Gaida, Lässer, Schwendinger

Lerntagebuch

LV: Softwareprozesse und Softwarequalität

University of Applied Sciences: Vorarlberg

Department of Computer Science
Dipl. Inform. Bernd G. Wenzel

Dornbirn, 2014

Abstract

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.

Inhaltsverzeichnis

Listings

Abbildungsverzeichnis

Todo list

1 LV 1 am 21.02.2014

1.1 Organisatorisches

Zu Beginn dieser Vorlesung wurden zuerst die organisatorischen Randbedingungen für diese Lehrveranstaltung geklärt.
Dabei wurde entschieden, dass anstelle einer Prüfung ein Lerntagebuch in Gruppen zu max. drei Personen erstellt wird. Das Lerntagebuch enthält zusätzlich ein Glossar mit den wichtigsten Begriffsdefinitionen rundum das Thema Softwareprozesse und Softwarequalität.

1.2 Motivation des Software-Engineering

Als Einstieg in das Thema dienten ein paar nicht ganz ernst gemeinte "Gesetze". Diese Gesetze sollten uns verdeutlichen, dass Software-Engineering nicht so trivial und einfach ist, wie es oft den Anschein hat. Wie schon die Folgerung von Kleinbrunner besagt

> "Wenn eine Programmieraufgabe leicht aussieht, ist sie schwer."

Es wurden unter anderem die "Gesetze"von Gutterson, Farvour, Brunk, Munbright und vielen anderen vorgestellt.

Um die Bedeutung von qualitativ hochwertiger Software zu verdeutlichen, stellte der Vortragende einige Probleme in bekannten Softwareprojekten wie beispielsweise der Verlust einer Segelyacht mit Crew vor der mexikanischen Pazifikküste aufgrund eines Softwarequalitätsfehlers.

Abschließend erhielten wir die Aufgabe bis zur nächsten Vorlesung ein Programm zu entwickeln zur Berechnung der reellen Lösungen einer quadratischen Gleichung. Ziel bei dieser Aufgabe ist es auf die Qualität des Codes zu achten.

2 LV 2 am 07.03.2014

Zu Beginn wurden kurz die Lösungen der Hausaufgabe besprochen. Es wurde verdeutlicht, dass sogar so eine einfache Aufgabenstellung von einigen Studenten anders implementiert wurde, als ursprünglich gedacht.

Aufbauend auf LV 1 wurden wichtige Begriffe wie Software, Software-Engineering, Softwareprozess usw. definiert. Die wichtigsten Definitionen, auch von zukünftigen Vorträgen, sind im Glossar angeführt. Dabei wurde auch das Vorgehensmodell und die Methode des Software-Engineerings vorgestellt.

Bei der Methode des Software-Engineerings geht es um einen strukturierten Ansatz für die Software-Entwicklung mit Ziel eine möglichst hohe Qualität der Software bei gleichzitig geringen Entwicklungskosten zu erreichen.

Das Vorgehensmodell definiert wie der Softwareentwicklungsprozess abläuft. Bekannte Modelle sind dabei Scrum, Wasserfall oder das V-Modell. Dabei stellt sich heute oft die Frage, ob man sich für einen klassisches oder agiles Vorgehensmodell entscheidet. Der größte Unterschied zwischen agil und klassisch ist, dass beim Agilen Ansatz der Kunden viel mehr involviert ist und öfters das Feedback von ihm in die Entwicklung mit einfließt. Auch sind die Iterationen beim Agilen Vorgehen meist kürzer.

Anschließend behandelten wir das Thema der soziotechnischen Systeme. Dabei wurde der Begriff genauestens definiert, die wesentlichen Eigenschaften und der Lebenszyklus sowie die Umfeldfaktoren vorgestellt.

Nach den soziotechnischen Systemen behandelten wir das Thema der kritischen Systeme. Dabei geht es um Systeme dessen Ausfall oder Fehlfunktion sich auf die Umgebung in Form von z.B. wirtschaftlichen Verlusten, Schäden an der Umwelt auswirken. Wichtige Faktoren dabei sind die Verfügbarkeit, Zuverlässigkeit, Betriebssicherheit und Systemsicherheit. Der Sammelbegriff für diese Punkte heißt Verlässlichkeit.

Zum Schluss der Lehrveranstaltung wurden die ethnischen Herausforderungen beim Software-Engineering behandelt. Dabei geht es um Vertraulichkeit gegenüber dem Arbeitgebern und Kunden, um den Schutz von geistigem Eigentum und den Schutz vor Computermissbrauch. Es gibt auch einen Knigge für das professionelle Verhalten des Software-

Engineering definiert durch ACM/IEEE.

2.1 Anforderungsanalyse

Bei der Anforderungsanalyse ist es wichtig die Projektbeteiligten eindeutig zu identifizieren. Hierbei ist es auch wichtig deren Erwartungen festzustellen. Sprachbarrieren zu beseitigen sowie die Unterscheide im Bezug auf Anforderungen zu aufzuklären. Anbei sehen sie einige Beispiele solcher Beteiligten.

- Endanwender
- Kooperationspartner
- Systemadministratoren
- Sicherheitsbeauftragte
- Marketingabteilung andere betroffene Mitarbeiter
- Hard- und Softwareentwickler Manager
- Technologie-Experten
- Anwendungsexperten
- Betriebsratsmitglieder
- Behörden

3 LV 3 am 21.03.2014

3.1 Anforderungsanalyse

3.1.1 Sammeln von Anforderungen

Um Anforderungen zu sammeln gibt es verschiedene Möglichkeiten:

- Fragebogen
- geschlossene Interviews
- offene Interviews

Sehr gut geeignet ist eine Mischung aus allen drei Varianten. Wichtig ist, dass man die Leute interagieren lässt - sprich man benötigt ein gewisses Maß an Flexibilität und muss sehr gut zuhören können.

Bei der Sammlung von Anforderungen treten einige Probleme auf. Eine sehr große Hürde stellt der Jargon und das implizite Wissen dar. Die Leute gehen davon aus, dass jeder ihren Jargon (mit allen Begrifflichkeiten) kennt und versteht. Das implizite Wissen ist den Leuten meistens nicht bewusst (z.B. aus Gewohnheit) und wird von ihnen zurückgehalten. Beim nicht expliziten Wissen, wissen die Leute, dass sie es haben und gehen davon aus, dass jeder dieses Wissen hat, da es für sie selbstverständlich ist.

Wichtig ist, dass man nicht sofort mit Interviews anfängt, sondern sich erst in die Domäne einliest. Andernfalls ist das Ergebnis von den Interviews nicht zu gebrauchen, da einfach das Verständnis für die Domäne nicht vorhanden ist.

Szenarien stellen eine weitere Technik zur Sammlung von Anforderungen dar. Es ist zu empfehlen bei der Erstellung von Szenarien Ad-hoc vorzugehen und zur Dokumentation UML-Diagramme zu verwenden. Wichtig dabei ist es auch auf die Details zu achten, um das Gesamtbild besser verstehen zu können.

Eine weitere Technik zur Sammlung von Anforderungen ist die Ethnografische Methoden. Durch diese Variante kann man durch Beobachten von Personen Anforderungen herausgefunden. Dabei wird diese Person neutral und ohne jegliche Interaktion mit dieser

beobachtet. Es müssen stets stillschweigend Notizen gemacht werden und am Ende wird mit einem Experten das Beobachtete diskutiert. Es ist von Vorteil dies mit der jeweiligen beobachteten Person zu machen und nicht mit deren Chef, da sonst weitere Beobachtungen durch wenig Kooperation der Mitarbeiter behindert werden könnten.
Durch diese Variante kann man sehr gut implizite, nicht explizite Anforderungen und Abweichungen erkennen. Auch die Anforderungen der Benutzer können dadurch sehr gut ermittelt werden.

