

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 :)

Software: Immaterial, 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
- Hierarchie → verwandte Elemente dürfen Eigenschaften und Implementierung teilen
- Master → Ausnutzen von Mustern
- Wiederverwendung von Komponenten
- Entwicklungskosten ↑ Verständlichkeit ↓

4. Hierarchie und Zerlegung

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

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

2. Modellierung

- Strategien → Bottom-Up, Top-Down
- Teile & Herrsche → Wiederverwendung von Implementierungen = hohe Produktivität
- High Module Cohesion → enge Verbindung der Elemente innerhalb eines Moduls
- Low inter-modular coupling → so wenig Verbindungen zwischen separaten Modulen wie möglich

5. Trennung von Zuständigkeiten

- Aufteilung eines Problems in unabhängige Teilaufgaben
- Verteilung der Arbeitslast
- Aufteilung nach Aspekten
- Model-View-Controller

3. Kapselung

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

6. Einheitlichkeit

- Software besser verständlich machen
- Bessere Wartbarkeit
- Entwurfsentscheidungen leicht verstehen

Software Engineering

Ziele: - Wirtschaftliche Art Software zu bekommen

- Methoden, Sprachen & Werkzeuge sind durch Konzepte verbunden.

- Aufgaben: - Analyse, - Spezifikation der Anforderungen, - Architektur, - Codierung und Modultest, - Integration, Abnahme, Test, Betrieb & Wartung, - Anlauf & Ersetzung

System-, Softwareentwurf, - Codierung und Modultest, - Integration, Abnahme, Test, Betrieb & Wartung, - Anlauf & Ersetzung

Ablauf: 1. Analyse und Modellierung 2. Objektorientierte Analyse 3. System- & Softwareentwurf

4. Objektentwurf

5. Codierung

6. Testen

Prozessmodelle

Agile Softwareentwicklung

Individuen und Interaktionen
Funktionierende Software
Zusammenarbeit mit Kunden
Reagieren auf Veränderungen

über Prozesse und Wartung
über Dokumentation
über Vertragsvereinbarungen
über Folgen eines Plans



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

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

Spiralmodell

Kosten Projektfortschritt

1. Analyse



2. Risiko-Analyse

+ Reduzierung des Gesamtrisikos

- + Gemeinsame Planung
- kein einheitliches Modell
- Anzahl der Spiralen die benötigt werden unklar

4. Planung der nächsten Phase

3. Entwickeln & Testen

Iteratives (iteratives) Modell:

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

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

Anforderungsanalyse

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

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 Anforderungsspezifikationen

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: Wozoll das Programm überhaupt können -> welche Funktionen sind gewünscht.

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

Bsp: Programm funktioniert, braucht aber sehr lange (Ticket automat)

Quality Attribute Utility Tree: - wesentliche Qualitätsanforderungen aus

Geschäftszielen ableiten, - Verfeinerung der Qualitätsanforderungen

- konkret messbare Szenarien, - Bewerten jedes Szenarios nach

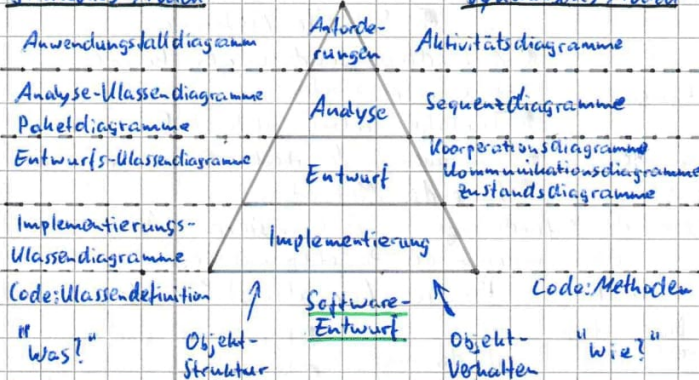
Wichtigkeit und Risiko **Unified Modelling Language (UML)**

UML ist eine grafische Modellierungssprache zur Spezifikation,

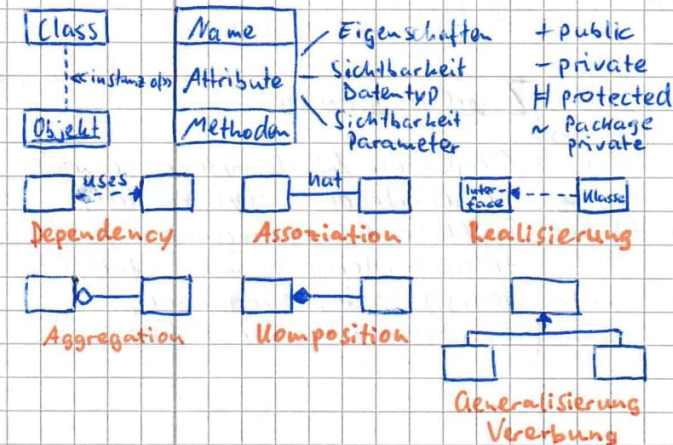
Dokumentation und Visualisierung von Software-Teilen und anderen Sprachen

