

Was ist ein **fault**?

Was ist ein **failure**?

Charakteristik von technischer Disziplin
(Engineering Discipline)

Welche Symptome traten in der
Softwarekrise auf?

Charakteristik von Software

Warum ist Software schwierig zu
entwickeln?

Software-Entwicklungs-Mythen

Was ist Software Engineering?

<div># 2</div> <div>Antwort</div> <div> <p>Ein Fehler in der gelieferten Software, die nicht mehr ihren Spezifikationen entspricht (nicht mehr das tut, was zu erwarten wäre). Bzw.: Die Unfähigkeit eines Systems oder einer Komponente seine geforderten Funktionen innerhalb der Leistungsanforderungen durchzuführen. (Atomkraftwerk explodiert)</p> </div>	<div># 1</div> <div>Antwort</div> <div> <p>Fehler, der von Menschenhand während der Laufzeit der Software getätigt wurde (fehlerhaftes Design, Anforderungen, Coding) (Anschalten der Kühlung (Atomkraftwerk) nicht korrekt programmiert)</p> </div>
<div># 4</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> Produkte wurden zu spät geliefert (hohe Kosten) Projekte überschritten ihr Budget (hohe Kosten, Verschwendung von Ressourcen) Produkte taten nicht das, was sie tun sollten/wozu sie entwickelt wurden (ineffizient, hohe Kosten) Produkte waren defekt (hohe Kosten (failure, Wartung), ethische Überlegungen) Projekte wurden beendet, bevor sie abgeschlossen waren (Verschwendung von Ressourcen) </div>	<div># 3</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> gut verstandene Technologien gut definierte Prozesse Vorhersagbarkeit eines Ergebnisses eines Prozessstils Wiederholbarkeit von Prozessschritten </div>
<div># 6</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> es gibt kein ähnliches System bisher (Probleme sind unbekannt, Annahmen zur Umgebung können falsch sein) Anforderungen sind nicht gut/ausreichend verstanden/formuliert Anforderungen verändern sich im Laufe der Entwicklung des Systems komplexe Interaktion Natur des Systems: nebenläufige Systeme (Deadlocks, ...), eingebettete Systeme (Hardwareinteraktion, Timing, ...), Informationssysteme (Komplexität, ...) Software ist einfach zu verändern („code and fix“) Software ist unauffällig (entweder fällt sie aus, oder nicht) </div>	<div># 5</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> Software wird entwickelt nicht hergestellt Software leiert nicht aus (aber: Veränderung von Anforderungen oder der Umwelt) Software ist komplex Software ist ein bestimmender Systemfaktor (bis zu 80% des Entwicklungsaufwandes) </div>
<div># 8</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> Kommunikation zwischen einer großen Anzahl von Parteien (Kunde, Endbenutzer, Software Designer, Entwickler, ...) wird aufrechterhalten Die Basis der Softwareproduktion ist der Entwicklungsansatz. Design der Software als Prozess, der folgendes unterstützt: <ol style="list-style-type: none"> Korrektheit und Zuverlässigkeit Kosteneffizienz Komplexität des Projektes Langlebigkeit des Produktes, Lebenszyklus, Veränderungen Kommunikation unter den Parteien </div>	<div># 7</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> Management: <ul style="list-style-type: none"> Standard-Bücher, -Software, -Tools ⇒ Software ist schwierig zu standardisieren Dem Zeitplan hinterher? Programmieren einstellen ⇒ Bemühung neue Leute einzustellen Kunde: <ul style="list-style-type: none"> allgemeine Erklärung der Ziele sind ausreichend Veränderungen (auch in den Anforderungen) sind einfach zu implementieren Anwender: <ul style="list-style-type: none"> Sobald das Programm läuft ist der Job erledigt Bis das Programm läuft gibt es keine Möglichkeit die Qualität des Systems zu überprüfen Nur ein arbeitsfähiges Programm ist lieferbar </div>

Definition des Software Engineering
Prozessmodelles

Built-and-Fix-Modell

Wasserfall-Modell (Bild)

WasserFall-Modell: System Design

Wasserfall-Modell: Requirements

Wasserfall-Modell: Design

Wasserfall-Modell: Implementation (und
Testing)

Wasserfall-Modell: Integration (und
Testing)

10

Antwort

- keine Prozessschritte
- keine Unterteilung von Bedenken
- keine Möglichkeit mit Komplexität umzugehen

—————> Development
 - - - - -> Maintenance

9

Antwort

alle Phasen des Software Engineeringprozesses: strikte Anwendung des Prozessmodells (wohldefinierter Input/Output)

12

Antwort

Probleme und Systemanforderungen, Realisierbarkeitsstudie, Definieren von Haupt-Subsystemen (zugeordnet zu Hard-/Software), System Design Document (SDD, informell, mit dem Kunden, manchmal: Bedienungsanleitungen, Benutzerschnittstellen, Testpläne)

11

Antwort

14

Antwort

„Wie“ das System arbeitet, architektonisches (high level) Design (zerteilt das Problem in Komponenten, globale Datenstrukturen, interne Schnittstellen), detailliertes Design (algorithmisches Design, interne Datenstrukturen, Programmiersprache(n))

13

Antwort

Anforderungsanalyse („Was“ soll das System tun (nicht „wie“), Softwareanforderungen beschreiben die beobachtbaren externen Verhalten (funktional, non-funktional)), Software Requirement Specification (SRS)

16

Antwort

Integrieren der getesteten Module zur Bildung des Systems, *integration testing*, Bestätigung (Validation), Kundenakzeptanztests

15

Antwort

Übersetzung der Designmodule in Code, Testen der Module in Isolation

Wasserfall-Modell: Maintenance
(Instandhaltung)

Wasserfall-Modell: Mängel

V-Modell: Bild

Software Qualitäten: Korrektheit

Wasserfall-Modell: Vorteile

Wasserfall-Modell: Alternativen

Software Qualitäten

Software Qualitäten: Zuverlässigkeit

# 18	Antwort	# 17	Antwort
<ol style="list-style-type: none"> Dijkstra: Definition der einzelnen Aufgaben, (Aufteilung der Gedanken) Aufteilen von komplexen Designproblemen in kleinere Einheiten ⇒ vereinfacht die Teamarbeit/Reproduzierbarkeit Spezifikation und Dokumentation ⇒ erzwingt Dokumentation, vereinfacht das Testen (entgegen der Anforderungsspezifikation) weitere Konzepte: <ol style="list-style-type: none"> Verifikation/Validierung (Vergleicht Zwischenergebnisse von Anforderungen und Design) Prototyping (mock-up (Modell) ist früh verfügbar, reduziert Risiken) evolutionäres Prozessmodell (Unterbringen von Veränderungen) 		<p>Produktbereitstellung, Wartungsarbeiten (korrekt, adaptiv, perfekt), Projekt abschließen</p>	

# 20	Antwort	# 19	Antwort
<ol style="list-style-type: none"> modifiziertes Wasserfallmodell V-Modell, V-Modell XT evolutionäre Prozess-Modelle Spiralmodell Rational Unified Process (RUP) Agile Prozesse ... 		<ol style="list-style-type: none"> keine Rückkopplungsschleifen dokumentorientiert, unflexibel großer Zeitabstand zwischen Beginn und Abschluss 	

# 22	Antwort	# 21	Antwort
<ol style="list-style-type: none"> Korrektheit Zuverlässigkeit Robustheit Wartbarkeit Performance Wiederverwendbarkeit Kompatibilität 		<pre> graph TD RA[Requirements Analysis] --> SD[System Design] SD --> PD[Program Design] PD --> C[Coding] C --> UI[Unit and Integration Testing] UI --> ST[System Testing] ST --> AT[Acceptance Testing] AT --> OM[Operation and Maintenance] AT -.-> RA ST -.-> SD UI -.-> PD </pre>	

# 24	Antwort	# 23	Antwort
<p>die Software garantiert eine gewisses Level an Qualität (siehe Korrektheit)</p>		<p>die Software verhält sich entsprechende der Anforderungsspezifikation</p>	

Software Qualitäten: Robustheit

Software Qualitäten: Wartbarkeit

Software Qualitäten: Performance

Software Qualitäten:
Wiederverwendbarkeit

Software Qualitäten: Kompatibilität

Softwareentwicklungsprinzipien

Softwareentwicklungsprinzipien: Strenge

Softwareentwicklungsprinzipien:
Formalität

<div># 26</div> <div>Antwort</div> <div>die Software ist einfach aufrechtzuerhalten und zu erweitern</div>	<div># 25</div> <div>Antwort</div> <div>die Software verhält sich auch „vernünftig“ in unerwarteten Umständen (Eingang, Stromausfall, ...)</div>
<div># 28</div> <div>Antwort</div> <div>Wiederverwendbarkeit von zuvor Verwendetem, getesteter und geprüfter Code</div>	<div># 27</div> <div>Antwort</div> <div>das System ist benutzbar (z.B. Eingangsreaktionszeit)</div>
<div># 30</div> <div>Antwort</div> <div><div>1. Strenge</div><div>2. Formalität</div><div>3. Trennung von Bedenken</div><div>4. Abstraktion</div><div>5. Modularität</div><div>6. Allgemeinheit</div></div>	<div># 29</div> <div>Antwort</div> <div>Standarisierung, Schnittstellen, ...</div>
<div># 32</div> <div>Antwort</div> <div>benutzen von (mathematischen) formalisierbaren Methoden und Notationen</div>	<div># 31</div> <div>Antwort</div> <div>benutzen einer Methode und konsequent anwenden auf jeden Schritt</div>

Softwareentwicklungsprinzipien:
Trennung von Bedenken

Softwareentwicklungsprinzipien:
Abstraktion

Softwareentwicklungsprinzipien:
Modularität

Softwareentwicklungsprinzipien:
Allgemeinheit

Was sind Anforderungen?