3.2 Klassifizierung von Anforderungen

Bei der Klassifizierung hat man die Aufgabe die Struktur der Anforderungen aus den unterschiedlichen Quellen zu definieren. Es ist wichtig Duplikate sowie Synonyme zu erkennen, die Beziehungen zwischen den Anforderungen zu definieren und eine Gruppierung der Anforderungen durchzuführen.

3.3 Validierung von Anforderungen

Es gilt zu prüfen, ob die Anforderungen gültig (filtern von falschen und unnötigen Anforderungen), konsistent, vollständig (sehr schwer zu prüfen), realisierbar (genügt das Budget, Zeit? ist die Technologie geeignet?) und verifizierbar sind.

Um dies Überprüfung durchzuführen kann man ein Anforderungsreview machen, einen Prototypen entwickeln oder Testfälle erzeugen.

Für ein Anforderungsreview benötigen wir aus allen wichtigen Gruppen der Stakeholder einen Vertreter, einen Systemarchitekten und einen Vertreter der Softwareentwickler (diese müssen am Ende ja wissen, was sie entwickeln sollen). Die Durchführung erfolgt durch den jeweiligen Anwendervertretern. Es werden dabei alle Anforderungen diskutiert. Durch das Anforderungsreview können Konflikte, Widersprüche, Fehler und Versäumnisse ermittelt werden. Sie sollten in einem Review-Bericht festgehalten werden. Die Prüfung auf Konsistenz und Vollständigkeit fallen unter die Kategorie der Notwendigen Prüfungen. Prüfungen bezüglich der Verifizierbarkeit, Verständlichkeit sowie Anpassungsfähigkeit sind Optionale Prüfungen.

3.4 Priorisierung von Anforderungen

Die Priorisierung ist speziell bei einer agilen Vorgehensweise sehr wichtig.

Der Grundgedanke ist durch Phasenweise Implementierung das Leben der Anwender durch kontinuierliche Softwareupdates zu erleichtern. Es minimiert auch das Risiko der Einführung der Software.

Die Prioritätssteuerung erfolgt über eine Bewertungsformel. Es wird jeweils eine Bewertung zu jeder einzelnen Anforderung durch das Nutzwertanalyseteam und das Kostenanalyseteam erstellt. Beide Teams bewerten völlig unabhängig voneinander und ohne Kommunikation zwischen den Teams. Nach der Auswertung erhält man eine Liste der Anforderungen welche für die nächste Ausbaustufe abgearbeitet werden sollten.

3.5 Dokumentation und Verwaltung

3.5.1 Modelle

Bieten eine detaillierte und formalisierte Dokumentation über den Ist-Zustands und Soll-Zustands. Es werden auch Sichten von Kunden oder anderen Abteilungen auf beide Zustände festgehalten.

Kontextmodelle

Mit diesem Modell werden die Systemgrenzen hinsichtlich des Gesamtsystems definiert. Hier ist es wichtig den Kontext zu definieren. Neben dem Gesamtsystem werden die technischen Systeme betrachtet. Bei den technischen Systemen ist die Definition des Scopes besonders wichtig. Wichtig bei der Erstellung eines Kontextmodelles die die Auswahl der Verfügbaren Notationen (UseCase-Diagramm, Architekturdiagramm, etc.).

Verhaltensmodelle

Es werden Daten- und Ereignis-orientiert die Abläufe im System definiert. Es gibt auch Mischformen (z.B. eine Banküberweisung). Zur Veranschaulichung dienen Diagramme wie Datenflussdiagramme, Prozessdiagramme, Zustandsdiagramme, Sequenzdiagramme oder Szenarios.

Datenmodelle

Für die Darstellung der Datenmodelle gibt es ER-Diagramme, UML-Klassendiagramme, XML-Schema- OWL und diverse andere Diagrammtypen. Sie dienen zur logischen Definition der Datenstrukturen hinsichtlich transients und persistents. Auch hier ist die Auswahl der verfügbaren Notation von großer Wichtigkeit.

Ein wichtiges Werkzeug bei den Datenmodellen ist das Data Dictionary. Hier speichert man sich die Klassen und Attribute. Man speichert sich zu den Entitäten gewisse Eigenschaften wie zum Beispiel den Namen, Beschreibung und Typ.

Objektorientierte Modellierung

Bei der Objektorientierten Modellierung geht es um die Vereinigung der Funktionalität von Daten und Verhaltensmodellierung. Hier ist die Rede von Daten, Datenflüssen sowie Ereignissen. Als Standard Notation wird hierbei UML verwendet. Bei der Modellierung werden folgende Highlights verwendet.

- Klassifikationshierarchien
- Aggregationsbeziehungen
- Operationen (Objektverhalten)

Strukturierte Methoden

Die strukturierte Methode beschreibt eine detailliert definierte Vorgehensweise bei der Software-Entwicklung. Normalerweise basierend auf einem Satz von Diagrammtypen in Verbindung mit Regeln und Richtlinien. Beispiele für Strukturierte Methoden sind:

- JSP
- V-Modell
- RUP

Durch den Einsatz von strukturierten Methoden wird gewährleistet, dass immer eine gewisse Qualität erreicht wird.

Die Nachteile sind mangelnde Unterstützung nicht-funktionaler Anforderungen und die Anwendbarkeit für konkrete Probleme. Letzteres benötigt sehr viel Erfahrung um die Entscheidung *Wann verwende ich welche Methode* korrekt treffen zu können. Es muss auch beachtet werden, dass die Anpassung einer Problemklasse sehr schwierig ist und dass ein enormer Dokumentationsaufwand oft das Wesentliche verdeckt. Eine weitere Gefahr bei

dieser Methoden ist der hohe Detaillierungsgrad, welcher das Verständnis für das System erschwert.

3.5.2 Anforderungsmanagementsystem

Erfahrungsgemäß sind die Anforderungen am Anfang eines Projektes stets unvollständig und sie ändern sich im laufe eines Projektes fortlaufend. Die Anforderungen helfen allerdings dabei das Problem bzw. die Probleme besser zu verstehen. Deshalb ist der Umgang mit Anforderungen und ihren Änderungen von enormer Bedeutung. Hier ist es wichtig auf eine gute Dokumentation zu achten und die richtigen Werkzeuge zur Unterstützung einzusetzen. Vor allem die Aufgabenübersicht (Anforderungssammlung, Beziehungsmanagement, etc) ist nicht zu vernachlässigen.

Man muss bei den Anforderungen zwischen verschiedenen Typen unterscheiden. Es gibt dauerhafte Anforderungen, die relativ stabil sind und mit dem Kernbereich des Systems verbunden. Sie können aus standardisierten Modellen des Anwendungsbereichs abgeleitet werden. Daneben gibt es veränderliche Anforderungen mit einer sehr hohen Änderungswahrscheinlichkeit. Es müssen wirtschaftliche, technische und gesetzliche Randbedingungen beachtet werden.