Diagramme

Statisches Modell



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



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

include -> Komplexe Anwendungsfälle aufteilen

extend -> Bedingte Erweiterung

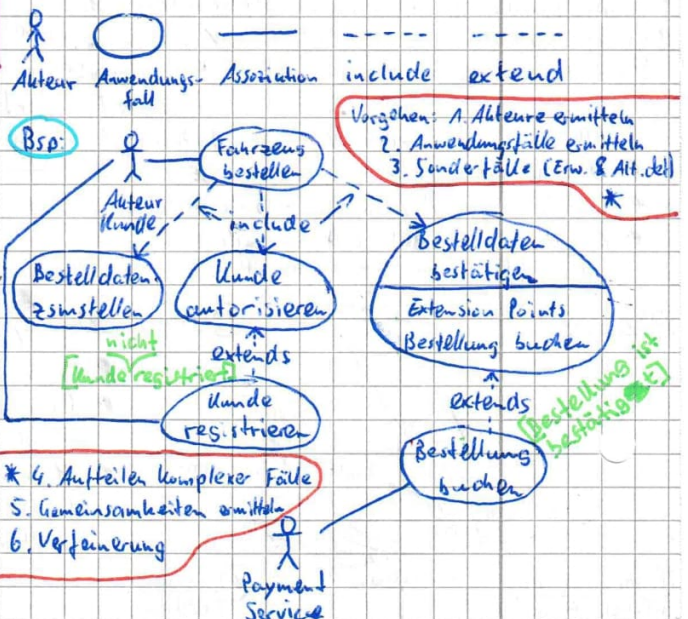
bei extend können (komplexe) Extension Points und Bedingungen ausgedrückt werden.

Schablone/Dokumentationsvorlage

Dok. Name: Registrieren, Ziel: Fahrzeug bestellen,

Vorbedingung: Portal offen, Nachbedingung: Daten

erhöhen, Leistung steht zur Verfügung, Akteure: Kunde



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

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 dann ab?

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

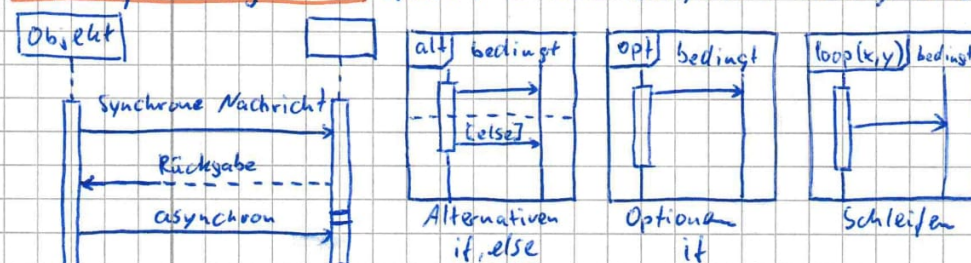
Kommunikationsregeln im EBC: 1. Akteure können nur mit Boundaries sprechen,

2. Systemgrenzobjekte können mit Kontrollobjekten und Akteuren sprechen,

3. Entitätsobjekte können nur mit Entity-Objekten 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
 Rückgabebetyp 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 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

DEV → Unit Tests (Komponente) INT → Integrationstest (Zusammenspiel) Akzeptanztest

Unit-Test: Testen einzelner Softwaremodule Integrationstest: Testen einzelner Softwaremodule im Zusammenspiel von

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 Allgemeinveröffentlichung

Muster/Patterns: Beschreibung eines wiederkehrenden Problems sowie eine 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

Martin Tienken



Wenn der Anwendungsfall sehr komplex ist.

1 TO je Frontend-Anwendung

1 TO je Anwendungsfall

Entitäten: Bsp. Terminplan

Termin

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 Variante 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 Von EBC → Sequenzdiagramm

Lösung:

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

- einzige Instanz
- privater Konstruktor
- Zugriff auf Konstruktor

1. Spalte: Akteure, die Anwendungsfall veranlassen
2. Boundaries 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