Benutzer- (Auftraggeber-)
Anforderungen

Systemanforderungen

Typen von Anforderungen

<div># 34</div> <div>Antwort</div> <div>trennen der Anliegen der wichtigen Aspekte von den Anliegen der unwichtigen Aspekte</div>	<div># 33</div> <div>Antwort</div> <div>sich mit verschiedenen Aspekten in separaten Schritten befassen</div>
<div># 36</div> <div>Antwort</div> <div>konzentrieren auf die Entdeckung von generellen Problemen</div>	<div># 35</div> <div>Antwort</div> <div>zerteilen des Problems in unabhängige Module (siehe Trennung von Bedenken)</div>
<div># 38</div> <div>Antwort</div> <div>Aussagen in natürlicher Sprache mit Diagrammen</div>	<div># 37</div> <div>Antwort</div> <div><div>1. „A requirement is a condition or capability that must be met or professed by a system component to satisfy a contract, standard, specification or other formally imposed document.“ (ANSI/IEEE Standad 729-1983) Eine Anforderung ist eine Bedingung oder Fähigkeit, die eine Systemkomponente erfüllen muss um einen Vertrag, einen Standard, eine Spezifikation oder eine anderes formales, verlangtes Dokument zu erfüllen/befriedigen.</div><div>2. Anforderungen sind eine Beschreibung der von außen sichtbaren Verhalten, das „was“</div><div>3. Anforderungen sind Interaktionen zwischen dem System und dem systemrelevanten Teil der Umwelt</div></div>
<div># 40</div> <div>Antwort</div> <div><div>1. funktionale</div><div>2. nicht-funktionale<div>a) Produktanforderungen</div><div>b) Unternehmensanforderungen</div><div>c) externe Anforderungen</div></div><div>3. Domainanforderungen</div></div>	<div># 39</div> <div>Antwort</div> <div>strukturiertes Dokument, detaillierte Beschreibung der Systemfunktionen, Dienstleistungen und optionalen Einschränkungen; kann Teil des Vertrages zwischen Auftraggeber und Auftragnehmer</div>

funktionale Anforderungen

nicht-funktionale Anforderungen:
Diagramm

nicht-funktionale Anforderungen

nicht-funktionale Anforderungen: Typen

Domainanforderungen

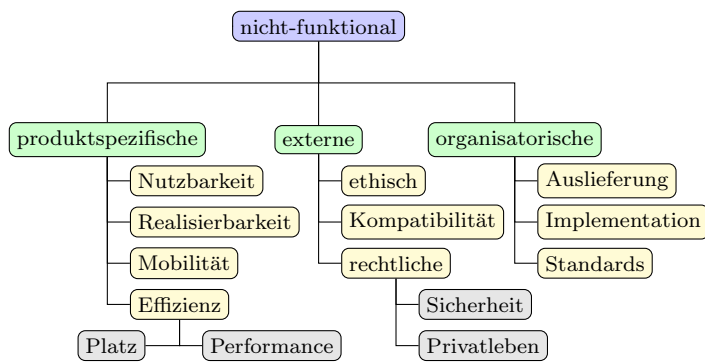
Anforderungsspezifikationseigenschaften

Aktivitäten während der
Anforderungsstufe

Aktivitäten während der
Anforderungsstufe: Diagramm

42

Antwort



41

Antwort

1. high-level „was“ das System tut, nicht wie
2. Aussage darüber, welche Services das System anbieten sollte
3. Interaktion zwischen System und Umwelt (Systemzustände, I/O)
4. wie sich das System in besonderen Situationen verhalten sollte
5. beschreibt den Service des System im Detail
6. oft beschrieben mit soll/sollte
7. „funktionale Anforderungen legen fest, welche Dienste (aus Sicht des Benutzers) das System anbieten soll/welche Aufgaben es erfüllen soll
8. eindeutig/widerspruchsfrei
9. zentrale Vorgaben für die Systementwicklung
10. beschrieben mit Use-Cases

44

Antwort

1. Produktanforderungen:
 - a) Geschwindigkeit, Speicherbedarf
 - b) akzeptable Fehlerquoten
 - c) Transportierbarkeitsanforderungen
 - d) Anforderungen bezüglich Benutzbarkeit
2. Unternehmensanforderungen:
 - a) Vorschriften für Entwicklung (bestehende Standards, spezielle Anwendungen)
 - b) Umsetzungsanforderungen (Programmiersprache, Entwurfsmethode, Lieferanforderungen)
3. externe Anforderungen:
 - a) Kompatibilität (Ausführbarkeit auf div. Geräten/ Betriebssystemen, Sprachpakete)
 - b) rechtliche Anforderungen (Datenschutz, Speicherung)
 - c) ethische Anforderungen (Langzeitspeicherung, keine fremden Kundendaten anzeigen, freundliches Design)

43

Antwort

1. falls nicht erfüllt, kann das System unbrauchbar/unbenutzbar sein (können sich auf wichtige Systemeigenschaften beziehen)
2. nicht primär den Funktionen/Services des Systems zugeordnet, sondern zu Qualität und zusätzlichen Charakteristiken (Quantitativ) / selten an einzelne Systemfunktionen gebunden
3. oft relevanter als funktionale Anforderungen (Unbedienbarkeit, ...)
4. oft allgemein formuliert (kann später zu Problemen führen)
5. direktes Überprüfen schwer (Tests und Metriken (festlegen))
6. Einschränkungen der Funktionen/Services (können Beschränkungen definieren, „sichtbare“ Eigenschaften):
 - Zuverlässigkeit (*Verfügbarkeit, Integrität, Sicherheit*), Genauigkeit der Ergebnisse, Performance/Timing, Mensch-Computer-Interface-Themen, körperliche und Betriebseinschränkungen, Übertragbarkeit und Kompatibilität, Antwortzeit, Speicheranforderungen, Standards, ...
 - auch: bes. System, Prog.sprache oder Entwicklungsmethode

46

Antwort

1. Korrektheit: Tatsachen in der Anforderungsspezifikation \Rightarrow erforderliche Eigenschaften des Systems
2. Eindeutigkeit: alle Spezifikationen lassen nur eine Interpretation zu
3. Vollständigkeit: jede Eigenschaft, die erforderlich für das System ist, wird in der Spezifikation ausgedrückt, die Reaktion des Softwaresystems auf alle Arten von möglichen Eingabewerten ist spezifiziert, ...
4. Überprüfbarkeit: es gibt einen effektiven (manuell/automatisch) Prozess, um zu überprüfen, ob ein Softwareprodukt die erforderlichen Eigenschaften erfüllt (formale Überprüfung (mathematisch) oder Bestätigung (Model Checking, Testing, Simulation)), oft nicht möglich für alle Anforderungen
5. verfolgt/nachvollziehbar: Herkunft aller Anforderungen ist klar, Anforderungsspezifikation ist bearbeitet, so dass es einfach ist, auf eine Anforderung zu referenzieren (Aufzählung)
6. unabhängige Gestaltung: Anforderungsspezifikation erfordert keine spezifische Software/Architektur/Algorithmen

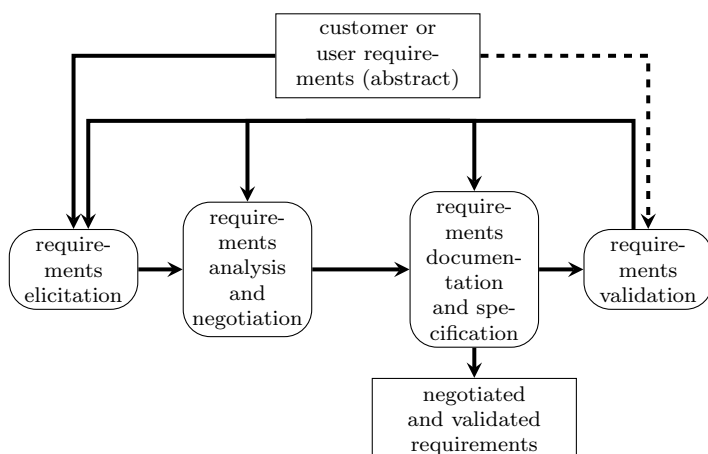
45

Antwort

1. falls nicht erfüllt, könnte es sein, dass das System nicht in der Domain funktioniert (undurchführbar)
2. abgeleitet von Applikations-Domain
3. Charakteristiken/Eigenschaften der Domain
4. Einschränkungen von existierenden Anforderungen
5. definieren spezifische Berechnungen
6. können in einer Domain-spezifischen Sprache ausgedrückt werden \Rightarrow oft schwer zu verstehen
7. oft implizit \Rightarrow schwer herauszufinden

48

Antwort



47

Antwort

1. Anfangspunkt: Kundenanforderungen (abstrakt), Systemspezifikationsdokument (Hardware und Software) (SSD)
2. Aktivitäten:
 - Anforderungserhebung (Interviews, Szenarien, Marktbeobachtung, ...), bestimmen, welche der möglichen widersprüchlichen Anforderungen wichtig sind
 - Anforderungsdokumentation und -spezifikation (verständliches Anforderungsdokument)
 - Anforderungsbestätigung (Konsistenz, Vollständigkeit, Übereinstimmung von dokumentierten Anforderungen und den abstrakten Kunden- oder Benutzeranforderungen)
3. soziale Aktivität: nicht eine einzige Person weiß alles über das System \Rightarrow Kommunikation ist nötig \Rightarrow schwierig (technische Sprache, Unklarheiten, dem Kunden nicht bekannte Anforderungen, Persönlichkeiten)

Interviews/ strukturierte
Befragungsaufgaben

Lernen von existierenden Systemen

Unsicherheit von Anforderungen

Anforderungen sammeln

FAST (Facilitated Application
Specification Technique/Erleichterte
Anwendungsspezifikationstechnik)

