Handzettel Klausur

Für die Klausur waren bei uns zwei A4 Blätter vorder- und rückseitig zugelassen. Die vier (bzw. eher 3,5) Seiten sind nachfolgend abgebildet.

Keine Garantie für Lesbarkeit oder Verständlichkeit was das Layout angeht. Am Anfang dachten wir es muss Platz gespart werden.
Habt Nachsicht mit uns, diese Lernzettel wurden bis spät in die Nacht angefertigt.
Gez. Martin am 12.10.2022 um 01:11 Uhr.
Um 8:30 Uhr beginnt die Klausur. Wird schon schief gehen :)

Ich bitte um dein Verständnis :)

Seite 1

Software Engineering - 12.10.2022

Martin Tienken

Software: Immateriell, keine natürliche Lokalität, keine stetige Funktion, hohe Komplexität, autonome Funktion, fehlende strukturelle Abgrenzung, hoher Bezug zur Realität (Anwendungsbereich)

Eigenschaften von Software: 1. An Software ist nichts natürlich, 2. Software wird nur entwickelt (keine Fertigungskosten) 3. Kopie & Original sind völlig gleich, 4. Software verschleißt nicht, 5. Fehler entstehen nicht durch Abnutzung, 6. Wiederverwendung ist extrem lukrativ

Prinzipien des Software-Engineerings

1. Abstraktion
- Lösung allgemeiner Probleme zuerst, dann konkreter werden
- Hierachie → verwandte Elemente dürfen Eigenschaften und Implementierung teilen
- Muster → Ausnutzen von Mustern
- Wiederverwendung von Komponenten
- Entwicklungskosten ↗ Verständlichkeit ↘

2. Modellierung
- Strategien → BottomUp, Top-Down
- Teile & Hersche → Wiederverwendung von Implementierungen = hohe Produktivität
- High Module Cohesion → enge Verbindung der Elemente innerhalb eines Moduls
- Low inter-modular coupling → so wenig Verbindungen zwischen seperaten Modulen wie möglich

3. Kapselung
- Verbergen und Schützen von inneren Details und Implementierung
- Kein unberechtigter Zugriff
- Nachteil: Erschwert die Testbarkeit von Modulen & die Fehlersuche

4. Hierachie und Zerlegung
- Zerlegung des Problems in Teilprobleme
- Bezug zu anderen Konzepten: Abstraktion, Modularisierung, Trennung von Zuständigkeiten

5. Trennung von Zuständigkeiten
- Aufteilung eines Problems in unabhängige Teilaufgaben
- Verteilung der Arbeitslast
- Aufteilung nach Aspekten → (Kontroll- & Datenfluss, fachliche & technische Aspekte, einzelne Produkteigenschaften, externe & interne Qualitätskriterien)
- Model-View-Controller

6. Einheitlichkeit
- Software besser verständlich machen
- Bessere Wartbarkeit
- Entwurfsentscheidungen leicht verstehen

Durch die Anwendung der Prinzipien wird Software besser wart- & anpassbar ✓

Software Engineering

Ziele: - Wirtschaftliche Art Software zu bekommen
- Methoden, Sprachen & Werkzeuge sind durch Konzepte verbunden

Aufgaben: ⊖ Analyse, ⊖ Spezifikation der Anforderungen, ⊖ Architektur-, System-, Softwareentwurf, ⊖ Codierung und Modultest, ⊖ Integration, Abnahme, Test, ⊖ Betrieb & Wartung, ⊖ Auslauf & Ersetzung

Ablauf: 1. Analyse und Modellierung 2. Objektorientierte Analyse 3. System- & Softwareentwurf 4. Objektentwurf 5. Codierung 6. Testen

Prozessmodelle

Agile Softwareentwicklung

Individuen und Interaktionen	über	Prozesse und Wartung
Funktionierende Software	über	Dokumentation
Zusammenarbeit mit Kunden	über	Vertragsvereinbarungen
Reagieren auf Veränderungen	über	Folgen eines Plans

- \+ Kundenintegration,
- \+ Schnelle Lieferung,
- \+ Anforderungsänderungen können immer auftreten
- \+ Tägliche Meetings mit Experte & Entwickler
- − Vertrauen in Mitarbeiter
- − keine fixen Zeiten
- − fehlende Motivation

! Agile A-E ist = Scrum !