4 LV 4 am 25.04.2014

4.1 Dokumentation und Verwaltung

4.1.1 Pflichtenheft

Das Pflichtenheft dient zur Zusammenstellung der vollständigen und detaillierten Benutzer- und Systemanforderungen. Wichtig ist, dass das Pflichtenheft vollständig ist, wobei dieses Ziel nur sehr schwer zu erreichen ist. Bei Projekte der Europäischen Union ist es sogar nicht mehr erlaubt Änderungen am Pflichtenheft nach Vertragsabschluss durchzuführen. Dies ist durch ein Wettbewerbsgesetz geregelt.
Ein Pflichtenheft kann aus mehr als einem Dokument bestehen und wird für Kunden, Manager, Entwickler, Tester, Warter und die Juristen geschrieben.
Ein beliebter Trick in diversen Ländern ist es, das Pflichtenheft via E-Mail abzuändern. Wenn diese Änderung allerdings nicht innerhalb von 14 Tagen widerrufen wird, dann ist sie Bestandteil vom Vertrag und muss entwickelt werden. Das gleiche gilt für Besprechungen, die schriftlich Protokolliert werden. Sollten diese Änderungen dann nicht berücksichtigt werden, kann ein Rechtsstreit eintreten.
Im Pflichtenheft erfolgt eine Klassifizierungen der Informationen hinsichtlich der Anforderungen (mandatory requirements), Wünsche (optional requirements), informative Bestandteile (dienen zum besseren Verständnis, sind unverbindliche Informationen) und Warnungen (Abbildungen, Tabellen, Anhänge als informative Bestandteile). Um informative Bestandteile wie ein Anhang zu einem vertraglicher Bestandteil zu machen, muss es mit Normative gekennzeichnet werden.
Der Aufbau vom Pflichtenheft kann auch nach vordefinierten Normen und Standards erfolgen. Ein Beispiel für eine solche Norm ist die IEEE/ANSI 830-1998 Norm. Wichtig ist das auch die Normen und Standards von Referenzen zum Bestandteil des Vertrages werden.

4.2 Spezifikation kritischer Systeme

Kritische Systeme sind jene, bei deren Benutzung durch das Auftreten eines Fehlers Gefahr besteht - zum Beispiel ein GPS-System oder ein Defibrillator in der Medizintechnik.

4.2.1 Anforderungen an kritische Systeme

Funktionalen Anforderungen unterstützen Definitionen von Funktionen zur Fehlerprüfung, Funktionen zur Wiederherstellung im Fehlerfall und Aspekte zum Schutz gegen Systemausfälle. Zu den nicht funktionale Anforderungen zählen die Systemzuverlässigkeit und die Systemverfügbarkeit. Es werden auch Negativanforderungen festgelegt, welche jene Verhalten beschreiben, das ein System auf keinen Fall zeigen darf beschreiben. Eine Negativanforderung wäre zum Beispiel das ein Flugzeug nicht schneller als 800km/h fliegen darf.

4.2.2 Risikomanagement

Beim Risikomanagement werden Gefahren bestimmt, klassifiziert und versucht deren Risiko zu verringern.
Prinzipiell gelten Murphy's Gesetze: Älles, was schiefgehen kann, wird irgendwann einmal schiefgehen.Alles, was schiefgehen kann, wird genau dann schiefgehen, wenn es den maximalen Schaden verursacht.öder Wenzels Korollar dazu "Murphy war ein Optimist."
Das Ziel beim Risikomanagement ist es, Risiken zu erkennen und dessen Auswirkungen zu minimieren. Risiken entstehen prinzipiell dort, wo Systemkomponenten aufeinandertreffen oder die Umwelt Einflüsse auf das System hat. Entgegenwirken kann man durch erfahrene Entwickler, Berater oder Fachleute des Anwendungsgebiets (siehe witziges Beispiel Ölschraube bei Scania).
Bei der Risikoanalyse und der Risikoklassifizierung spielen die Begriffe Unfallwahrscheinlichkeit, Schadenswahrscheinlichkeit und Schadenshöhe eine wichtige Rolle. Durch die Klassifizierung kann man abschätzen, ob ein Risiko vernachlässigbar ist oder ob es unannehmbar ist. Unannehmbare Risiken verursachen große Schaden, haben eine hohe Schadenswahrscheinlichkeit und sollten unter allen Umständen versucht werden zu vermeiden.
Um herauszufinden woher ein Risiko kommt wird es zerlegt um die Ursache für das Risiko zu entdecken. Hierfür gibt es viele verschiedene Techniken wie Reviews, Checklisten, Petri-Netze, Fehlerbaum sowie einige andere Techniken.
Um die Risiken minimieren zu können, werden in der Praxis die Techniken zur Risikovermeidung-, Risiko-Erkennung- und die Risikobeseitigungen- sowie die Schadensbegrenzungsstrategie kombiniert.

4.2.3 Betriebssicherheit

Wichtig ist es das System in Hinsicht auf dessen Betrieb zu sichern. Vor allem Schutzmechanismen und Ausfallsysteme sind von großer Bedeutung. Auch die Betriebssicherheit

im Laufe des Produktlebenszyklus ist zu beachten. Um Fehler in diesem Bereich klassifizieren zu können, gibt es verschiedene Klassen. Klasse 1 bis Klasse 5 werden dabei in der Software Industrie als Standard angesehen.

- Klasse 1: Fehler die andere Anwendungen betreffen können.
- Klasse 2: Fehler die Benutzer der selben Anwendung betreffen können.
- Klasse 3: Funktion steht ohne Workaround nicht zur Verfügung.
- Klasse 4: Funktion steht nur mittels Workaround zur Verfügung.
- Klasse 5: Verbesserungsvorschlag

4.2.4 Systemsicherheit

Bei der Systemsicherheit spielen vor allem die Bedrohungen eine wichtige Rolle. Die Bedrohungen können dabei in Kategorien unterteilt werden. (Direkte Bedrohungen, Indirekte Bedrohungen, etc.). Vor allem die Monopolartige Stellung einiger Unternehmen führt dazu das Sicherheitslücken im großen Stil auftreten können. Zudem gibt es keine 100

4.2.5 Zuverlässigkeit

Damit ein System zuverlässig ist, müssen sowohl die Hardware als auch die Software und die Benutzer zuverlässig sein. Die Zuverlässigkeit einer Software ist messbar. Dabei werden Messwerte bezüglich Systemfehler, Zeit zwischen Systemfehlern und ähnlichem verwendet. Vor allem die Zeit zwischen Systemfehler und Wiederverfügbarkeit des Systems ist wichtig für die Aussage über die Zuverlässigkeit eines Systems.
Die Vorgehensweise bei der Zuverlässigkeitsspezifikation ist es die möglichen Systemfehler zu ermitteln, deren Auswirkungen zu analysieren und mögliche Vorkehrungen zur Beseitigung dieser Risiken zu realisieren. Besonders wichtig ist es allerdings zu berücksichtigen, dass extreme Zuverlässigkeit nicht testbar ist.

4.2.6 Formale Spezifikationsmethoden

In die formale Spezifikationsmethoden wird schon seit langer Zeit gearbeitet. Der Durchbruch dieser Technik blieb allerdings bis heute aus. Die Gründe dafür sind neben der Entstehung von anderen Software-Engineering-Methoden auch die Marktveränderungen in diesem Bereich und die beschränkte Einsetzbarkeit der Methode. Allerdings ist diese Methode vor allem in den Bereichen des Luftverkehrs, der Raumfahrt und in der Medizintechnik weit verbreitet. Die Nutzung in Softwareprozessen kann entweder mittels

einem Linearen Prozess oder einem Parallelen Prozess erfolgen. Die Spezifikation von System-Schnittstellen kann mit unter allerdings recht Umfangreich werden. Auf Grund dieses Problems entsteht hier eine Chance für den Model Driven Architekuture Ansatz. Formale Spezifikationstechniken können als gute Ergänzungen zu informellen Techniken betrachtet werden. Vor allem im Kostenvergleich können sie überzeugen.