ANsätze zu FAST

Objektorientierte Analyse

Objektorientierte Analyse (Infos)

<div># 50</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> 1. Analyse von Benutzeranleitungen 2. benutzen/spielen mit existierenden Systemen 3. Marktanalyse (konkurrierende Systeme, Marktforschung, welche Eigenschaften eingebunden werden sollen) 4. Reverse Engineering </div>	<div># 49</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> 1. context-of-system: wieso wir dieses System entwickeln, wer die Benutzer sind, kritische Funktionalität, Anforderungen 2. open-ended questions: produziert eine große Menge an Informationen, falls vorher nicht viel bekannt war 3. close-ended questions: spezifische Fragen 4. rephrase questions: stellt sicher, dass die Fragen richtig verstanden wurden/Inkonsistenz/Unklarheiten </div>
<div># 52</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> 1. erzeuge Ideen (frei von Kritik/Urteil) ⇒ so viele Ideen wie möglich 2. diskutieren, überarbeiten, organisieren der Ideen 3. bewerten, priorisieren </div>	<div># 51</div> <div>Antwort</div> <div> <p>Wenn der Kunde nicht weiß, was er wirklich braucht/will ⇒ Prototypen. Der Prototyp sollte früh gebaut werden (nur wenn dies passiert, ist es schneller einen Prototyp zu entwickeln, als das System zu bauen) und zur Anforderungsbestimmung benutzt werden. Prototyping ist auch ein Teil von verschiedenen Lebenszyklen und Prozessmodellen (Spiral-Modell).</p> </div>
<div># 54</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> 1. JAD (IBM), die Methode „Performance Resources“ </div>	<div># 53</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> 1. überwinden des wir/die-Denkens (Entwickler, Benutzer, Kunde) 2. Team-orientiert (Zusammenarbeit) 3. Richtlinien: <ol style="list-style-type: none"> a) Teilnehmen am ganzen Treffen/Meeting ist ein Muss b) Teilnehmer sind gleichberechtigt c) Vorbereitung ist wichtiger als das Meeting d) Vor-Meeting-Dokumente sind nur „vorgeschlagen“ e) externer Standort wird bevorzugt f) setze eine Agenda und behalte sie bei g) keine technischen Details </div>
<div># 56</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> 1. Use-Case-Ansatz (von den meisten UML-basierten Ansätzen verfolgt) 2. sprachliche Analyse Methode (Anforderungen in natürlicher Sprache ⇒ umwandeln in objektorientiertes Analysemodell, mehr traditioneller objektorientierter Ansatz) 3. identifizieren von Entitätsojekten (dauerhafte Informationen werden durch das System verfolgt) 4. identifizieren von Randobjekten (Interaktionen zwischen dem System und dem Akteur) 5. identifizieren von Kontrollobjekten (zuständig für die Realisierung von Use Cases) 6. identifizieren von Associations, Aggregationen, Attributen 7. modellieren des zustandsunabhängigen Verhalten von Objekten 8. Use Cases auf das Sequenzdiagramm mappen 9. modellieren von vererbten Beziehungen 10. bewerten des Analysemodells </div>	<div># 55</div> <div>Antwort</div> <div> <p>Mapping auf das Wasserfallmodell:</p> <ol style="list-style-type: none"> 1. Anforderungen sind die objektorientierte Analyse 2. architektonisches Design ist das Systemdesign 3. detailliertes Design ist das Objektdesign <p>Objektorientierte Software Engineering Prozesse sind eher kontinuierlich:</p> <ol style="list-style-type: none"> 1. welche Objekte sind interessant/wichtig 2. was tun sie 3. UML Use Cases, Sequenzdiagramme, Klassendiagramme </div>

Was ist ein Objekt?

Objektorientiertes Software Engineering

Softwaremodelle

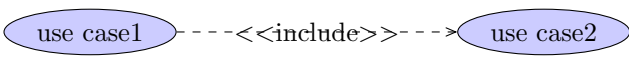
Analyse mit UML

Statische UML Elemente

Dynamische UML Elemente

Use Case Diagramme (Infos)

Use Case Diagramme <<include>>

<div># 58</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> Software ist definiert mit: <ol style="list-style-type: none"> Objekten: Einheiten, die mit Dingen in der realen Welt korrespondieren Klassen: abstrakte Objekte Objekte kapseln Daten (Schnittstelle zu den Daten) die Schnittstelle ist relevant, nicht die Implementation Wiederverwendbarkeit durch Vererbung Abstraktion durch Polymorphismus </div>	<div># 57</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> diskrete Einheit mit gut definierter Grenze und einer Identität, die Zustand und Verhalten kapselt Konzept, Abstraktion oder eine Sache mit gestärkten Grenzen und Bedeutung für die Probleme, die auf der Hand liegen dienen zwei Zwecken: Förderung des Verständnisses der realen Welt und eine konkrete Grundlage für die Implementation am Computer alle Objekte haben eine Identität und sind unterscheidbar </div>
<div># 60</div> <div>Antwort</div> <div>visuelle Notation für Analyse und Design</div>	<div># 59</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> Softwaresysteme sind komplex: Aufteilen in Teilprobleme Softwaremodelle sind die Abstraktion der realen Welt Benutzung: <ol style="list-style-type: none"> frühe Lebenszyklusphasen: Bewerten der Eigenschaften des realen Systems (entlocken, dokumentieren, verifizieren, validieren der Anforderungen, Simulation) Entwurfsphase: Dokumentarchitektur, beurteilen von Leistung/ Verhalten Umsetzung (Implementation)/ Codingstufe: automatisch synthetisierender Code (Klassenskelette, Verhalten der Zustandsmaschine (State Machine)) Testphase: Was wird getestet (Anforderungsmodelle) Wartungsebene: dokumentieren des Systems für die <i>Nachwelt</i> </div>
<div># 62</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> Sequenzdiagramme Zustandsgrafik (State Chart Diagram) Dynamische Modellierung </div>	<div># 61</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> Use Case Diagramme Klassendiagramme </div>
<div># 64</div> <div>Antwort</div> <div> <p>ein Use Case benutzt die Funktionalität eines anderen Use Cases (Use Case 1 benutzt Use Case 2)</p>  <pre> graph LR UC1([use case1]) -.-> <<include>> UC2([use case2]) </pre> </div>	<div># 63</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> oft der erste Schritt in einem objektorientierten Entwicklungsprozess ein Use Case repräsentiert ein Anwendungsszenario (atomar) dokumentiert die Funktionalität, die das System zur Verfügung steht (was das System tut) Akteure (etwas oder jemand, der mit dem System interagiert, stellt Input und Output zur Verfügung) und Haupt-Use Cases (Interaktion zwischen Akteur und System) </div>

Use Case Diagramme <<extend>>

Klassendiagramm („Bestandteile“)

Klassendiagramm (Eigenschaften)

Klassendiagramm (Verbindungen unter Klassen)

Klassendiagramm: Association

Klassendiagramm: Dependencies

Klassendiagramm: Aggregation

Klassendiagramm: Komposition

66

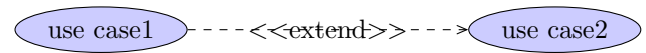
Antwort

1. Phänomen (Objekt in der Welt, wie es in der Domäne wahrgenommen wird)
2. Konzept (Eigenschaften eines Phänomen) ist ein 3-Tupel: Name (unterscheidet sie von anderen), Zweck (Eigenschaft, die bestimmt ob ein Phänomen Teil des Konzeptes ist), Elemente (Phänomene, die Teil des Konzepts sind)
3. Typ (Abstraktion im Kontext des Programmierens (name: int)), Beispiel: **SimpleWatch**
4. Klasse (Abstraktion im Kontext der objektorientierten Sprachen, kapselt Zustand(Variablen) und Verhalten(Methoden))
5. Instanz (existierende Instanz einer Klasse), Beispiel: **myWatch:SimpleWatch**

65

Antwort

ein Use Case benutzt die Funktionalität eines anderen Use Cases (Use Case 1 benutzt Use Case 2) im Ausnahmefall bzw. optional (nicht zwingend wie bei include)



68

Antwort

1. Association: eine semantische Bedingung oder Einschränkung als Ausdruck repräsentiert
2. Abhängigkeiten (Dependencies):
3. Aggregation
4. Komposition
5. Generalisierung
6. Sichtbarkeit (+ public, - private, # protected)

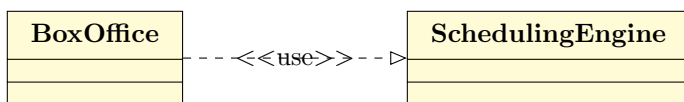
67

Antwort

1. Allgemeingültigkeit (die am allgemeinsten gehaltenen Aspekte des Problems)
2. Abstraktion (nicht einzelne Objekte, sondern das was sie gemeinsam haben)
3. „Die Beschreibung für eine Menge an Objekten, die die gleichen Attribute, Beziehungen und Verhaltensweisen teilen. Eine Klasse repräsentiert ein Konzept, indem das System modelliert wurde.“
4. wird während der Anforderungsanalyse benutzt (modellieren von problematischen Domainkonzepten), Systemdesign (modellieren von Untersystemen und Schnittstellen), Objektdesign (modellieren von Klassen in einer Programmiersprache)

