messen)

60. Feature-Points-Methode

- Erweiterung der Function-Point-Methode in zusätzlicher algorithmischer Betrachtung
- Count-Total beinhaltet außerdem eine Abschätzung der Algorithmenkomplexität
- Sprachunabhängig und basiert auf Daten früh aus der Evolution
- Nachteil:
  - Subjektive Abschätzungen
  - keine dirkte physikalische Bedeutung
  - Daten schwer sammelbar sind

61. Metriken und Sprachen

- Abschätzungen wieviel LOF für einen Funktionspunkt durchschnittlich benötigt werden
- BSP:
  - Assembler (300)
  - FORTRAN (100)
  - Pascal (90)
  - Code Generatoren (15)

62. Argumente für Software-Metriken

- Ohne würde es keinen Anhaltspunkt geben, auf denen Verbesserungen entstehen können
- Antworten auf folgende Fragen möglich:
  - Welche Nutzeranforderungen ändern sich am häufigsten?
  - Welche Module im System sind am Fehleranfälligsten?
  - Wieviel Testzeit muss für jedes Modul eingeplant werden?
  - Wieviele Fehler kann ich beim Testen durchschnittlich erwarten?
- Schwierigste Part ist das Datensammeln

3.7.2. Messen von Softwarequalität

63. Wie kann die Qualität von Software definiert werden?

- Konformität zu vorher spezifizierter Funktionalität und Performanceanforderungen
- Qualität wird also über die Anforderungen gemessen
- Standards geben heute vor, welche Aspekte immer erfüllt sein müssen
- Implizite Qualitätsanforderungen auch wichtig (wie z.B. gute Wartbarkeit)

64. Faktoren der Software-Qualität

- KLOC (direktes Messen)
- Nutzbarkeit, Wartbarkeit… (Indirekte Messwerte)

65. Messen der Qualität

- Jede Softwarequalitätsmessung kann nur unvollständig sein
- Die Wortanzahl einer verbalen Beschreibung der Funktionaliät kann als Indikator für Komplexität benutzt werden
  - Umso mehr Wörter, desto schwerer zu Komplexer
- Wie viele Sprünge im Code? -> Wie ist der Testaufwand?
- Modulaufrufe anderer Module -> Wartbarkeit
- Zugriff auf globale Daten oder lokale Variablen geben Auskunft über Verständlichkeit des Codes
- Anzahl an nicht standartisierten Features -> Portabilität

3.7.3. Testen

66. Testplan

- Für Qualitätskontrolle wird Testplan benötigt
- Kontrakt zwischen Kunden und Entwickler und Entwickler und Qualitätssicherungsteam

67. Audit ????

- Qualitätssicherungsleute prüfen das Tests sinnvoll durchgeführt

68. Software-Reviews

- Reviews werden an verschiedenen Punkten in Entwicklungsprozess gemacht
- Sollen Fehler aufdecken, um diese so früh wie möglich eliminieren zu können
- 50-60 % aller fehler entstehen schon im Entwurf
  - Reviews reparieren bis zu 75 % dieser Fehler

69. Kosten von Fehlern in der Software

- Steigen exponentiell mit Fortschreiten der Entwicklung
- Kosten während Entwuf = 1, während Tests 15 und nach Release 60-100

70. Fehlervermehrung im Lauf der Softwareentwicklung

- Fehler aus jeder Phase werden an die nächste weitergegeben
- In dieser entstehen nun wieder neue Fehler
- Ohne Reviews kaum Fehler in Spezifikation und Entwurf entdeckt
- Mit Reviews werden ¾ der Fehler erkannt und behoben, da schon im Entwurf „getestet“ wird

71. Qualitätszertifikat

- ISO 9001 ist nicht industriebezogen
- ISO 9003 von 9001 für Software abgeleitet
- 20 Hauptmerkmale werden behandelt, wie
  - Document control, Design control, Process control
  - Training, Contract review, Quality system
  - …
- Zertifikat welches Kunden versichert, daß Firma nach Standarts arbeitet

72. Software-Metriken und Software-Qualität

- Immer unvollständig
- Teilweise individuelle Einschätzung
- Zum Teil nur indizierte, abgeleitete Angaben über Qualität
- BSP:
  - Anzahl von Branches -> Testbarkeit
  - Zugriff auf globale Variablen -> geringe Wartbarkeit des Systems
  - Anzalh lokaler Variablen -> aufwendiges Debuggen
  - Nicht-standart-Features -> Portabilität