5 LV 5 am 23.05.2014

5.1 Besprechung Hausaufgabe - Anforderungen

5.1.1 Funktionale Anforderungen

- Ergebnis x element R : $ax^2 + bx + c = 0$
- kein Absturz bei komplexer Lösung
- vernünftige Reaktion auf Eingabefehler
  - a = 0
  - unvollständige Eingaben
  - $4ac > b^2$
- relativer max. Rundungsfehler von 0,1% mit Beweis (für die wirkliche Realisierung darf die Formel nicht 1 zu 1 programmiert werden. In der Realität wird eher eine Newton-Annäherung implementiert).
- es gibt immer ein definiertes Ergebnis (Ausnahmebehandlungen)
- keine Endlosschleife (wenn Newton angewendet wird, dann braucht es eine vernünftige Abbruchbedingung)
- Zeitbedarf max 1 msec
  - CPU der Ver
  - Rechner/Prozessor
  - max. 2x SQRT der Standardimplementierung
- Signatur: Name, Parameter, Ergebnistyp

Jetzt soll jeder seine Lösung anhand dieser Anforderungen bewerten. Die Lösung darf auch dahingehend verbessert werden, dass alle Anforderungen erfüllt werden.

5.2 Referenz-Architekturen

Refernez-Architekturen beschreiben eine Definition typischer Strukturen. Diese sind nicht unbedingt für die Implementierung geeignet.

- ISO-OSI-Referenzmodell
- ECMA-Toaster (für CASE-Umgebungen)
- Schichtenarchitektur

5.3 Verteilte Systeme

Die Motivation hinter Verteilten Systemen ist vor allem eine bessere Ressourcenteilung. Hier sind Faktoren wie Offenheit, Nebenläufigkeit sowie Skalierbarkeit von enormer Bedeutung. Der Nachteil an solchen Systemen ist deren Komplexität.

Wir haben folgende Varianten für verteilte Systeme kennengelernt

- Mehrprozessor-Architekturen
- Client/Server-Architekturen
- Verteilte Objektarchitekturen
- Peer-to-Peer-Architekturen
- Dienstorientierte Architekturen

5.4 Echtzeit-Software

Echtzeitsysteme beschreiben ein Softwaresystem bei welchem die einwandfreie Funktionsweise auch von der Zeit zwischen den Auftreten eines Stimulus und der Durchführung der entsprechenden Antwort abhängig ist. Hierbei wird zwischen einem weichen und einem harten Echtzeitsystem unterschieden.

5.5 Bedienoberflächen

Bei der Bedieneroberfläche sind drei wichtig Punkte zu beachten. Punkt eins sind die Anforderungen, eine Oberfläche sollte so aufgebaut sein das sie den Anforderungen entspricht. Punkt zwei ist die Gestaltung. Eine Oberfläche sollte nach den wünschen der Anwender gestaltet werden. Punkt drei ist der Entwurfsprozess. Dieser sollte in Zusammenarbeit mit dem Kunden erarbeitet und durchgeführt werden.

5.6 Objekt-orientierter Entwurf

Der Objekt orientierte Entwurf ist untergliedert in drei Phasen.

- Objekt-orientierte Analyse
- Objekt-orientierter Enwurf
- Objekt-orientierte Programmierung

5.7 Entwicklung - Entwicklung kritischer Systeme

Bei der Entwicklung kritischer Systeme, gibt es verschiedene (komplementäre) Strategien. Hierbei spielt vor allem die Erkennung von Fehlern eine große Bedeutung

- Fehlervermeidung
- Fehlerentdeckung
- Fehlertoleranz

Um Fehler zu erkennen, stehen folgende Techniken zur Verfügung:

- Verlässliche Softwareprozesse
- Qualitätsmanagement
- Formale Spezifikation
- Statische Verifikation
- Starke Typisierung
- Sichere Programmierung
- Ausnahmebehandlung
- Geschützte Information

5.7.1 Fehlervermeidung

Die Fehlervermeidung beginnt bereits bei den Anforderungen. Diese müssen genau inspiziert werden. Auch ein Anforderungsmanagement ist notwendig, da sich Anforderungen ändern. Ebenfalls wichtig ist, dass mindestens ein Vier-Augen-Prinzip beim der Codeimplementierung und dem Entwurf verfolgt wird. Des Weiteren gilt es Tests zu planen und zu managen. Auch welche Konfigurationen erstellt und ausgeliefert werden, muss klar sein.

Sichere Programmierung

Im Rahmen der sicheren Programmierung, sind mögliche Gefahrenquellen wie zum Beispiel Gotos, Rundungsfehler von Gleitkommazahlen, Pointer, dynamische Speicherallokierung, Parallelität, Rekursionen, Interrupts, Vererbung, Aliasing, sowie keine Überprüfung von Arraygrenzen und Konfigurationsdaten.

Ausnahmebehandlung

Es sollten prinzipiell nur Sprachen verwendet werden, die eine vernünftige Ausnahmebehandlung bieten. Der Code wird dadurch wartbarer und Fehler lassen sich vermeiden.

5.7.2 Fehlertoleranz

Es gibt zwei Typen der Fehlererkennung

- vorbeugende Fehlererkennung
- rückblickende Fehlererkennung

Ein wichtiges Hilfsmittel dafür stellt die Validierung da. Auch wichtig ist das Fehler reproduzierbar sind, sowie das ein System im Fehlerfall wiederhergestellt werden kann.

Fehlertolerante Architekturen

Eine Variante ist die Verwendung von redundanter Hardware. Dabei ist es wichtig, dass auf jeder Hardware Software von unterschiedlichen Entwicklern laufen, um die Ergebnisse vergleichen zu können um so fehlertoleranter zu werden.

Eine andere Variante ist die Verwendung von sogenannten Wiederherstellungsblöcken.

5.8 Weiterentwicklung

5.8.1 Ursachen der Wartung

Wartung bedeutet nicht gleich Fehlerbehebung. Sie macht nur 17% bei der Wartung aus. Circa 65% machen Erweiterungen aus und weitere 18% dies Softwareanpassung an z.B. neue Hardware.

5.8.2 Wartungsaufwand

Oftmals wird in der Praxis das Entwicklerteam nach dem Release auf ein neues Projekt angesetzt und ein anderes Team liest sich in die Software ein und wartet sie dann bzw. die Wartung wird an eine andere Firma übertragen. Das ist unter anderem ein Grund für hohen Wartungsaufwand. Oft spielen auch die Fähigkeiten (z.B. mangelnde Erfahrungen im Anwendungsgebiet, der eingesetzten Technologie) der Entwickler, veraltete/fehlerhafte/nicht vorhandene Dokumentationen oder fehlendes Konfigurationsmanagement eine entscheidende Rolle.

Ebenfalls wichtige Einflussgrößen sind die Anzahl und Komplexität der Schnittstellen, die Anzahl der veränderlichen Anforderungen und die Geschäftsprozesse, in denen das system verwendet wird.

5.8.3 Weiterentwicklungsprozesse

Die hier angeführt Grafik verdeutlicht einen Prozess zur Software Weiterentwicklung. Es zeigt den Prozess von Sommerville

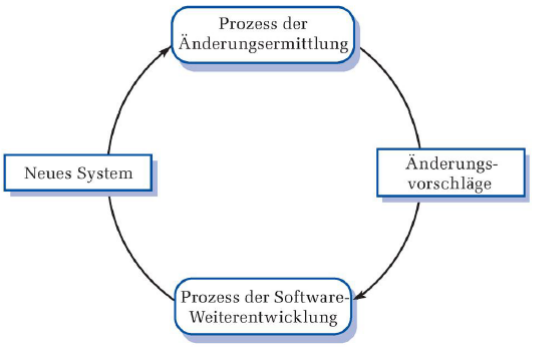

Abbildung 5.1: Grundprozesse nach Sommerville

Bei der Weiterentwicklung ist vor allem das Ermitteln von Änderungen ein wichtiges Thema

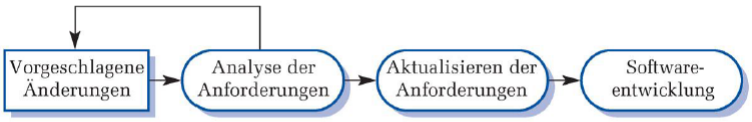

Abbildung 5.2: Änderungsermittlungsprozess nach Sommerville

Dabei ist zu beobachten, dass es manchmal von nöten ist schnell Änderungen durchzuführen. Hierbei helfen natürlich gute Prozesse für die Weiterentwicklung.