SCRUM

- Product Owner: Identifizieren der Anforderungen und Priorisieren
- Scrum Master: Verwalten und sichtbar machen des Projektprozesses
- Team: Entwickelt das Produkt
- Stakeholder: Agieren als Beobachter und Berater

Sequentielles Wasserfallmodell

Anforderungen → Analyse → Entwurf → Implementierung → Test → Einsatz / Wartung

- \+ Auswertung der Ergebnisse nach jeder Phase möglich
- \+ Gutes Erklärmodell für einzelne Tätigkeiten des SE
- − Keine Beteiligung des Kunden - Systemtests oft verkürzt
- − Ausführbares Produkt erst am Ende

Iteratives (iteratives) Modell:

Anforderungen → Analyse → Entwurf → Implementierung → Test → Einsatz & Wartung

- \+ Rückschritte sind möglich
- − kein Parallelisieren der Teilaufgaben
- − Disziplinierte Durchführung erforderlich

Spiralmodell

1. Analyse
2. Risiko-Analyse
3. Entwickeln & Testen
4. Planung der nächsten Phase

- \+ Reduzierung des Gesamtrisikos
- \+ Gemeinsame Planung
- − kein einheitliches Modell
- − Anzahl der Spiralen die benötigt werden unklar

Anforderungsanalyse

Mithilfe der Anforderungsanalyse werden die Bedarfe des Nutzers ermittelt. Fachliche Anforderungen sind Qualitätsanforderungen wie Einfachheit, Verlässlichkeit, Performance, Benutzbarkeit oder Effizienz

Seite 2

Modelle in der Softwareentwicklung

Repräsentationsarten von Software - Spezifikation - Diagramme & Quellcode - Kennzahlen - Prospekte

Deskriptives Modell: Ein Modell, das ein Objekt beschreibt. Zuerst gibt es das Original, dann das Modell Bsp. Foto

Präskriptives Modell: Dient als Vorlage für ein Objekt. Zuerst das Modell dann das Original Bsp Bauplan

Prognostisches Modell: Deskriptive Modelle, beschreiben einen zukünftigen Zustand. Bsp Wettervorhersage

Lastenheft: Dient der ersten Anforderungssammlung **Pflichtenheft:** Liefert Anforderungsspezifikation

Inhaltliche Eigenschaften: ⊖ Vorstellung des Kunden wiedergeben, ⊖ Vollständig in der Spezifikation enthalten ⊖ keine Widersprüche der einzelnen Anforderungen, ⊖ Neutral (nicht mehr als gefordert), ⊖ Nachvollziehbar dokumentiert, ⊖ Das realisierte System kann auf die Anforderungen geprüft werden.

Strukturelle Eigenschaften: ⊖ für alle Interessenten verständlich, ⊖ Präzise (keine Interpretationsmöglichkeit) ⊖ leicht verwaltbar (einfacher Zugriff und Speicherung der Spezifikation) ! Merkmale konkurrieren!

Mögliche Struktur einer Anforderungsanalyse: 1. Einleitung 2. Zielsetzung 3. Ausgangssituation 4. Funktionale Beschreibung des Systems 5. Qualitätsanforderung 6. Weitere Systemanforderungen und Rahmenbedingungen

Bestandteile einer Anforderung: - Identifikator: Eindeutige Identifikation der Anforderung, - Beschreibung: Anforderung wird beschrieben in 1-3 Sätzen, - Problembeschreibung: Problem, welches die Anforderung verursacht, - Quelle: Identifiziert die anfordernde Person oder ein Dokument aus dem sich die Anforderung ergibt (bsp. Rechtsvorschrift)

Nichtfunktionale Anforderungen: Benutzerfreundlichkeit, Zuverlässigkeit, Leistung, Unterstützung, Implementierung, Schnittstelle, Betrieb, Auslieferung, Nutzungsbedingungen, ... (wie ist die Software beschaffen)

Funktionale Anforderungen: Was soll das Programm überhaupt können -> welche Funktionen sind gewünscht.

!!! funktionale Anforderungen sind zwar wichtig, die nicht funktionalen Anforderungen sollten auch beachtet werden !!!