73. McCabe's zyklomatische Komplexität

- Hypothese: Komplexität hängt von Komplexität des Kontrollflußgraphen ab (Program Graph)
- Experimente zeigen Beziehungen zwischen McCabe’s Metrik und Anzahl an Fehlern im Source Code (und der Zeit diese zu korrigieren)
- V(G) ist Indikator für maximale Modulgröße und sollte bestimmten Wert nicht überschreiten, sonst schwer zu testen
- Problem: Metrik sehr spät im Software Projekt gemessen
- Deshalb müssen Metriken gefunden werden, die schon im Entwurf messen können

3.7.4. Metriken für den Entwurf

74. Metriken für den Entwurf

- Grundidee ist, daß externe Komplexität eines Moduls in Verbindung mit Anzahl an Flüssen (in/export) zwischen Modul und Umwelt steht
- Interne Kompexität = LOC
- Lokale Flüsse
  - Direkt => Modul A gibt parameter zu B
  - Indirekt => Modul A gibt Wert zu B zurück
- Globale Flüsse
  - Modul A schreibt DS und Modul B liest vom DS
- Fan-In / Fan-Out (alles was reingeht und rauskommt egal ob lokal oder global)

- Complexity = length * $(f_i * f_o)^2$

75. Anwendung von Metriken

- Filter, welcher die komplexesten Module identifiziert
- Zum Untersuchen von Entwicklermethoden
- Untersuchen der steigenden Komplexität während der Wartung
- Die Teile / Module finden die die meisten Fehler / Arbeit machen
    - -> Redesign + Reimplementation

3.8. Legacy Systeme

76. Legacy-Systeme – Charakteristik und Probleme

- Wurden mit Anahme einer kurzen Lebenszeit entwickelt, sind aber immernoch im Einsatz
- Altsysteme sind Komplex und Austausch wäre sehr teuer
- Altsysteme verursachen hohe Wartungskosten
- Komplexe software, viel Support, wenig Dokumentation
- Bei Austausch gefahr, daß Geschäftswissen verloren geht
- Falsche Entscheidungen können schwere Folgen haben

77. Probleme mit der veralteten Softwarearchitektur

- Systeme bestehen aus vielen verschiedenen Einzelsystemem, welche Daten gemeinsam nutzen
- Systeme benutzen Teilweise Files und keine DBMS
- Daten sind of Redundant

3.9. Reengineering und Wartung

78. Reengineering – wann und warum?

- Software Evolution zwischen zwei Extremfällen der Systemersetzung und weiterlaufender Wartung
- Reengineering meist billiger als Neuentwicklung eines Systems
- Ausgangspunkt für RE ist exisitierenedes System
- Geringeres Risiko (inkrementell) und Kosten

79. Methoden des Reengineerings

- Source Code Translation
    - Programmübersetzung in andere Programmiersprache
    - Zielsprache in neue Version updaten (z.B. wenn alte Compiler nicht mehr kompatibel)
- Programm Neustrukturierung
    - Programme mit „Spagetti“-Struktur schwer zu untersuchen
    - Viele Automatische Neustrukturierungstools vorhanden
- System Neustrukturierung
    - Programm Neustrukturierung verbessert nicht die Systemarchitektur, da nur isolierte Betrachtung
    - System muss selbst Restrukturiert werden
- Inkremental Evolution bei Systemersetzung weniger Riskobehaftet als Bing Bang

80. Wartung von Software - Ursachen, Methoden

- Um Guten Zustand eines Systemes aufrechterhalten
- Einbringen von Änderungen nachdem das System verkauft wurde
- Drei Maintenance Arten
  - Korrigierende (Fehler beheben)
  - Adaptive (neues Umfeld in HW/SW)
  - Perfektive (neue Anforderungen)
- Vorausschauende Wartung (vorher einfach und gut strukturieren, dass Wartung billig wird)
- Kosten für Änderungen wärend Wartung extrem hoch
- Data Reengineering (Datenübernahme, Datenabgleich, Datenredefinition)
- Datenzentralisierung (RDBMS…)