Seite 35-39 (excl. Reengineering)

6 LV 6 am 06.06.2014

6.1 Entwicklung

6.1.1 Reengineering

Beim Reengineering wird ein altes bzw. bestehendes System ümgewandelt"hinsichtlich der Programmiersprache. Derzeit findet z.B. sehr häufig die Umstellung von COBOL-Programmen statt.

Sommerville bietet eine Beurteilung von Systemen an, die dabei hilft die Entscheidung zu treffen, ob das Reengineering eines Systems sinnvoll ist oder nicht.Dabei sollten nur Systeme herangezogen werden bei denen die Qualität auf einem hohen Niveau ist. Produkte mit geringere Qualität sollten dafür nicht herangezogen werden.

6.2 Test

Tests gehören zur Softwareentwicklung, damit die Qualität sichergestellt werden kann. Tests haben auch immer einen Bezug auf die Anforderungen.

6.2.1 Motivation

Der Hauptgrund für Tests ist das Entdecken von Problemen um diese dadurch beheben zu können. Zudem können Tests dabei helfen die Endanwender von der korrekten Funktionalität der Software zu überzeugen. Wichtig ist allerdings zu wissen, dass Tests nur die Anwesenheiten von Problemen bestätigen, nicht deren Abwesenheit.

6.2.2 Grundlegende Konzepte

Es müssen folgende Definitionen unterschieden werden:

Irrtum: inkorrekte Wahrnehmung oder Aktion eines Menschen

Fehler:inkorrektes Codeelement

Fehlverhalten: unerwartetes Verhalten eines IT-Artefakts

Defekt:Ursache eines Fehlverhaltens Ursprünglich verstand man unter Bugs wirklich den Einfluss von Käfern auf die Hardware, wie der erste dokumentierte Bug von Grace Hoppers 1947 zeigt. Hier sorgte eine Motte für das Fehlverhalten einer Schaltung. Um Defekte zu vermeiden, kann man standardisierte Prozesse einführe die Fehler erkennen an dadurch die Qualität der Software steigern. Um Defekte zu entdecken, kann man Reviews von Spezifikationen und Code sowie Tests durchführen. Standardisierter Prozesse sind unter anderem Code-Reviews und Tests. Bei der Defektbehandlung wird im allgemeinen zwischen Defektvermeidung, Defektentdeckung, Defektbeseitigung und Defektisolierung unterschieden.
Im Testkontext wird zwischen Blackbox-Tests, Whitebox-Tests und Regressionstests unterschieden.

Blackbox-Test: es wird gegen die Spezifikation ohne Kenntnis des Codes getestet.

Whitebox-Test: es wird unter Kenntnis und Verwendung des Codes getestet.

Regressionstest: es wird die erwartete Funktionalität sichergestellt und dass die Software frei von allen bislang bereits beseitigten Fehlern ist.
Dabei werden Testfälle und Testssuits erstellt.

abstrakten Testfall: es werden abstrakte Definitionen eines einzelnen Tests anhand des Testziels, notwendige Vorbereitung, Eingabevariablen und erwartete Ergebnisse und Beobachtungen, abhängig von den Eingabevariablen definiert.

konkrete Testfall: ist eine ausführbare Definition eines einzelnen abstrakten Tests. Er besitzt eine Referenz auf den abstrakten Testfall.

Testsuite: ist eine Sammlung von abstrakten und konkreten Testfällen.

6.2.3 Testprozess

Generischer Testprozess

Es wird festgelegt welche Eingabewerte verwendet werden und welche Testergebnisse erwartet werden. Die Testergebnisse werden in der Analyse herangezogen und mit den erwarteten Ergebnissen verglichen. Erkennt man dabei in der Analyse Abweichungen, kann es auf einen Fehler im Code hinweisen, aber auch, dass in der Vorbereitung des Tests etwas geplant wurde, was sich nicht testen lässt.

Testplanung und Testvorbereitung

In diesem Schritt müssen die Aktivitäten, sprich Sammlung von Informationen über den Testkandidaten, die Definition validierbarer Testziele, Festlegung der Teststrategie, Definition abstrakter Testfälle, Festlegung des anzuwenden Testverfahren und die Ableitung konkreter Testfälle, geplant werden. Die Ergebnisse daraus sind neben den validierbaren Testzielen und den Testmodellen auch die verschiedenen Testfälle.

Testdurchführung

Bei der Testdurchführung werden alle konkreten Testfälle bzw. Testsuiten gemäß der festgelegten Testverfahren ausgeführt. Dabei werden Ergebnisse gesammelt die anschließend ausgewertet werden können.

Ergebnisanalyse und Verfolgung

Bei der Ergebnisanalyse wird ein Soll/Ist-Vergleich der Ergebnisse jedes einzelnen Testfalls durchgeführt. Des weiteren finden eine Vorklärung der Ursache von Abweichungen statt. Das Ergebnis sind identifizierbare Abweichungen (daher Defekte) und eine summarische Auswertung aller Testfälle. Die dafür benötigten Daten sind neben den Messwerten der Tests auch Beobachtungen der Testausführung.

Testbewertung

Bei der Testbewertung werden die Testziele mit den summarischen Auswertungen der Testfälle verglichen. Es liegt ein Ergebnis vor, wenn das Testende erreicht wird (z.B. Testziel erreicht oder alle Testfälle wurden ausgeführt). Hier ist es wichtig das es sich um validierbare Testfälle handelt.

6.2.4 Teststrategien

Motivation

Das planlose Äusprobierenïst die häufigste Form des Testens und ist unzureichend. Es liefert keine reproduzierbaren Ergebnisse, erlaubt keine Aussage über die Qualität der getesteten Software und bietet keinerlei Gewährleistung. Dieser Ansatz ist hochgradig unprofessionell.
Es muss berücksichtigt werden, dass ein vollständiges Testen aller Möglichkeiten meistens theoretisch gesehen nicht möglich ist und auch auf jeden Fall unwirtschaftlich ist.

Checklisten und Benutzungsprofile

Es können für die verschiedensten Inhalte Checklisten angelegt werden. Beispielsweise Funktions–Checklisten, die eine Liste der wesentlichen Funktionen bzw. anderer Eigenschaften beinhalten (Blackbox), oder Struktur-Checklisten, die eine Liste der Systembestandteile beinhaltet (Whitebox). Hier gibt es viele verschiedene Arten von Checklisten. Es gibt theoretisch keine Grenzen.

- Funktions-Checklist
- Struktur-Checklist
- Querschnitts-Checklist

Wichtig bei dem erstellen von Checklisten ist, dass die verwendete Struktur der Checkliste zum Anwendungsfall passt. Denn sichere Aussagen über die Softwarequalität erfordern oft umfangreiche Checklisten.

Partitionen

Es ist meistens theoretisch unmöglich alle Möglichkeiten zu Testen. Partitionen können helfen, Probleme von Checklisten zu lösen bzw. zu lindern.
Grundannahme ist, dass alle Elemente einer Äquivalenzklasse gleich behandelt werden. Es reicht daher, ein beliebiges Element aus einer Äquivalenzklasse zu testen.
Die Anwendung von Partitionen auf Checklisten erhöht den Überdeckungsgrad und verbessert damit die Qualität und reduziert den Gesamtaufwand.
Die Verwendung mehrerer einfacher, von einander unabhängiger und in sich konsistenter Partitionen ist möglich und daher auch sinnvoll. Nachteil ist, dass eine gleiche Gewichtung aller Äquivalenzklassen zu unwirtschaftlichen Verteilung des Testaufwand führt.