Bsp: Programm funktioniert, braucht aber sehr lange (Ticketautomat)

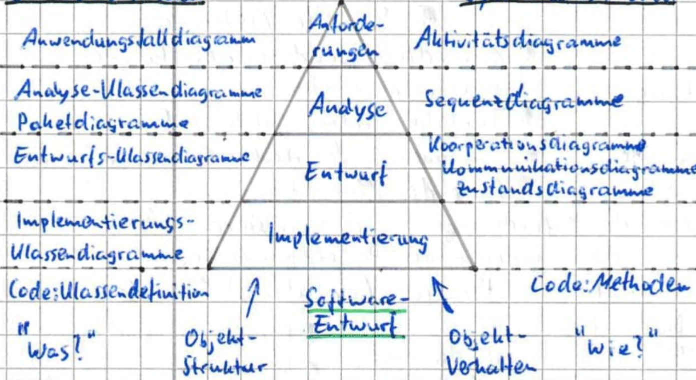
Quality Attribute Utility Tree: - wesentliche Qualitätsanforderungen aus Geschäftszielen ableiten, - Verfeinerung der Qualitätsanforderungen - konkret messbare Szenarien, - Bewerten jedes Szenarios nach Wichtigkeit und Risiko

Bsp. Utility → Performance → Data Latency → (M, L) Szenario 1 / (H, M) Szenario 2 (AG, AN)

Wichtigkeit 1 Parameter vom Auftraggeber
Risiko 2 Parameter vom Auftragnehmer
Skala: H -> Hoch, M -> Mittel, L -> Niedrig

Unified Modelling Language (UML)

UML ist eine grafische Modellierungssprache zur Spezifikation, Dokumentation und Visualisierung von Software-Teilen und anderen Sprachen



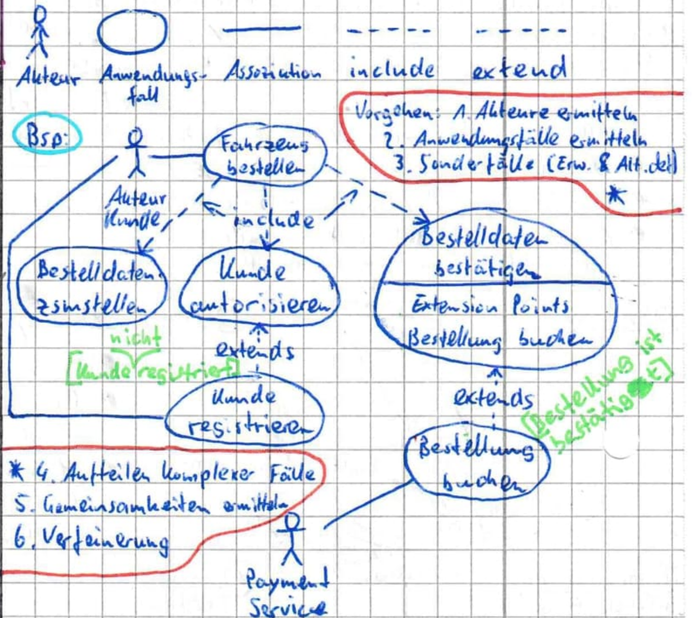
Diagramme

UML-Anwendungsfalldiagramm: Use-Case-Diagramme beschreiben Interaktionen von Akteuren mit Systemen über Anwendungsfälle.

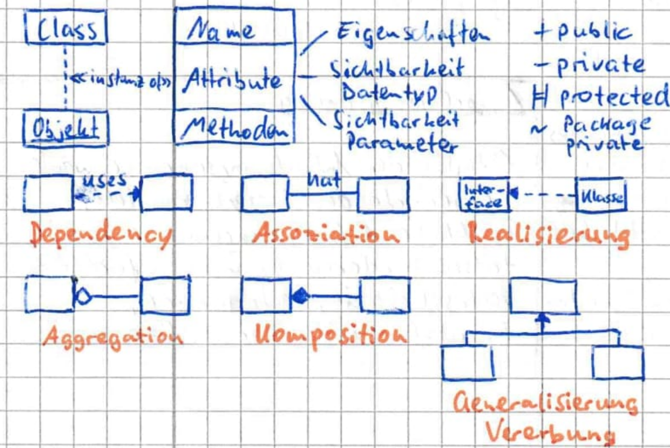
include -> Komplexe Anwendungsfälle aufteilen
extend -> Bedingte Erweiterung
↳ bei extend können (komplexe) Extension Points und Bedingungen angegeben werden.