3.10. Programmierungskonzepte

81. Generische Progammierung und Templates

- Generische Units sind parametrisierte Templates
- Sie müssen initialisiert werden, um dem Compiler den verwendeten Datentypen mitzuteilen
- DT kann auch abstrakt sein
- BSP: Sortierungsalgorithmus, Datentyp ist formeller Parameter

82. STL Library - die Idee

- Gemeinsame Komponenten zusammenfassen in Bibliothek
- STL mehrdimensionaler Raum (Alg, Containers / DS , DT) aus dem benötigte Prozedure mit benötigtem Datentyp und Datenstrukturen (Array, Listen…) gewählt werden kann
- STL verringert notwendige LOC, da ohne Templates i * j * k verschiedene Codeversionen
- Mit Template-Funktionen nur noch j * k zu programmierende Codesstücke
- Mit Template-Klassen für Container sogar nur j + k

83. Patterns

- Viel Allgemeiner als Template
- Kein Softwarestück, sindern ein Konzept von bewährten Lösungen
- Jedes Pattern besteht aus
  - Kontext
  - Problem
  - Lösung
- Arten von Pattern
  - Konzeptuell
  - Architektonisch
  - Entwurfsmuster
  - Programmierungsmuster
  - Dokumentationsmuster

84. Frameworks

- Idee ist Erweiterung der virtuellen Maschine

- Framework ist wiederverwendbare mini-Architektur welche generische Struktur und Verhalten von Softwarefamilien abstrahiert
    - Kontext spezifiziert Struktur
    - In Klassen geteilt
    - Wie Klassen und Objekte zusammenarbeiten
- Framework (Executable) ist eine Implementation von Design-Pattern
- Framework und Wiederverwendbarkeit
    - Dynamisches Binden, z.B. dynamische Konfiguration von Features während Laufzeit
    - Wiederverwendbar steigt, aber auch Nachteile:
        - Schwer zu durchschauen und zu verstehen
        - Erhöhte Komplexität
        - Alle Features von Vater-Klassen werden geerbt, auch wenn nicht benötigt -> somit steigende Größe
        - Performancenachlass durch dynamisches Binden

85. Metaprogrammierung (generative Programmierung)

- Applikationen bei denen Kontext zur Kompilierungszeit bekannt, somit statisches Binden anstatt von dynamischen möglich
- Meta-Programming benutzt Templatespezialisierungen (if-tool) und Templaterekursion (loop-tool) zum schreiben von C++ programmen, welche zur Komilierungszeit vom C++ Compiler interpretiert werden
- Compile Time Computation und Code Generation

86. Adaptive Programmierung

- OO Technologie kapselt Daten und Funktionen in Klassen
- Implementation ist geschützt gegen Änderungen an DS, da nur über Schnittstellen Zugriff
- Mancha Applikationen leiden unter periodischen Änderungen in Klassenstrukturen und Klassenhierachien
- Apdaptive Programmierung ermöglicht den Applikationen eine Schnittstelle zur Klassenhierarchie

87. Aspekt-orientierte Programmierung

- Funktionalität ist bei OO in Klassen gekapselt
- Problem:
    - Es gibt Dienste die nicht nur in einer Klasse gekapselt werden können (z.B. Tracing)
    - So ein Service ist auf viele Objekte verteilt
    - Wie kann man einen solchen Dienst trotzdem zentralisiert Implementieren?
    - Aspekte sind Funktionalitäten, Modelle solcher Dienste
    - Asprekte werden in eigenen separaten Units gekapselt
    - Während des Preprocessings werden die Aspekte durch einen WEAVER im Programm verteilt

4. Anhang

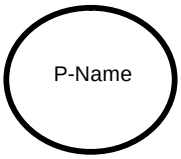
4.1. Datenflussdiagramme

- Mit Datenflussdiagrammen kann die Funktionalität eines Systemes modellierbar
- Funktionalität = Transformationen von Daten durch Prozesse
- beschreibt auf grafische Weise, woher und wohin die Daten fließen
- ist Modell der Systemfunktionalität als gerichteter Graph
- Steuerung des Ablaufs aber nicht ersichtlich