Benutzungsprofile

Ein Benutzungsprofil ist eine quantitative Definition der unterschiedlichen Aufrufhäufigkeiten. Beispiele hierfür sind Funktionen für Endbenutzer oder für Administratoren. Damit wird sichergestellt, dass die am häufigsten verwendeten Funktionen auch am intensivsten und regelmäßig getestet werden.
Um Benutzungsprofile definieren zu können, muss zwischen existierenden Produkten, neuen Produkten und neuartigen Produkten unterschieden werden. Bei existierenden Produkten wird die Messung in unterschiedlichen Anwenderumgebungen des aktuellen Produktes und der Vorgängerversion durchgeführt. Bei neuen Produkten kann ein Konkurrenzprodukt herangezogen werden um die Messung durchzuführen. Zudem sollte bei neuartigen Produkten immer eine Einbindung von Marketingstrategen, Softwarearchitekten und Systemarchitekten erfolgen.

Für die Konstruktion von Benutzerprofilen kann die Top-Down-Methode verwendet werden. Hierbei werden Kundenkategorien mit einer Gewichtung nach relativem Benutzungsgrad definiert. Zusätzlich werden die Anwendertypen mit einer Gewichtung der Anwendungshäufigkeit pro Kundenkategorie definiert.

Überdeckungen

Das Grundlegenden Konzepte hinter der Teststrategie Überdeckung ist unter anderem die Partitionierung der Eingaben. Wichtig ist, dass die Partitionierung vollständig und überlappungsfrei ist. Die Grenzen bilden die Teile des Eingaberaums, an dem Unterdomänen zusammenstoßen. Im einfachsten Fall ist diese Grenze linear. Ein Grenzpunkt liegt genau auf einer Grenze und ein Knotenpunkt liegt auf mindestens zwei Grenzen. Es wird dabei angenommen das die Ergebnisse auf Korrektheit überprüft werden können.

Im bereich der Teststrategie Überdeckung spielt das überdeckungsbasierte Testen eine große Rolle.Um überdeckungsbasiertes Testen zu ermöglichen, muss eine Identifikation des Eingaberaums (Eingabevariablen mit ihren Beschränkungen) durchgeführt werden. Dieser Eingaberaum muss in Unterdomänen (Äquivalenzklassen) eingeteilt werden. Des Weiteren müssen diese Unterdomänen analysiert werden in Bezug auf ihre Grenzen. Anschließen müssen die Testpunkte pro Domain evaluiert werden. Der letzte Schritt ist behandelt dann die Ausführung des Tests mit jedem ausgewählten Testpunkt. Dabei können allerdings eine Vielzahl von Problemen auftreten.

- Testpunkte können vom Testkandidaten nicht bearbeitet werden (Ursache hierfür sind unvollständige Spezifikationen oder Implementierungen oder Spezifikationen auf Basis von Unterdomänen)
- Testpunkte haben mehrere widersprüchliche Verarbeitungsregeln (Ursache hierfür sind überlappende Unterdomänen)

- Probleme mit der Grenzdefinition (Ursachen hierfür sind Abschlussprobleme, Grenzverschiebungen mit minimalen Abweichungen (Rundungsfehlern), fehlende Grenzen durch fehlerhafte Partitionierung und überflüssige Grenzen durch übertriebene Partitionierung)

Um überdeckungsbasiertes Testen zu ermöglichen gibt es eine Vielzahl von verschiedenen Ansätzen. Eine Ansatz ist der EPC. Bei der EPC (Extreme Point Combination) Strategie, wird für jede Unterdomäne für jede Dimensionen i des Eingaberaums ein Minimum und Maximum definiert und als Testpunkte dienen alle 4^n möglichen Kombinationen. Mit dieser Strategie ist man im Eindimensionalen auf der sicheren Seite, da Abschlussprobleme, Grenzverschiebungen (nicht alle), fehlende Grenzen und überflüssige Grenzen (abhängig von der Lage des inneren Testpunkts) erkannt werden. Im Mehrdimensionalen allerdings ist diese Strategie nicht zu empfehlen, da unter Umständen erhebliche Probleme abhängig von der Art und Lage der Grenzen entstehen können. Ein anderer Ansatz ist die schwache $Nx1$ Strategie. Mit der schwachen $Nx1$-Strategie können Grenzverschiebungen erkannt werden. Es werden für jede Unterdomäne und für jede Grenze n linear unabhängige Testpunkte auf der Grenze, ein Testpunkt nicht auf der Grenze, ein Testpunkt außerhalb der Unterdomäne, falls die Grenze abgeschlossen ist, ein Testpunkt innerhalb der Unterdomäne, falls die Grenzen offen sind definiert. Zusätzlich wird ein Testpunkt im Inneren der Unterdomäne definiert. Insgesamt hat man dann $(n+1)*b+1$ Testpunkte pro Unterdomäne mit b Grenzen. Der Aufwand dieser Strategie ist sehr groß, doch sie erkennt dafür Abschlussprobleme, Grenzverschiebungen, fehlende Grenzen und die meisten überflüssigen Grenzen (allerdings nicht alle).

7 LV 7 am 20.06.2014

7.1 Teststrategien

Beispiel: Das hier angeführte Beispiel zeigt die Anwendung der $Nx1$ Strategie für unser Anwendungsbeispiel.

$< Stm >:=< Assign > | < Loop > | < Case >$

$x12 = (-b + -sqrt(b^2 - 4ac))/2a$

Des weiteren wies Bernd Wenzel darauf hin, das sich jeder bis zur nächsten Stunde überlegen sollte welche Test mit der $Nx1$ Strategie für unser Beispiel getestet werden müssten.

7.1.1 Überdeckungen

Ein anderer Ansatz ist die schwache $Nx1$ Strategie. Mit der schwache $Nx1$-Strategie können Grenzverschiebungen erkannt werden. Es werden für jede Unterdomäne und für jede Grenze n linear unabhängige Testpunkte auf der Grenze, ein Testpunkt nicht auf der Grenze, ein Testpunkt außerhalb der Unterdomäne, falls die Grenze abgeschlossen ist, ein Testpunkt innerhalb der Unterdomäne, falls die Grenzen offen sind definiert. Zusätzlich wird ein Testpunkt im Inneren der Unterdomäne definiert. Insgesamt hat man dann $(n+1)*b+1$ Testpunkte pro Unterdomäne mit b Grenzen. Der Aufwand dieser Strategie ist sehr groß, doch sie erkennt dafür Abschlussprobleme, Grenzverschiebungen, fehlende Grenzen und die meisten überflüssigen Grenzen (allerdings nicht alle). Eine andere Strategie ist die schwache $1x1$ Strategie. Sie reduziert den hohen Aufwand der $Nx1$ Strategie ohne dabei gravierende Qualitätseinbußen zu haben. Sie erkennt Abschlussprobleme, die Grenzverschiebungen werden meistens vermieden. Es werden fehlende Grenzen erkannt. Sie bietet einen guten wirtschaftlichen Kompromiss.

7.1.2 Endliche Automaten und Markov-Modelle

Die meisten Anwendungsprogramme lassen sich als endliche Automaten mit Ausgabe darstellen. Man versucht auf eine beherrschbare Anzahl an Zustände und Übergänge zu erreichen. Für die Vereinfachung ist es notwendig, die Anzahl der Zustände deutlich kleiner als 100 zu halten. Viele sagen, dass 7 Zustände schon zu viel seien. Für Webapplikationen bietet sich diese Methode zum Testen sehr gut an, da jede Seite als eigener Zustand definiert werden kann. Man kann eine Beschränkung für nur relevante Seiten definieren, wodurch sich die Anzahl der Zustände verringert.