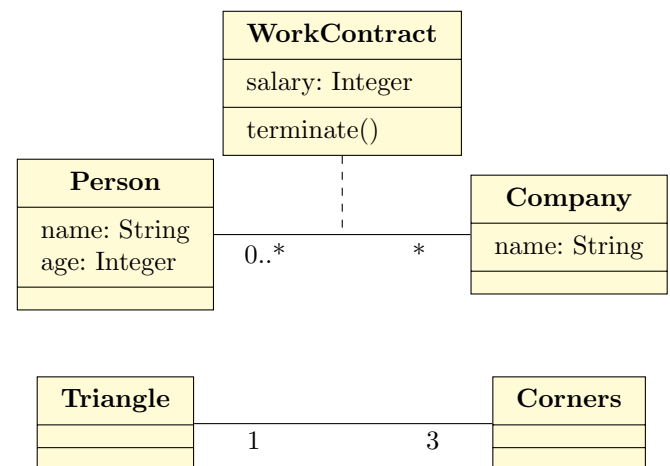
70

Antwort



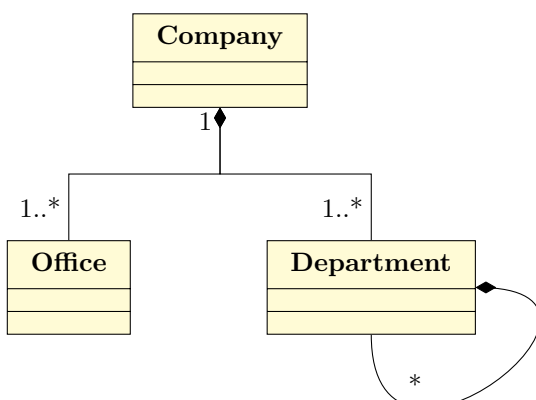
69

Antwort



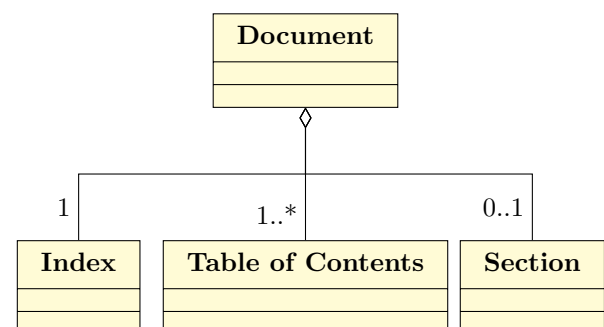
72

Antwort



71

Antwort



Klassendiagramm: Generalisierung

Sequenzdiagramm (Infos)

State Chart Diagramm

Dynamische Modellierung

Anforderungsspezifikation (Requirement Specification)

Inhalt eines SRS (DIN-Norm)

Designziele

klassisches Design

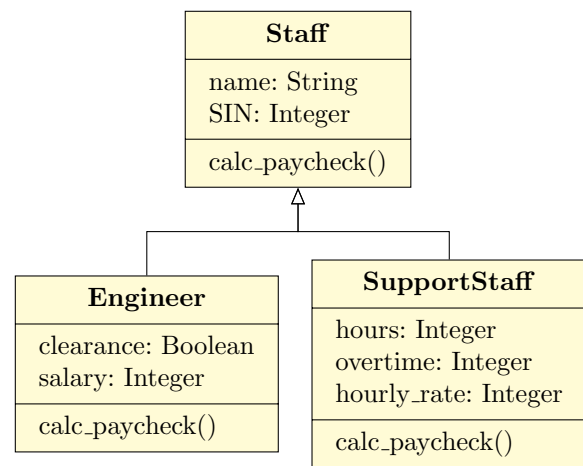
74

Antwort

1. wird während der Anforderungsanalyse benutzt (dynamisches Verhalten definieren), Design (dokumentieren der Unterschnittstellen), gut für Echtzeitdaten
2. Bestandteile:
 - a) Klassen \Rightarrow Säulen
 - b) Nachrichten \Rightarrow Pfeile
 - c) Rückgabewerte \Rightarrow gestrichelte Pfeile
 - d) Bestätigungen/Aktivierungen \Rightarrow schmale Rechtecke
 - e) lifelines \Rightarrow gepunktete Linien

73

Antwort



76

Antwort

Sammlung von State Chart Diagrammen

75

Antwort

Beschreibung einer State Machine (nur für dynamisch interessante Objekte)

78

Antwort

1. Zielbestimmung (Muss-, Wunsch-, Abgrenzungskriterien)
2. Produkteinsatz (Anwendungsbereiche, Zielgruppen, Betriebsbedingungen)
3. Produktübersicht
4. Produktfunktionen (genau & detailliert)
5. Produktdaten (langfristig zu speichernde Daten aus Benutzersicht)
6. Produktleistungen (Anforderungen an Zeit, Genauigkeit)
7. Qualitätsanforderungen
8. Benutzeroberfläche (grundlegende Anforderungen, Zugriffsrechte, evtl Mock-Up's)
9. funktionale Anforderungen
10. nicht-funktionale Anforderungen (Gesetze, Normen, Sicherheitsanforderungen, Plattformabhängigkeiten)
11. technische Produktumgebung (Software/Hardware, organisatorische Rahmenbedingungen, Schnittstellen)

77

Antwort

„Eine Spezifikation, die die Anforderungen für ein System oder eine Systemkomponente setzt; ... typischerweise sind funktionale Anforderungen, Anforderungen an Performance, Schnittstellen und Design enthalten, sowie Entwicklungsstandards.“

„Eine *Software Requirement Specification* ist ein Dokument, das eine vollständige Spezifikation enthält, was das System tut ohne zu sagen wie.“

1. vertragliche Zustimmung (\Rightarrow Rechtsstreit)
2. soll \Rightarrow verpflichtend
3. sollte \Rightarrow erwünscht
4. Mehrdeutigkeiten sollten verhindert werden
5. IEEE 830.1998
6. DIN 69905 Lastenheft/Pflichtenheft

80

Antwort

1. Zerlegung des Systems in Module
2. beschreiben der Interaktionen der Module
3. Struktur der Definitionen/ Aufrufe der Prozedur
4. keine abstrakten Datentypen/ Vererbung
5. Typen:
 - a) Architektonisches Design: bestimmen der Hauptelemente (Module) und Beziehungen
 - b) Oberflächendesign: wie entwirft man Wechselbeziehungen/ Kommunikation/ gegenseitige Abhängigkeiten (Verträge) von Modulen
 - c) Datendesign: Datenstrukturen
 - d) prozedurales Design: bestimmen einer Ablaufbeschreibung für die Elemente

79

Antwort

1. vom „Was“ zum „Wie“
2. bewältigen von Komplexität
3. Design für Veränderung (Vorwegnehmen der Veränderung auf allen Stufen, senken der Kosten der Veränderung)
4. Designprozess sollte nicht unter dem Tunnelblick leiden
5. Design sollte zurückverfolgbar zum Analysemodell sein
6. nicht das Rad neu erfinden
7. minimieren der intellektuellen Distanz zwischen der Software und dem Problem der realen Welt
8. Einheitlichkeit und Integration
9. Änderungen unterbringen
10. vorsichtig abbauen, wenn abweichende Daten/Ereignisse/... eintreffen
11. Design wird nicht programmiert
12. beurteilen der Qualität des Designs während es erstellt wird
13. bewerten des Designs

klassisches Design: Modul

Objektorientiertes Design (Infos)

Objektorientiertes Design (Aktivitäten)

Objektorientiertes Design (Methoden)

Objektorientiertes Design
(Generalisierung & Vererbung)

Design mit UML (Modelltypen)

Design mit UML (typisch benutzte
Modelle)