4.1.1. Bestandteile

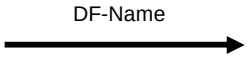
- **Prozesse** beschreiben die Funktionalität des Systems. Die durch Flüsse zu einem Prozeß transportierten Daten werden verarbeitet und in transformierter Form an abgehende Flüsse übergeben. Ein Prozeß, der andere Prozesse kontrolliert, heißt *Kontrollprozeß*. Er kommuniziert (über Flüsse) ausschließlich mit anderen Prozessen oder Begrenzern.

transformiert Ein- in Ausgabedaten



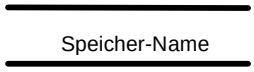
- **Datenflüsse** werden benutzt, um den Transport von Informationen von einem Teil des Systems in einen anderen zu beschreiben. Flüsse sind wie Lager getypt, d.h. sie erlauben nur den Transport eines bestimmten Informationstyps. Quelle bzw. Ziel eines Flusses kann ein Prozeß, ein Lager oder ein Begrenzer sein

Informationskanal



- **Speicher bzw. Lager** stellen ruhende Daten bzw. Informationen im System dar. Ein Lager kann nur Daten eines bestimmten Typs aufnehmen und ist über Flüsse mit Prozessen oder Begrenzern verbunden. Die Prozesse steuern dabei den Zugriff auf die abgelegten Informationen.

Aufbewahrung von Daten



- **Terminator bzw. Begrenzer** modellieren die Schnittstellen des Systems zu seiner Außenwelt. Sie unterliegen nicht der Kontrolle des Softwaresystems und sind nicht Gegenstand der Anforderungsermittlung, werden aber aus Verständnisgründen in das Modell aufgenommen. Typische Beispiele für Begrenzer sind natürliche Personen oder externe Computersysteme, mit denen kommuniziert wird.

externer Kommunikationspartner

4.1.2. Syntaktische Regeln von DFDs

- zwischen Terminatoren gibt es keine direkten Datenflüsse
- zwischen Speichern gibt es keine direkten Datenflüsse
- zwischen Terminatoren und Speichern gibt es keine direkten Datenflüsse

- alle Elemente eines DFD müssen eindeutig benannt werden
  - einzige Ausnahme sind Datenflüsse, welche ganze Datensätze zwischen Prozess und Speicher übertragen
- keine Speicher, die entweder nur beschrieben oder nur gelesen werden
- keine Prozesse, die Daten ausgeben, die sie nicht erhalten haben, oder die Daten erhalten, ohne sie auszugeben oder zu verarbeiten
- keine Darstellung von Kontrollflüssen

4.1.3. Kontextdiagramme

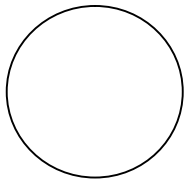
- Sind DFD's der obersten Ebene und enthalten keine Speicher
- Stellen Beziehungen des Systems mit Umwelt dar
- Daher nur Terminatoren und Prozesse verbunden durch Datenflüsse und keine Speicher

4.2. Petri-Netze

- Modellierung eines Systems konkurrierender/kooperierender Prozesse geeignetes Werkzeug

4.2.1. Bestandteile / Darstellungselemente

- *Stelle*: Zustand eines Prozesses; repräsentiert eine Bedingung



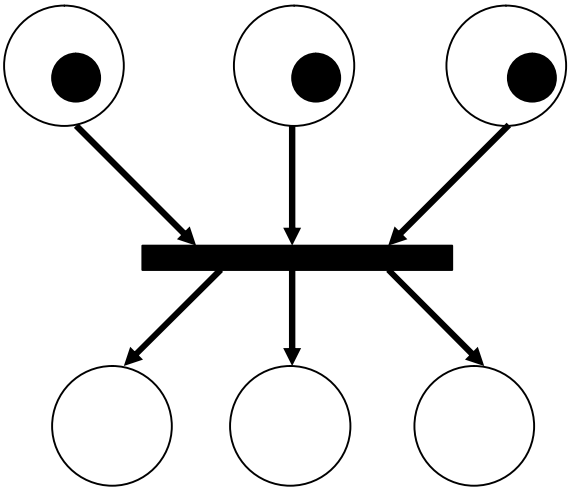
- *Transition*: gesteuerter Zustandsübergang; modelliert ein Ereignis, welches durch das Schalten („Feuern") der Transition ausgelöst wird

- *Pfeil*: zwischen Stelle und Transition und umgekehrt

- *Marke*: ihre Anwesenheit in allen vor einer Transition liegenden Stellen ist die notwendige Bedingung, dass die Transition schalten (feuern) kann

4.2.2. Schaltregeln

- Eine Transition t kann schalten, wenn jede Eingabestelle von t eine Marke enthält und wenn jede Ausgabestelle von t leer ist.
- Schaltet eine Transition t, dann wird aus jeder Eingabestelle von t eine Marke entfernt und zu jeder Ausgabestelle von t eine Marke hinzugefügt.