Man muss die Zustände, Übergänge und die I/O-Beziehungen definieren um das Model zu validieren. Für die Testziele ist es relevant zu bewerten, ob alle Zustände erreichbar, vollständig und notwendig sind. Bei den Übergängen ist die Durchführbarkeit ein wichtiger Punkt. Hier sollte darauf geachtet werden, dass keine Wiederholung der Prüfung der Zustandserreichbarkeit der bereits ausgeführter Übergänge stattfindet.

Diese Endlichen Automaten können mittels eines Markov-Modelsl (auch Markov-Kette) getestet werden. Es wird ein endlicher Automat mit Wahrscheinlichkeiten an jedem Zustandsübergang erstellt. Bei einem unvollständigen Modell ist die Summe der Wahrscheinlichkeiten aller ausgehenden Übergänge < 1. Dann handelt es sich dabei um ein unvollständiges Modell. Eine weitere Ausprägung davon stellt das UMM (Unified Markov Model) da.

Die Wahrscheinlichkeit eines konkreten Übergangs ist das Produkt aller Wahrscheinlichkeiten auf dem Pfad vom Start bis zum Ausgangszustand, multipliziert mit der Wahrscheinlichkeit dieses konkreten Übergangs. Bei Schleifen kann dies unangenehm werden.

Das Unified Markov Model (UMM) ist ein kohärenter Satz von hierarchisch geschachtelten Markov-Modellen und ist deutlich besser in der Praxis einsetzbar als das Markov-Modell. Es eignet sich vor allem für alle Arten von statistischen Aussagen (z.B. Zuverlässigkeit, Verfügbarkeit, ...).

7.1.3 Kontrollfluss

Beim Kontrollfluss ist das Testziel die Ausführung aller Pfade im Programm. Der Control Flow Graph (CFG) besteht aus Knoten, welche ausführbare Anweisungen sind (Unterscheidung zwischen Startknoten und Endknoten), und Kanten, welche mögliche Verarbeitungssequenzen darstellen.

Prinzipiell wird versucht alle Pfade auszuführen. Lediglich Schleifen sind davon ausgenommen, da sie ein Problem darstellen. Es musst bei Schleifen getestet werden:

- komm ich überhaupt in die Schleife rein

- funktioniert sie mit einer, zwei, drei,... Durchläufen
- erreiche ich die Grenzfälle

Die Empfehlung hier ist es, den Pfad von hinten aufzubauen, da so die Identifikation von nicht benötigtem Code unterstützt wird.

7.1.4 Datenfluss

Beim Data Dependency Graph (DDG) stellen Knoten Zustände von Variablen dar und die Kanten stehen für die Operationen des Programms.

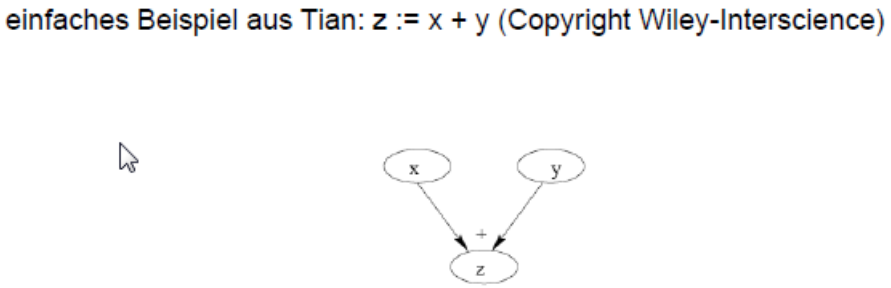

Abbildung 7.1: DDG

Beim DDG wird die Art des Zugriffs auf einen Zustand unterschieden. Es gibt einen lesenden und einen schreibenden Zugriff. Somit ist es möglich Zugriffsfolgen zu definieren, die Probleme darstellen könnten. Zum Beispiel geht von einem Schreib-Lese-Zugriff (S-L) oder einem Lese-Lese-Zugriff keine potentielle Gefahr aus. Ein Schreib-Schreib-Zugriff deutet allerdings auf mögliche Fehler und Ineffizienz hin.

7.2 Testverfahren

Bei den Testverfahren haben wir gelernt aus was ein Testverfahren besteht und in welchen Phasen der Entwicklung welche Testtypen zur Anwendung kommen. Einer dieser Testtypen ist der Unit-Test. Bei den Unit Tests wird ein White Box Test durch den Entwickler realisiert. Es handelt sich dabei um informelles Testen um Fehler zu lokalisieren. Eine andere Art ist der Komponenten-Test. Bei einem Komponenten-Test wird entweder ein White Box Test oder Black Box Test durch den Entwickler realisiert. White Box Tests werden hier eher für prozedurale Sprache verwendet und Black Box Tests vor allem

OO-Sprachen. Bei einem Integrationstest wird ein Black Box Test in Bezug auf die zu integrierenden Komponenten und ein White Box Test in Bezug auf die Integrationsschicht (durch Entwickler) durchgeführt.

Bei einem Systemtest werden Black Box Tests durch professionelle Tester durchgeführt.

Des weiteren gibt es noch Regressionstests, Abnahmetests und Betatests.

Dort wo Black Box Tests durchgeführt werden, spielen Checklisten, Benutzungsprofile, Endliche Automaten eine wichtige Rolle. Bei den White Box Tests ist überdeckungsbasiertes Testen und ein Kontrollfluss wichtig.

7.3 Testwerkzeuge

7.3.1 Debugger

Debuggen bedeutet nicht Testen! Debuggen kann allerdings zum Testen verwendet werden um beispielsweise den für einen Test benötigten Status herzustellen. Ein großer Vorteil ist, dass wir uns so nicht zum planlosen Probieren verführen lassen.

7.3.2 Ausführbare Tests

Dabei handelt es sich um automatisierte Tests. Um das zu realisieren gibt es zum Beispiel JUnit (Java, C), NUnit (CSharp), CPPUnit (C++), JSUnit (JavaScript), PHPUnit (PHP) etc. Hierbei ist wichtig das sich der Testcode und das SuT gegenseitig testen.

7.3.3 Testrahmen

Wichtig ist es denn richtigen Testrahmen zu erstellen. Hierbei sind vor allem Komponenten wie SuT, Testframe und Testplan wichtig. Die Tests können im Bottom-Up Prinzip abgearbeitet werden.

7.3.4 Emulatoren/Simulatoren

Simulator Vereinfachtes Modell eines Systems zum Zweck der Analyse dieses Systems.

Emulator Nachbildung wesentlicher Verhaltensaspekte eines Systems durch ein anderes.

Simulatoren und Emulatoren finden in verschiedenen Bereichen Anwendung. Ein Beispiel wäre dabei EADS-Astrium

7.3.5 Überdeckungsanalyse

Nach der Durchführung eines White-Box Tests muss eine Messung der Testüberdeckung auf Ebene der Anweisungen erfolgen. Dies erfordert allerdings eine Instrumentierung des Codes. Beispiel eines solchen Systems wäre Cobertura.

7.3.6 GUI Capture/Replay

Hierbei handelt es sich um ein Testverfahren über eine Grafische Oberfläche. Hierbei müssen Werkzeuge verwendet werden, die diese Tests automatisiert. Diese Werkzeuge werden Capture und Replay Werkzeuge genannt. Laut Bernd Wenzels Beobachtungen macht das Testen über ein solches System keinen Spaß.

8 LV 8 am 27.06.2014

9 Verifikation vs. Validierung

Verifikation Bei der Verifikation überprüft man die Software gegen ihre Spezifikation.