Schnittstellen und Verträge

# 82	Antwort	# 81	Antwort
<ol style="list-style-type: none"> modellieren des Systems als Sammlung von Klassen und Objekten Beschreibung von Objektschnittstellen Verkapselung, abstrakte Datentypen Wiederverwendbarkeit durch Vererbung Polymorphismus 		<ol style="list-style-type: none"> gut definierte Komponente eines Systems (bietet eine Anzahl von Diensten zu anderen Modulen, besteht aus Name, Oberfläche (Schnittstelle), Körper) beherrschen der Komplexität, erleichtert Dokumentation, ermöglicht Teamarbeit Beschreibung von Schnittstellen („Vertrag von Server/Client“) und Sprachen (CORBA IDL) Cohesion - Zusammenhalt (Bindungen innerhalb der Module, inwieweit ein Modul genau eine Funktion ausführt) Coupling - Kupplung (Bindungen zwischen den Modulen, inwieweit ein Modul mit anderen Modulen verbunden ist (direkter Zugriff, Datentransfer)) erwünscht: <i>high Cohesion, low Coupling</i> 	
# 84	Antwort	# 83	Antwort
<p>Booch Methode: Makroentwicklungsprozess (architektonische Planung, Partitionieren, Schichtung) und Mikroentwicklungsprozess (Regeln zur Implementation von Besonderheiten)</p> <p>Rumbough Methode: Systemdesign (Analysemodell repräsentiert das System, Layout von Komponenten, Aufteilen in Untersysteme, Ausführung unter Berücksichtigung der Umwelt) und Objekt Design (Design von Algorithmen/Datenstrukturen)</p> <p>Jacobson Methode: Zurückverfolgen zum Analysemodell ist möglich, Anpassung des Analysemodells an die Real-World-Umgebung, kategorisieren von primären Designobjekten (Schnittstelle, Einheit, Kontrolle), bestimmen von Kommunikation (organisieren in Untersysteme)</p> <p>Wirfs-Brock Methode: Analyse \Rightarrow Design, definieren von Protokollen durch Verträge zwischen den Objekten, Spezifikationen von Klassen und Untersystemen</p>		<ol style="list-style-type: none"> Aufteilung des Modells in Untersysteme identifizieren von Parallelisierung zuordnen der Untersysteme zu Prozessoraufgaben entwickeln des User-Interface-Designs wählen einer Strategie für das Datenmanagement identifizieren von globalen Ressourcen (und Kontrollmechanismen, Aufgabenmanagement) betrachten von Randbedingungen (und wie sie behandelt werden) <ol style="list-style-type: none"> Sonderfälle (Systemstart, Initialisieren, Herunterfahren, schwere Fehler und Ausnahmen, beschädigte Daten, Netzwerkfehler) Boundary Use Case (für alle Untersysteme und dauerhafte Objekte: Konfiguration, Starten/Herunterfahren, Fehlerbehandlung) für Komponentenfehlertyp (Netzwerkausfall, ...) wird entschieden, wie das System reagieren soll (anlegen eines Exceptional Use Case) bewerten und betrachten von „trade-offs“ (Kompromissen) Aufrechterhaltung (Objekte arbeiten eigenständig), Wiederverwendbarkeit, reduzieren von semantischen Lücken (real-world \leftrightarrow Software) 	
# 86	Antwort	# 85	Antwort
<ol style="list-style-type: none"> Designmodelle: zeigen Objekte, Objektklassen, Beziehungen statische Modelle: beschreibt statische Struktur, Objektklassen und Beziehungen dynamische Modelle: beschreibt dynamische Interaktionen zwischen Objekten 		<ol style="list-style-type: none"> Objekte sind Elemente von Klassen (definieren von Attributen und Operationen) Klassenhierarchie (Generalisierung) Generalisierung in UML (Vererbung in Objektorientierung) Abstraktion (klassifizieren von Entitäten) Vererbungsgraph (organisatorisches Wissen über die Domain / das System) <i>Cohesion</i>: wie sehr passt die Funktionalität einer Klasse zusammen <i>Coupling</i>: B ist Unterklasse von A \Rightarrow B erbt von A \Rightarrow Kupplung zwischen A und B, spiegelt die real-world-Struktur (keine Designentscheidung) 	
# 88	Antwort	# 87	Antwort
<ol style="list-style-type: none"> Schnittstellen müssen spezifiziert sein, manchmal können andere Objekte parallel entwickelt werden unterschiedliche Schnittstellen pro Objekt/Modul möglich benutzen von UML Klassendiagrammen Operationen A, B gleichzeitig, if ... A ... B ... and ... B ... A ... ist möglich (logische Parallelisierung, Ausführung auf verschiedener Hardware, A/B nicht Datenfluss-unabhängig) benutzen von vererbter Parallelisierung zur Verteilung des Systems benutzen eines Aktivitätendiagramms zur Modellierung von Arbeitsflüssen, wie Workflows, dicke Barren sind Synchronisationspunkte, „Swim-Lanes“ sind Abtrennungen, die verschiedene Systeme bezeichnen 		<ol style="list-style-type: none"> Untersystem-/Komponentendiagramme: strukturelle Verfeinerung von Objekten: Eine UML-Komponente repräsentiert einen modularen Teil eines Systems, das seinen Inhalt kapselt und dessen Erscheinungsformen ersetzbar ist innerhalb seiner Umwelt. Sequenzdiagramme: modellieren eine Abfolge von Interaktionen unter Objekten statechart-Diagramme: modelliert Zustandsverhalten von einem Objekt, zeigt wie das Objekte und Zustandsübergänge reagieren, ausgelöst von Service-Anfragen, Parallelität möglich (getrennt durch gestrichelte Linien) activity-Diagramm: modelliert gleichzeitiges Verhalten eines Objektes Deployment-Diagramm: repräsentiert die Zuteilung von verschiedenen UML Nodes, Hardware/Software Zuordnung (wie ein Komponentendiagramm in 3D) Use-Case-Diagramm: modelliert Interaktionen (inklusive Beschreibung) 	

Design-by-Contract (Eiffel-Sprache)

Subcontracts (Vererbung)

Data Dictionaries

Softwarearchitektur und Patterns

Softwarearchitektur und Patterns
(architektonische Stile)

Softwarearchitektur und Patterns
(architektonische Stile: pipe and filter)

Softwarearchitektur und Patterns
(architektonische Stile: call and return)

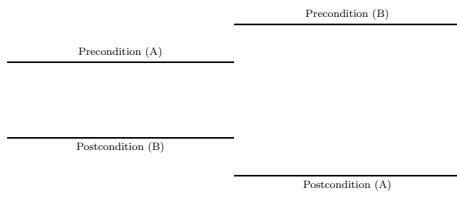
Softwarearchitektur und Patterns
(architektonische Stile: Layers in
hierarchischer Architektur)

90

Antwort

möglicherweise Subkonstruktor

1. beibehalten oder schwächen der Precondition (mehr kann reinkommen)
2. beibehalten oder stärken der Postcondition (weniger kann heraus)



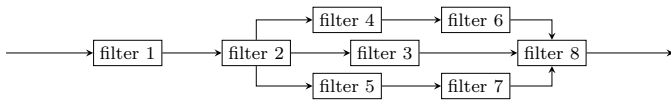
92

Antwort

Die Architektur eines Softwaresystems definiert dieses System in Bezug auf rechnerische Komponenten und Interaktionen zwischen diesen Komponenten. Es involviert eine Beschreibung von Elementen aus denen das System gebaut wird, Interaktionen zwischen ihren Elementen, Patterns, die ihre Zusammensetzung führen und Einschränkungen dieser Patterns.

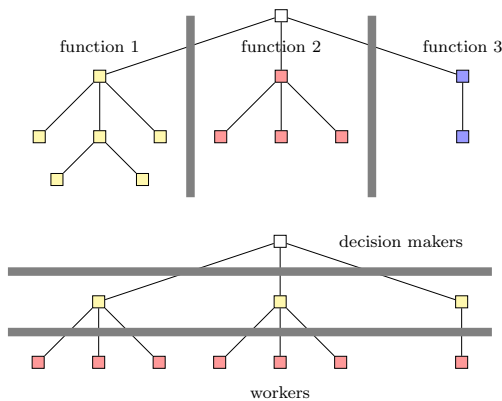
94

Antwort



96

Antwort



89

Antwort

1. Server muss die *Postcondition* gewährleisten, kann die *Precondition* annehmen
2. Client muss die *Precondition* gewährleisten, kann die *Postcondition* annehmen
3. UML Object Constraints Language (OCL):

HashTable
numElements:int
put(key:entry;Object)
get(key):Object
remove(key:Object)
containsKey(key:Object):boolean
size():int

put: precondition: !containsKey(key),
postcondition: get(key)==entry

get: precondition: containsKey(key),
postcondition: !containsKey(key))

remove: precondition: containsKey(key),
postcondition:
!containsKey(key)

91

Antwort

1. low-level Datenentscheidungen (spät im Designprozess)
2. Repräsentation von Datenstrukturen (nur für diejenigen bekannt mit direkten Dateninteraktionen (Information hiding))
3. Liste von allen Namen, Einheiten, Beziehungen, benutzten Attributen (Namenmanagement, Vermeidung von Duplikaten, Wissen über Speicherorganisation)

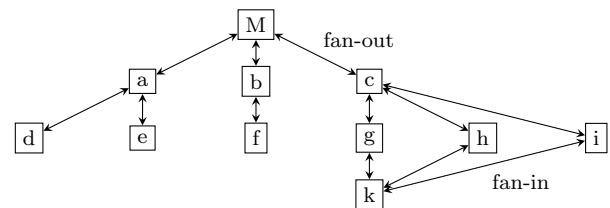
93

Antwort

1. pipe and filter
2. call and return (empfohlen: high fan-in, low fan-out)
3. Layers in hierarchischer Architektur (Layer: Gruppe von eng verwandten und sehr verbundenen Funktionalitäten)
4. blickdichtes Layering: vm kann nur vom unterliegenden Layer aufgerufen werden (Aufteilung von Bedenken, Wartbarkeit, Flexibilität)
5. transparentes Layering: vm kann von jedem anderen Layer aufgerufen werden (Laufzeiteffizienz (keine Parameter/Weitergabe von Nachrichten, Datenkonversation, Formatänderungen))
6. Client/Server (Kommunikation von Empfängern) - Client: Prozess, der auf Daten zugreifen/Ressourcen benutzen/Operationen ausführen möchte, Server: Prozess, der Daten und gemeinsam genutzte Ressourcen managed

95

Antwort



Was ist ein Design Pattern?

Design Patterns (Arten)