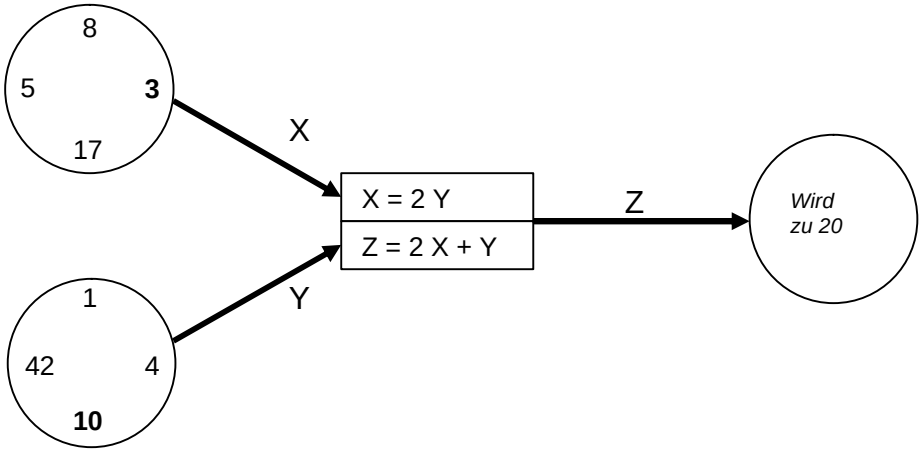
- Deadlocks werden durch Mutual Exclusion vermieden

4.2.3. Unzulänglichkeiten von Petri-Netzen

- Marken repräsentieren den zwar Kontrollfluss, sind jedoch anonym → Inhalt der Marke ist unbekannt
- wenn mehrere Transitionen geschalten werden können, keine Angabe von Prioritäten für die Marken möglich
- keine Beschreibung von Zeitbeschränkungen möglich (wenn z.B. Transition warten muss)

4.2.4. Erweiterte Petri-Netze

- Marken werden durch Werte repräsentiert
- Transitionen sind Prädikate zugeordnet, die sich auf die Werte der Marken der Eingabestellen beziehen und den Übergang mitbestimmen (Transitionen können nur noch beim Vorliegen bestimmter Marken schalten)
- Transitionen sind Funktionen zugeordnet, die aus den Werten der Marken der Eingabestellen die Werte der Marken der Ausgabestellen berechnen
- Angabe von Zeitbeschränkungen:
  - jeder Transition ein Tupel $<t_{min}, t_{max}>$ zuordnen, das die minimale und maximale Wartezeit bestimmt



5. Quellen- und Literaturverzeichnis

Skript Prof. Kroha
http://www.tu-chemnitz.de/informatik/HomePages/ISST/

Srkipt L. Rosenhainer
http://www.tu-chemnitz.de/informatik/HomePages/ISST/

Online Skript
http://www.minet.uni-jena.de/www/fakultaet/ips/braunpet/swt/

Online Skript
http://www.fortytwo.uni-oldenburg.de/~gahl/gremlins/soft/soft.html

Interessantes
http://www.requirements-analysis.info/Einfuhrung/Notationen/notationen.html

Introduction to Software Engineering – University Of California
http://www.ics.uci.edu/~taylor/ics52_fq01/syllabus.html

JSP
http://iis-web.coloradotech.edu/bsanden/CS649/1

http://www.soc.staffs.ac.uk/~cmtkch/teaching/cds/level_2/sep/supplemental/jsp/jackson%20structured%20programming.htm

CASE
http://www.gi-ev.de/informatik/lexikon/inf-lex-case.shtml

http://www.stud.uni-siegen.de/henning.schmick/referat/index.htm#Index

Holger Kreissl,
Chemnitz, den 4.02.2003