- Sind die Spezifikationen richtig umgesetzt?
- Wurde das Produkt richtig erstellt?

Validierung Bei der Validierung überprüft man die Software gegen ihre Anforderungen und Erwartungen ihrer Benutzer.

- Wurde das richtige Produkt erstellt?
- Wurden die Benutzeranforderungen richtig und vollständig erfasst, verstanden und in der Spezifikation umgesetzt?

Die Verifikation und Validierung von Software können in dem Maße als äquivalent betrachtet werden, wenn die Anforderungen der Benutzer richtig und vollständig verstanden und dokumentiert und korrekt und vollständig in der Spezifikation berücksichtigt und umgesetzt wurden. Die Verifikation trägt zur Validierung bei, es sind jedoch zusätzliche Maßnahmen für die Validierung notwendig.

Mögliche Fehlerquellen treten auf, wenn der Anwender ein vermeintliches Fehlverhalten beobachten kann. Es kann sich dabei um ein Missverständnis (İt's not a bug, it's a feature!") oder um ein tatsächliches Fehlverhalten handeln (z.B. Programmierfehler, Fehler in der Spezifikation, unerwartete Umgebungseinflüsse,...).

10 Planung der Qualitätsüberprüfung

Prüfverfahren können entweder statisch oder dynamisch sein. Die Ziele dabei sind es Fehler zu Entdecken (=Verifikation) und das Aufbauen von Vertrauen bzgl. der Anwendbarkeit der Software (=Validierung). Sommerville gibt hier einen Prozess für die Fehlerbehandlung vor.

TODO Thomas Grafik S 10?

11 Verifikationstechniken

11.1 Softwareinspektion

Die Motivation bei der Softwareinspektion ist unter anderem die Fehlerlokalisierung auf der Basis von Tests oder auch das gleichzeitig verfolgen weiterer Prüfziele wie beispielsweise die Portierbarkeit, Wartbarkeit oder Eignung von Algorithmen zu inspizieren.

Die Voraussetzungen für die Softwareinspektion ist, dass die genauen Spezifikationen verfügbar sind, die Teammitglieder mit den einzuhaltenden Standards und Normen vertraut sind und mindestens eine kompilierbare Codeversion verfügbar ist.

Sommerville definiert einen Standardablauf für Softwareinspektionen. Der Zeitbedarf für den Überblick beträgt ca. 500 Anweisungen/Stunden und einer Gesamtdauer von maximal 2 Stunden. Für die individuelle Vorbereitung werden noch circa 125 Anweisungen/Stunde benötigt und für die Inspektionssitzung nochmals circa 100 Anweisungen/Stunde, maximal aber 2 Stunden.

TODO THomas Grafik S 15?

11.2 Automatisierte statische Analyse

Bei der Softwareinspektion handelt es sich um eine Form der statistische Analyse. Man verwendet für die Umsetzung oft die Hilfsmittel der Softwareinspektion wie Checklisten und Heuristiken. Hier gibt es ein sehr hohen Automatisierungspotential.

Werkzeuge für die statische Analyse sind die Erkennung Anomalien im Code, wobei nicht jede Anomalie ein Fehler ist. Sie sollten aber alle in einem Review besprochen werden.

Es werden Fehlerklassen wie Datenfehler (z.B. uninitialisierte Variablen verwenden,...) Steuerungsfehler (z.B. unerreichbarer Code,..), Ein-Ausgabefehler (z.B. zweimal ausgegebene Variable ohne zwischenzeitliche Zuweisung,..), Schnittstellenfehler (z.B. falsch zugeordnete Parametertypen,..) und Speicherverwaltungsfehler (z.B. nicht zugewiesene Zeiger,..) eingeführt.

Bei der statischen Analyse werden verschiedene Phasen abgearbeitet.

Analyse der Steuerung: dies beinhaltet das Erkennen von Schleifen (nicht-explizite Schleifen, mehrere Eintrittspunkte/Austrittspunkte), von unerreichbarem Code (nach unbedingtem GOTO, nicht erfüllbare Bedingungen) und eine Pfadanalyse für Überdeckungstests.

Analyse der Datenverwendung: hier wird überprüft ob nicht initialisierte Variablen gelesen werden, ob nicht gelesene Variablen überschrieben werden, ob Variablen unbenutzt sind oder ob redundante Tests ausgeführt werden.

Analyse der Schnittstellen: hier werden die die Anzahl und die Typen der Parameter in Deklarationen und Aufrufen, deklarierte aber nicht aufgerufene Prozeduren und Funktionen sowie nicht verwendete Funktionsergebnisse ermittelt/überprüft.

Analyse des Informationsflusses: hier geht es primär um die Erkennung der Abhängigkeiten zwischen Eingabevariablen und Ausgabevariablen (z.B. unbenutzte Eingaben).

11.3 Formale Methoden

Die Vision ist es formale Methoden als die ultimative statische Verifikationsmethode zu verwenden. Die Voraussetzung hierfür ist die formale Spezifikation. Der Aufwand hierfür ist allerdings sehr hoch und wird daher praktisch nur für kritische Systeme verwendet. Vielen Kunden fordern dies allerdings inzwischen (z.B. im militärischen Bereich).

Es stehen inzwischen automatische Überprüfungen der Spezifikationen auf Inkonsistenzen und die automatische Generierung von Code aus den Spezifikationen zur Verfügung.

Es gibt allerdings noch einige Probleme. Die Spezifikationen entsprechen meistens nicht den Benutzeranforderungen (Fehler, Lücken,...), da die formalen Spezifikationen oft unverständlich für die Mehrheit der Anwender ist. Die Komplexität und der Umfang der Beweise stellt ein weiteres Problem dar und ein unzutreffendes Nutzungsmuster macht Beweise ungültig.

11.4 Softwaretest

Das Ziel ist es Fehler zu erkennen beispielsweise via Whitebox-Tests. Wurde bereits behandelt.

TODO Thomas vll link zur LV?

12 Validierungstechniken

12.1 Allgemeine Validierungsmaßnahmen

Analysephase: die Qualität der Anforderungen werden sichergestellt. Dabei werden Anforderungs-Interviews mit einer Vielzahl unterschiedlicher Beteiligt bzw. Betroffener durchgeführt und Reviews der Anforderungen (durch Beteiligte/Betroffene, Experten des Anwendungsgebiets und Experten für Systemeigenschaften wie Zuverlässigkeit,..)

Entwicklungsphase: es werden Design-Reviews mit Beteiligten/Betroffenen durchgeführt. Dabei werden ein GUI-Prototyp und Interaktionsszenarios vorgestellt.

Validierungsphase:

Einsatzphase:

12.2 Softwaretest

12.3 Validierung der Zuverlässigkeit

12.4 Validierung der Betriebssicherheit

12.5 Validierung der Systemsicherheit

13 Zusammenfassung

Dies hier ist ein Blindtext zum Testen von Textausgaben. Wer diesen Text liest, ist selbst schuld. Der Text gibt lediglich den Grauwert der Schrift an. Ist das wirklich so? Ist es gleichgültig, ob ich schreibe: „Dies ist ein Blindtext“ oder „Huardest gefburn“? Kjift – mitnichten! Ein Blindtext bietet mir wichtige Informationen. An ihm messe ich die Lesbarkeit einer Schrift, ihre Anmutung, wie harmonisch die Figuren zueinander stehen und prüfe, wie breit oder schmal sie läuft. Ein Blindtext sollte möglichst viele verschiedene Buchstaben enthalten und in der Originalsprache gesetzt sein. Er muss keinen Sinn ergeben, sollte aber lesbar sein. Fremdsprachige Texte wie „Lorem ipsum“ dienen nicht dem eigentlichen Zweck, da sie eine falsche Anmutung vermitteln.