Schablone/Dokumentationsvorlage

Dokum: Name: Registrieren, Ziel: Fahrzeug bestellen, Vorbedingung: Portal offen, Nachbedingung: Daten erhoben, Leistung steht zur Verfügung, Akteure: Kunde



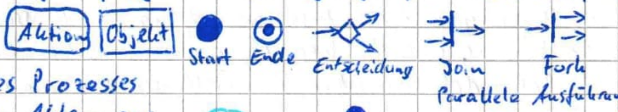
UML-Klassendiagramme: modellieren statische, strukturelle Beziehungen zwischen den Komponenten eines Systems



Seite 3

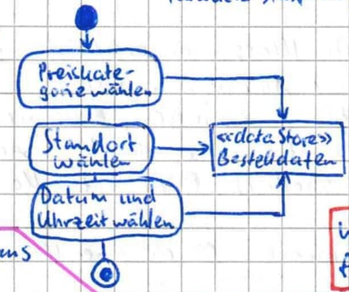
UML-Aktivitätsdiagramme: bieten eine Detaillierte Sicht auf Anwendungsfälle, ⊖ Detaillierte Sicht auf Ablauf innerhalb von Klassen, ⊖ Fokus auf Abfolge und Bedingungen, ⊖ Geschäftsprozess-modellierung mit der UML

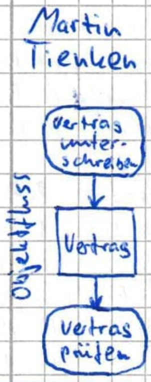
Martin Tienken



Wie erstelle ich Aktivitätsdiagramme:
1. Wer sind die wichtigen Akteure des Prozesses
2. Welche relevanten Aktionen führt der Akteur aus
3. Welche temporalen/kausalen Zusammenhänge bestehen zwischen den Aktionen
4. Welche Objekte werden benötigt bzw durch eine Aktion erzeugt oder verändert.
5. Welche asynchronen Signale werden ausgetauscht und wie hängen Kontrollflüsse zwischen den Aktionen davon ab?

Bsp.





Wenn der Anwendungsfall sehr komplex ist.

Entity-Boundary-Control-Muster: Mit dem EBC-Muster können aus Use-Case-Diagrammen Klassendiagramme erstellt werden.

○ Entity: Persistentes Wissen, Daten, aus dem Domänenmodell hergeleitet → für Datenobjekte

|-○ Boundary: Schnittstelle zu Systemakteuren (z.B. Benutzer oder System) → für Systemgrenzen

Control: Objekte die Prozesswissen repräsentieren → für Anwendungsfälle

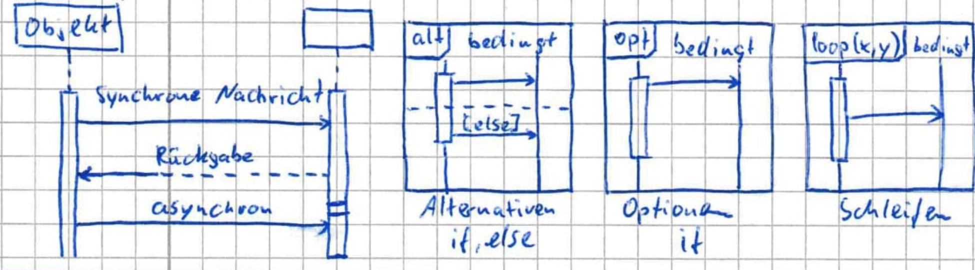
1 |-○ je Front-end-Anwendung

1 Control je Anwendungsfall

Entitäten: Bsp Terminangebot → Termin

Kommunikationsregeln im EBC:
1. Akteure können nur mit Boundarys sprechen,
2. Systemgrenzobjekte können mit Kontrollobjekten und Akteuren sprechen,
3. Entitätsobjekte können nur mit entity objekts kommunizieren.
4. Kontrollobjekte können mit boundary, entity und control-objects kommunizieren

UML-Sequenzdiagramme: stellen Nachrichten dar, die zwischen Akteuren und Objekten ausgetauscht werden



Benennung der Nachrichten als Methodenaufruf
Namen der aufzurufenden Methode
Nummer der Nachricht in Reihenfolge
Argumente der aufzurufenden Methode
Rückgabetyp der aufzurufenden Methode

Aktivitätsdiagramme eignen sich für die Darstellung bedingter Prozesse eher.