Design Patterns: Adapter - Wrapper

Design Patterns: Brücke

Design Patterns: Decorator

Design Patterns: Facade

Design Patterns: Observer

Design Patterns: Proxy/Stellvertreter

<div># 98</div> <div>Antwort</div> <div> <div>1. Adapter - Wrapper</div> <div>2. Brücke</div> <div>3. Decorator</div> <div>4. Facade</div> <div>5. Observer</div> <div>6. Proxy/Stellvertreter</div> <div>7. Singleton</div> </div>	<div># 97</div> <div>Antwort</div> <div> <div>1. Jedes Pattern beschreibt ein Problem, das wieder und wieder in unserer Umwelt auftritt und es beschreibt den Kern einer Lösung für dieses Problem, sodass man diese Lösung tausendfach benutzen kann, ohne es zweimal gleich zu benutzen.</div> <div>2. verbessert die Dokumentation, abstrakteres Level der Programmierung, verbesserte Kommunikation</div> </div>
<div># 100</div> <div>Antwort</div> <div> <div>1. Pattern und Klassifikationsname: Brücke</div> <div>2. Absicht: abkoppeln einer Abstraktion von der Implementation, sodass diese beiden variieren unabhängig</div> <div>3. Motivation: eine Brücke kann eine dauerhafte Verbindung zwischen Abstraktion und Implementierung verhindern.</div> </div>	<div># 99</div> <div>Antwort</div> <div> <div>1. Pattern und Klassifikationsname: Adapter Wrapper</div> <div>2. Absicht: Anbieten eines alternativen Interfaces für eine Klasse</div> <div>3. Motivation: fremde Klasse mit unpassendem Interface in passendes Interface verwandeln</div> </div>
<div># 102</div> <div>Antwort</div> <div> <div>1. Pattern und Klassifikationsname: Facade</div> <div>2. Absicht: baut eine Klasse als Fassade vor viele andere Klassen, ermöglicht Methoden, die wieder Methoden in hinteren Klassen aufrufen, vereinfacht Schnittstellen</div> <div>3. Motivation: einfache Klasse für Zugriff von Außen, voller Zugriff auf Methoden des Basissystems</div> <div>4. related Patterns: Wrapper, Mediator</div> </div>	<div># 101</div> <div>Antwort</div> <div> <div>1. Pattern und Klassifikationsname: Decorator</div> <div>2. Absicht: fügt individuellen Objekten Zuständigkeiten hinzu, aber nicht der Klasse</div> <div>3. Motivation: Bspw. TextArea einen Rahmen und eine Scrollbar hinzufügen</div> </div>
<div># 104</div> <div>Antwort</div> <div> <div>1. Pattern und Klassifikationsname: Proxy, Stellvertreter, Surrogat</div> <div>2. Absicht: kontrollierter Zugriff auf ein Objekt mithilfe eines vorgelagerten Stellvertreters</div> <div>3. Motivation: Proxy hält eine Referenz auf das reale Objekt, stellt gleiche Schnittstelle zur Verfügung</div> </div>	<div># 103</div> <div>Antwort</div> <div> <div>1. Pattern und Klassifikationsname: Observer/ publish-subscribe</div> <div>2. Absicht: abstrakte Kommunikationsmöglichkeit</div> <div>3. Motivation: Das beobachtete Objekt bietet Anmeldung für Observer an, informiert diese dann über interne Änderungen. Muss Struktur des Observers nicht kennen, meldet einfach jede Änderung weiter ⇒ Observer muss darauf reagieren können</div> </div>

Design Patterns: Singleton

Mapping der Designmodelle zum Code

Implementieren von
Schnittstellenverträgen

Mappen auf Relationale Datenbanken

System Design Dokument

Objekt Design Dokument

Testing und
Software-Qualitätssicherung

SQA: Beurteilungen

<div># 106</div> <div>Antwort</div> <div> <p>Optimierung: erfüllen der Performance-Anforderungen (reduzieren von mehrfachen/ redundanten Assoziationen)</p> <p>Realisierung von Assoziationen: mappen der Assoziationen zum Quellcode</p> <p>mapping of contracts to exceptions: erhöhen und behandeln von Ausnahmen (Exceptions) falls der Vertrag gebrochen</p> <p>mapping of class models to storage schema: auswählen von einer Strategie zur dauerhaften Speicherung (Datenbank, flat files), definieren eines relationalen Datenbankschemas</p> <p>Implementation der Sichtbarkeit von Attributen: private, protected, public</p> </div>	<div># 105</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> 1. Pattern und Klassifikationsname: Singleton 2. Absicht: Klasse mit genau einer Instanz, normalerweise globaler Zugriff 3. Motivation: Klasse hat komplette Zugriffskontrolle 4. Anwendbarkeit: Bspw. Druckerbuffer, Fliesystem, Window Manager </div>
<div># 108</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> 1. mappen der UML-Konstrukten auf Tabellen (nicht alles kann gemappt werden) 2. Klasse → Tabelle 3. Klassenattribut → Spalte in Tabelle 4. Klasseninstanz → Zeile in Tabelle 5. many-to-many Assoziationen → eigene Tabelle 6. one-to-many → Fremdschlüssel 7. kein direktes Mapping → etwas in SQL, ... machen </div>	<div># 107</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> 1. die meisten objektorientierten Sprachen haben keinen eingebauten Support für Verträge (ausgenommen Eiffel) 2. manche objektorientierten Sprachen haben ein Exception-Handling-Support eingebaut (throw-catch in Jave) 3. überprüfen jeder <i>Precondition</i> vor dem Beginnen einer Methode 4. überprüfen jeder <i>Postcondition</i> nach einer Methode 5. überprüfen der <i>Invarianten</i> vor und nach jeder Methode </div>
<div># 110</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> 1. Einführung (trade-offs, Interface-Dokumentation, Definitionen, Referenzen) 2. Packages (Dateiorganisation, ...) 3. Klasseninterfaces (öffentliche Schnittstellen (inkl. Operationen, Attribute) jeder Klasse, Abhängigkeiten unter den Klassen) </div>	<div># 109</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> 1. Einführung (Sinn des Systems, Designziele, Definitionen, Referenzen, Überblick) 2. Vorhandene Architektur 3. vorgeschlagene Softwarearchitektur (Funktionalität, Subsysteme, Datenspeicherung, Zugriffskontrolle, Boundary Conditions) </div>
<div># 112</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> 1. technische Treffen (walk-through, Inspektionen) 2. Ziele: SQA, Training 3. Nicht-Ziele: Fortschrittsbewertung, Budget, Fehlerbehebung, Vergeltungsmaßnahmen (reprisals), politische Intrigen 4. bewerten des Leiters, auf den Inhaber lautende Standards (SQA), Wartung des Orakels (Anwalt des Teufels), Schreiber, Benutzervertreter, Produzent, Kritiker 5. auswerten vor der Bewertung 6. bewerten des Produktes, nicht des Herstellers (Fragen stellen anstatt anzuklagen, vermeiden eines kritisierenden Stils (technische Korrektheit) 7. Fragen aufwerfen, nicht lösen </div>	<div># 111</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> 1. 3 Fault pro 1000 Zeilen Code 2. Techniken: <ol style="list-style-type: none"> a) Verifikations- und Validationstechniken (Codeanalyse, Bewertung, Testing, formale Verifikation), je früher desto besser (Kosten) <ol style="list-style-type: none"> i. Verifikation: beweisen, dass ein Produkt seinen Spezifikationen entspricht (formale Verifikation (Model Checking, formale Codeverifikation (Beweise)), Korrektheitsbeweise) ii. Validation (Bestätigung): experimentieren mit dem System, um zu zeigen, dass es die Anforderungen erfüllt (Simulation, Testing, Model Checking) b) Qualitätsbeurteilung des Codes (Softwaremetriken) </div>

SQA: Code walk-through

SQA: Codeinspektionen

Testabdeckung (Coverage)

Testingtypen (Diagramm)

Testingtypen: Unit/Module Testing

Testingtypen: Integrationstesting

Testingtypen: Systemtesting

Testingtypen: Akzeptanztesting

114

Antwort

1. ähnlich wie walk-through, anderes Ziel: suchen nach häufig gemachten Fehlern, keine Simulation des Verhalten des Computers
2. nicht initialisierte Variablen, Sprünge zu/in Schleifen, nicht kompatible Zuordnungen, nicht-terminierende Schleifen, Arraygrenzen, Speicherzuweisung/-freigabe, Parametermismatch
3. Analysetool kann hilfreich sein

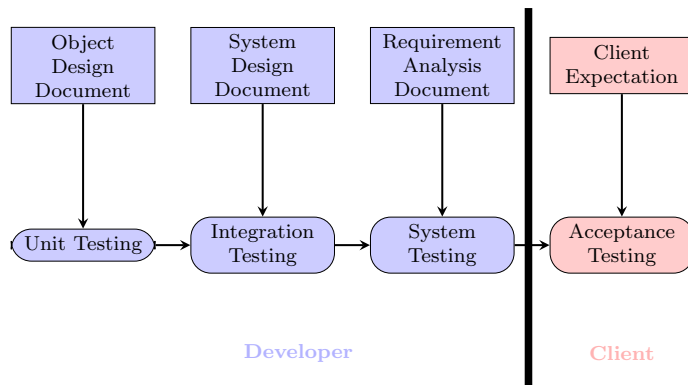
113

Antwort

1. „Computer spielen“ \Rightarrow „interpretieren des Codes“
2. 3-5 Teilnehmer
3. finden von Fehlern, nicht fixen
4. Moderator, Sekretär, Prüfer, Designer (erklären des eigenen Designs)

116

Antwort



118

Antwort

1. wird während des Zusammenbaus des ganzen Systems ausgeführt, testet die Module zusammen, und ob das System den Anforderungen entspricht (ausgeführt mit Integratoren, getestet von Spezialisten)
2. Driver: Komponente, die testedUnit aufruft, Testfälle kontrolliert
3. Stub: Komponente, von der die testedUnit abhängt („Fake“)
4. Testmethoden:
 - a) **big bang:** zusammenfügen und sehen, ob es funktioniert
 - b) **top-down:** testen die oberste Schicht zuerst, dann abwärts (Stubs sind notwendig, Test kann durch die Funktionalität des Systems entworfen werden)
 - c) **bottom-up:** Untersysteme in der untersten Schicht werden zu erst getestet (Drivers sind notwendig (Testen des UI (wichtigster Teil) zum Schluss)
 - d) **sandwich:** Kombination von bottom-up und top-down, konvergiert auf der Zielebene
 - e) **modified sandwich:** Kombi aller anderen Testmethoden

117

Antwort

1. überprüfen, ob ein Module den Designspezifikationen entspricht (oft während der Programmierung)
2. getestet wird: Methoden im Objekt, Klassen mit mehreren Attributen/Methoden, Komponenten, die aus mehreren Klassen und internen Schnittstellen bestehen
3. System wird nicht als Ganzes getestet
4. gesamte Abdeckung: Testen aller Operationen mit allen möglichen Kombinationen von Parametern, sowie einstellen und testen aller möglichen Kombinationen von Objektattributwerten, sowie testen allen möglichen Zustände (Anzahl steigt schnell an)
5. zwei Arten von Tests:
 - a) normal: Einheit tut, was sie tun soll
 - b) Robustheitstest: wenn es einer „feindlichen Umgebung“ ausgesetzt wird, ist die Einheit robust