Softwarequalität: Qualitätsmerkmale von Software sind Funktionalität, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit

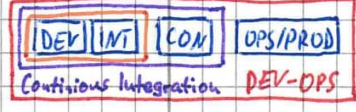
Clean Code: Bezeichnet Quellcode, Dokumente, Konzepte, Regeln und Verfahren, die intuitiv verständlich sind. Dadurch werden Programme effizienter wartbar und die Entwicklungszeiten werden verkürzt.

1. Aussagekräftige Namen: - deutliche Absicht, - konsistent, - Synonym, - aussprechbar, - keine Buchstaben ohne Sinn
2. Funktionen: - geringe Schachtelung, - Single Responsibility, - max. 2 Übergabeparameter
3. Kommentare: - möglichst wenige, - klar verständliche Kommentare
4. Klassen: Single Responsibility, so klein wie möglich

Das Gesetz von Demeter: Eine Methode einer Klasse k kann nur zugreifen auf ⊖ Methoden von k, ⊖ Methoden von Übergabeparametern, ⊖ Methoden von Instanzvariablen von k, ⊖ Methoden von selbst erzeugten Objekten

Bei Clean Code laufen alle Tests durch, es gibt keine Code-Dupletten und die Anzahl der Klassen/Methoden sind minimal

Testing- und Staging-Areas:



ENT - Entwicklungsumgebung
INT - Integrationssysteme mit Mocks externer Systeme
CON - Konsolidierungsumgebung für das Testen mit bestimmten Systemen
PROD - Produktivumgebung des Kunden
PROD → Smoketest, CON → Systemtest, Akzeptanztest
DEV → Unit Tests (Komponente) INT → Integrationstest (Zusammenspiel)

Unit-Test: Testen einzelner Softwaremodule. Integrationstest: Testen einzelner Softwaremodule im Zsm.spiel um spezifische Aufgaben und Aktivitäten durchzuführen. Systemtest: Gesamtes System auf Fehler prüfen. Akzeptanztest: Prüfen ob alle Kundenanforderungen erfüllt wurden. Smoketest: Testen der (neuen) Software vor der Allgemeinnutzung

Muster/Patterns: Beschreibung eines wiederkehrenden Problems sowie einer bewährten und generischen Lösung dafür

Microservices: Architekturstil bei dem die Anwendung in kleine Dienste aufgeteilt wird, die unabhängig voneinander installiert werden ⊖ eigenständige Prozesse ⊖ können verschiedene Sprachen implementieren ⊖ unterschiedliche Speichertechnologien ⊖ minimal zentralisierte Verwaltung

Was ist ein Pattern allgemein? Schema einer Lösung für ein bestimmtes wiederkehrendes Entwurfsproblem
⊖ Es ist kein Fertigentwurf sondern nur ein Muster

Seite 4

Klassifizierung von Patterns:
Creational Patterns → Helfen System unabhängig von Objekten zu erstellen
Structural Patterns → Umgang mit der Zusammensetzung von Klassen und Objekten in größeren Strukturen
Behavioural Patterns → Umgang mit der Wechselwirkung zwischen Objekten und Klassen
↳ Behandeln komplexe Kontrollflüsse die zur Laufzeit schwer zu verstehen sind

Strategy Pattern:
Problem: Verwandte Klassen unterscheiden sich dadurch, dass sie gleiche Aufgaben teilweise durch verschiedene Algorithmen lösen
+ Entfernt bedingte Anweisungen, + Vermeiden von Fallunterschieden + Auswahl von Varianten geschieht durch Konfiguration des Kontextes mit geeignetem Strategieobjekt.
− Erhöhter Wissensbedarf, − Erhöhter Kommunikationsaufwand, − Erhöhte Anzahl von Objekten

Singleton:
Problem: Es soll nur ein Objekt einer Klasse instanziiert werden
Lösung:

Singleton:
-singleton: Singleton -Singleton()
getSingleton(): Singleton

- einzige Instanz
- privater Konstruktor
- Zugriff auf Konstruktor

Von EBC → Sequenzdiagramm
1. Spalte: Akteure, die Anwendungsfall veranlassen
2. Boundarys die mit Akteur interagieren
3. Kontrollobjekt kann auf Fachentität zugreifen
4. Fachentitäten

Revision #4
Created 11 October 2022 23:00:14 by Martin Tienken
Updated 11 October 2022 23:13:30 by Martin Tienken