120

Antwort

1. testet, wie der Kunde benutzt, wenn sie geliefert wurde (kann behoben werden in der Anforderungsspezifikation oder im Vertrag)
2. demonstriert, dass das System bereit für den betrieblichen Einsatz ist
3. Auswahl von Tests von Kunden, durchgeführt vom Kunden
4. kann Teil des Vertrages sein
5. Alpha Tests: Tests in Entwicklungsumgebung
Beta Tests: Tests in Kundenumgebung

119

Antwort

testen des System als Ganzes

Testingtypen: Regressionstesting

Testingtypen: Stresstesting

Testing: Dijkstra

formelles Testing

Selektives Testing - wie viele Pfade
muss man testen? (Coverage)

Basic Path Testing

Testen auf Grundlage von Verträgen
(mögliche Testauswahlkriterien)

Testplan

<div># 122</div> <div>Antwort</div> <div>testen des Systems unter extremen Bedingungen</div>	<div># 121</div> <div>Antwort</div> <div>ausgeführt während der Wartung/ wenn sich eine Komponente verändert hat (sicherstellen, dass das was vorher funktioniert hat auch weiterhin funktioniert)</div>
<div># 124</div> <div>Antwort</div> <div> <p>D: Eingabe Domain, R: beliebige Menge, Ergebnis</p> <p>P: Teilfunktion, beschreibt das Verhalten des Programms bei Input/Output; $P : D \rightarrow R$</p> <p>OR: Ausgabewert der Anforderungen (aus dem SRS)</p> <p>testing: bestimmen ob für ein gegebenes $d \in D, P(d)$ die Anforderungen erfüllt sind (angegeben in OR), falls ja \Rightarrow Erfolg, ansonsten <i>fail</i></p> <p>$d \in D$: Testfall (Test Case)</p> <p>$T \subseteq D$: Testmenge/-folge</p> <p>T: ist erfolgreich, gdw. alle $d \in T$ erfolgreich sind</p> <p>$C \subseteq 2_f^D$: Auswahlkriterium, 2_f^D ist Menge aller endl. Teilmengen von D</p> <p>T: erfüllt C gdw. $T \in C$</p> <p>für alle i, k: nehmen an, dass T_i, T_k erfüllt C, C ist konsistent, falls (T_i ist erfolgreich, gdw. T_k ist erfolgreich)</p> <p>C: ist komplett, falls P nicht korrekt, so gibt es T(nicht erfolgreich) das C erfüllt</p> <p>falls C komplett und konsistent: jeder Test T erfüllt C entscheidet P's Korrektheit</p> </div>	<div># 123</div> <div>Antwort</div> <div> <p>„Programmtesting kan benutzt werden um zu zeigen, dass ein Fehler vorliegt, aber niemals um zu zeigen, dass es keinen Fehler gibt.“</p> <p>Man muss immer genau schauen, welchen Teil des Systems man mit seinem Testing erreicht (Test-Coverage).</p> </div>
<div># 126</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> beurteilen wie viele Tests benötigt sind, um Aussagen und Path Coverage zu erreichen (Programm ist ein einzelner Eintrag, einfacher Ausgang, Kontrollflussgraph verfügbar, alle Entscheidungen sind Binär) berechnen der <i>cyclomatic complexity</i> $V(G)$: $V(G) = \# \text{ einfacher Entscheidungen} + 1 = \# \text{ abgeschlossener Flächen} + 1$ Schleifentesten (n ist Anzahl erlaubter Durchläufe) <ol style="list-style-type: none"> überspringen der Schleife erster Durchlauf zweiter Durchlauf m Durchläufe ($m < n$) $(n - 1), n, (n + 1)$ Durchläufe </div>	<div># 125</div> <div>Antwort</div> <div> <p>Statement Coverage: Jedes Statement wird mindestens einmal besucht</p> <p>Edge Coverage: Jede Kante wird genau einmal traversiert</p> <p>Condition Coverage: Edge Coverage ist erfüllt und alle möglichen Werte für die Bestandteile der Aussage innerhalb der <i>Conditions</i> wird mindestens einmal ausgeführt</p> <p>Multiple Condition Coverage: Jede boolesche Kombination der Bedingungen in jeder <i>Condition</i> muss mindestens einmal ausgeführt werden</p> <p>Path Coverage: Alle Pfade von einem Startknoten zu einem Endknoten müssen mindestens einmal durchlaufen werden</p> </div>
<div># 128</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> Systemübersicht Features zum Testen und Nicht-Testen (funktionale Aspekte, Beschreibung der zu testenden Features) Kriterien für „Fail“ oder „Pass“ Herangehensweise an Testprozess, Begründung zur Wahl der Testmethode Testfälle <ol style="list-style-type: none"> Testfallbezeichner Testitems Eingabe- und Ausgabespezifikation Umgebungsanforderungen (Hardware, Software) besondere prozedurale Anforderungen (Zeitlimits, ...) Abhängigkeiten zwischen Testfällen </div>	<div># 127</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> Input entspricht der Precondition Inputs mit verletzter Precondition Inputs, bei dem das Key-Element den richtigen Typ hat Inputs, bei dem das Key-Element den falschen Typ hat </div>

Äquivalenzklassenbildung

Softwaremetriken: McCabes’s
Cyclometric Complexity

Softwaremetriken: Software
Produktmetriken

Softwaremetriken: Objektorientierte
Softwaremetrik

Model Checking als
Verifikationstechnologie

State-Based-Modeling: State

State-Based-Modeling:
Zustandsübergänge in diskreten
Systemen

State-Based-Modeling: Interne
Prozesssynchronisation

# 130	Antwort
<ol style="list-style-type: none"> bestimmen eines Kontrollflussdiagramms zyklomatische Komplexität $C = e - n + 2p$ (e = Kanten, n = Knoten, p = Anzahl von verbundenen Komponenten (1 für zusammenhängend)) C bestimmt die Anzahl von Pfaden keine „GoTo's“, ein einziger Eingang & Ausgang $\Rightarrow C - 1 = \text{Anzahl von Entscheidungsknoten}$ $3 \leq C \leq 7 \Rightarrow \text{gute Werte, } C = 10 \text{ ist Maximum}$ 	

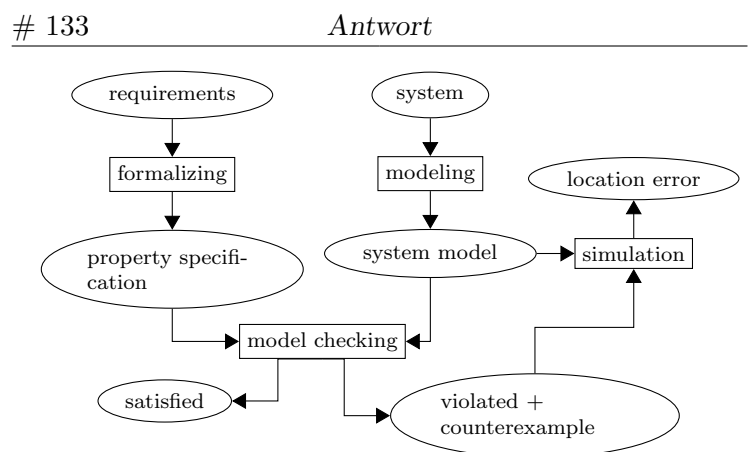
# 132	Antwort
<p>Tiefe des Vererbungsbaumes: tiefere Vererbungsbaum bedeutet komplexeres Design</p> <p>Methode fan-in/fan-out: unterscheiden von internen (gut) und externen (schlecht) Methoden (siehe Produktmetriken)</p> <p>gewichtete Methoden pro Klasse: Anzahl von Methoden in einer Klasse werden nach ihrer Komplexität gewichtet (hoher Wert ist schlecht)</p> <p>Anzahl von übergeordneten Operationen: hoher Wert zeigt, dass die Oberklasse nicht gut modelliert ist für die Unterkasse</p>	

# 134	Antwort
<ol style="list-style-type: none"> Charakteristik der hervorstechenden Eigenschaften eines Systems an einem bestimmten Beobachtungspunkt Konvention: ein gegebener Zustand kann solange beobachtet werden, wie die Merkmale, die von Interesse sind, unverändert bleiben Merkmale von Interesse in diskreten simultanen Softwaresystemen: <ol style="list-style-type: none"> Kontrollpunkt („Programmzähler“) aller Prozesse aktuelle Werte einer lokalen und globalen Variablen am Beobachtungspunkt Inhalt aller Kommunikationskanäle im System (Nachrichten, die gesendet, aber noch nicht empfangen wurden) Zustände werden oft als Zustandsvektor repräsentiert (bitweise Repräsentation eines oben genannten Merkmals) 	

# 136	Antwort
<ol style="list-style-type: none"> beide Telefone arbeiten völlig unabhängig von einander um das Phänomen darzustellen, dass die beiden sich gegenseitig anrufen wollen, müssen wir eine interne Prozesssynchronisation hinzufügen synchrone Nachrichtenweitergabe mit den folgenden Nachrichten: <p>!digits Partner anrufen</p> <p>?digits Anruf erhalten</p> <p>!onhook beenden eines Anrufes durch auflegen</p> <p>?onhook Benachrichtigung über die Beendigung des Anrufes vom Partner</p> 	

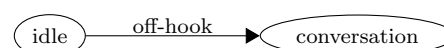
# 129	Antwort
<p>Aufsplitten der Inputdomain in zwei äquivalente Klassen, sodass jede Eingabeklasse so viele Programmfehler wie möglich aufzeigt. Zur Bestimmung der Äquivalenzklassen: schauen auf die Eingabebedingungen des Systems.</p> <p>Testen eines Wertes für jede Äquivalenzklasse.</p>	

# 131	Antwort
<p>fan-in/fan-out: Anzahl an Funktionen oder Methoden, die alle anderen Funktionen oder Methoden aufrufen kann (X). Fan-Out ist die Anzahl der Funktionen, die von X aufgerufen werden. Hohes fan-in zeigt enges <i>Coupling</i> (Veränderungen an X sind nicht gut), hohes fan-out deutet auf hohe Komplexität</p> <p>Länge des Quellcodes: je „länger“ das Programm, desto mehr Errors</p> <p>zyklomatische Komplexität</p> <p>Größe von Indikatoren: größere Identifikatoren sind einfacher zu verstehen</p> <p>Tiefe von bedingter Verschachtelung: tiefe Verschachtelung ist schwer zu verstehen</p> <p>Nebel-Index (fog index): durchschnittliche Länge von Wörtern und Sätzen in Dokumenten, größerer „Nebel“ ist schwerer zu verstehen</p>	



# 135	Antwort
<ol style="list-style-type: none"> unmittelbare Veränderungen des beobachteten Merkmals des Systems repräsentiert einen Berechnungsschritt Sequenzen von Zustandsübergängen charakterisieren Berechnungen des Systems 	

Beispiel:



Model Checking: System Deadlock

Model Checking: Deadlockerfassung mit
SPIN

Model Checking:
Zustandsraumexploration (Erkundung)
- Schritte

Model Checking: DFS vs. BFS

GSS Construction: Beispiel

Model Checking: Konstruktion
(Berechnung) eines globalen
Zustandsraums (Regeln)

Model Checking: Counterexample

Model Checking: SPIN

138

Antwort

algorithmischer Ansatz für das Fehlen eines Deadlocks im Model Checking:

1. betrachten des globalen Zustandsraum des Systems, erhalten durch das Multiplizieren der Zustandsräume der beiden Prozesse (Kartesisches Produkt der beiden Zustandsräume)
2. systematische Suche nach dem globalen Zustandsraum für Zustände, die keinen Nachfolger haben

140

Antwort

1. Vorteile **BFS**: findet einen verletzenden Zustand immer auf dem kürzesten Weg
2. Vorteile **DFS**: Stack enthält automatisch das *Counterexample* (BFS muss den Vorgänger als zusätzliche Information speichern), Speichereffizient - muss nur einen Teil der Zustände speichern (BFS muss alle Zustände speichern)
3. Schlussfolgerungen: **DFS** ist in unserem Fall besser, **BFS** kann nur für mäßig große Modelle verwendet werden, realistische nur mit **DFS**

142

Antwort

1. Übergänge in lokalen (individuellen) Zustandsmaschinen (Statechart)
2. Zustände im globalen Zustandsraum sind Paare der Form (s_1, s_2) , wobei s_1 ein Zustand aus State Machine 1 und s_2 ein Zustand aus State Machine 2 ist

144

Antwort

1. Syntax Check
2. Slicing: führt eine Datenflussanalyse in Bezug auf die Eigenschaft durch, bestimmt irrelevante Teile des Modells
3. Simulation: zufällig, interaktive oder pfadgeführte Simulation, wichtigste Debugginghilfe
4. Verification: Model Checker, führt eine Überprüfung der Sicherheit und Lebendigkeit durch
5. LTL Property Manager: hilft temporale Logikformeln zu bearbeiten und zu pflegen
6. FSM View

137

Antwort

1. höchst unerwünschtes Verhalten von gleichzeitig ausgeführtem Code
2. zeitgleich ausgeführte Prozesse warten aufeinander in einer kreisförmigen Wartezeit ohne „pre-emption“ (Vorbelegung)
3. Lösung: Prozessen erlauben die „Entstehung“ versuchen abubrechen, indem der Hörer aufgelegt wird

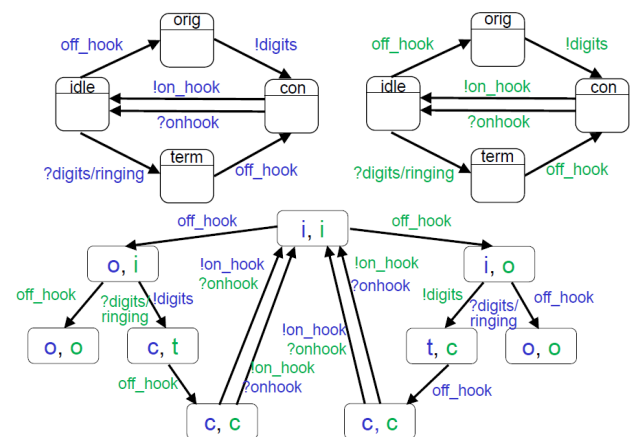
139

Antwort

1. Konstruktion eines *Global State Space (GSS)* (Regeln)
2. algorithmische Exploration des GSS (Suchen nach globalen Systemzuständen, die die Eigenschaft verletzen)
3. Durchsuchen des GSS:
 - a) systematische Suche, um eine Eigenschaftsverletzung der globalen Systemzustände zu finden
 - b) Strategien: DFS (benutzt Hash, Stack (benutzt um ein Counterexample zu erstellen))/BFS
 - c) Eigenschaften: *complete* (alle erreichbaren Zustände wurden besucht), *sound* (falls ein verletzender Zustand gefunden ist, ist garantiert, dass es ein möglicher (erreichbarer) Fehler des Systems ist), *if state space is finite* (Garantie, dass ein Fehler, falls vorhanden, in endlicher Zeit gefunden wird; falls kein Fehler gefunden wurde, nachdem die Suche terminiert: Korrektheitsbeweis)
4. Deadlocks sind die Zustände, die im GSS nicht mehr verlassen werden können (gerichteter Graph)

141

Antwort



143

Antwort

1. falls eine Eigenschaftsverletzung gefunden wurde, zeigt SPIN einen Ausführungspfad vom Ausgangszustand bis zum verletzenden Zustand (hilfreich beim Debugging)

Model Checking: Never Claim

Model Checking:
Zustandsraumexplosion (Problem)

Model Checking:
Zustandsraumexplosion (Techniken)

Korrektheitsbeweis: Regeln

Model Checking: linear temporal logic

Agile Methiden

SEI Capability Maturity Model (CMM):
Bild

SEI Capability Maturity Model (CMM):
Beschreibung

<div># 146</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> annehmen von n lokalen gleichzeitigen Prozessen (Proctypes) annehmen von K als Obergrenze für die Anzahl an Zuständen in jedem Prozess Worst Case: K^n Konsequenzen: Speicheranforderungen steigen exponentiell in n, Suchanstrengungen wachsen exponentiell in n Suche selbst ist Worst-Case linear </div>	<div># 145</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> nur eine Instanz pro Promelamodell synchron mit dem Rest des Modells ausgeführt kann einen Schritt ausführen, wenn Bedinungslabel des Überganges erfüllt is im aktuellen Zustand des Promelamodells nichtdeterministischer Automat Endzustand des <i>Never Claim</i> zeigt immer eine Eigenschaftsverletzung an stotternde Semantik: Endzustand wird für immer wiederholt </div>
<div># 148</div> <div>Antwort</div> <div> <p>Beweisstrategien für verschiedene Kontrollflusskonstrukte:</p> <ol style="list-style-type: none"> sequentielle Zusammensetzung: generelle Form: S1;S2 Beweisregel: $\frac{\{F_1\}S_1\{F_2\}, \{F_2\}S_2\{F_3\}}{\{F_1\}S_1; S_2\{F_3\}}$ bedingte Aussagen (if ... then ... else) Beweisregel: $\frac{\{P \cap C\}S_1\{Q\}, \{P \cap \neg C\}S_2\{Q\}}{\{P\} \text{ if } C \text{ then } S_1 \text{ else } S_2; \{Q\}}$ </div>	<div># 147</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> On-The-Fly Search: Effizienzproblem: jeder Knoten wird zweimal besucht (Generierung der Zustände, Suche), benötigt Speicher des ganzen Zustandsraums, Gefahr: löschen von Knoten die wieder besucht werden würden; Bsp: SPIN Partial Order Reduction: reduzieren der Anzahl des erforschten Zustände und Übergänge durch Nutzung von Redundanz im Zustandsraum, Anforderungen zur Reduzierung: Eigenschaft gilt im reduzierten Zustandsraum gdw. gilt im ganzen Zustandsraum Bit-State Hashing: kein Wiederbesuchen von Zuständen \Rightarrow exponentiell \rightarrow linear, SPIN benutzt Hashfunktionen basierend auf Checksummenpolynomen (Jenkin's Hash), speichern eines Bits (0,1), Probleme bei Duplikaten </div>
<div># 150</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> Extreme Programming Pair Programming (XP-Art) Probleme: kann schwierig sein, Interessen zu behalten, der Kunden, die im Prozess involviert sind, Verträge können ein Problem sein (wie mit anderen Ansätzen zur Durchführung) </div>	<div># 149</div> <div>Antwort</div> <div> <ol style="list-style-type: none"> \Box A: always A (immer A) \Diamond A: eventually A (möglicherweise A) $A \mathcal{U} B$: A until B (A solange bis B mindestens einmal gilt) </div>
<div># 152</div> <div>Antwort</div> <div> <p>Initial Ad-hoc, chaotisch, heroische Programmierer</p> <p>Repeatable einige Prozessdisziplin und Tracking</p> <p>Defined Prozess ist dokumentiert und standardisiert</p> <p>Managed Qualität quantitativ gesteuert/bewertet</p> <p>Optimizing kontinuierliche Prozessverbesserung</p> </div>	<div># 151</div> <div>Antwort</div> <div> <pre> graph TD 1[1] --- 2[2] 2 --- 3[3] 3 --- 4[4] 4 --- 5[5] style 1 fill:none,stroke:none style 2 fill:none,stroke:none style 3 fill:none,stroke:none style 4 fill:none,stroke:none style 5 fill:none,stroke:none </pre> </div>