Software Engineering

Fragen und Antworten

Vorwort

An dieser Stelle ein herzliches Dankeschön, an alle die an dieser hoffentlich sinnvollen Zusammenfassung mitgearbeitet haben! Ich möchte nur einige Feinheiten des Dokumentes erklären:

Warum sind manche Wörter oder Sätze in roter Schrift dargestellt?

Wenn etwas rot markiert ist, bedeutet das nur, dass es möglicherweise keine passende Übersetzung ist, oder der Inhalt etwas deutungsbedürftig ist.

Und in pinker Schrift?

Diese Farbe definiert Kommentare, die in den Fließtext gehören oder einfach Kommentare des Autors, die für das Verständnis wichtig sein können.

Was hat es mit grau hinterlegter Schrift auf sich?

Diese Textpassagen sind möglicherweise unwichtig, da sie entweder sehr ins Detail gehen, oder mehr beantworten, als in der Frage spezifiziert ist. **Achtung!** Diese Hinterlegungen sind mit Vorsicht zu genießen. Besser zu viel lernen als zu wenig!

Inhaltsverzeichnis

Kapitel 1 – Introduction to Software Engineering

Warum ist die traditionelle Ingenieurs-Metapher von Bauingenieur-Wesen und Software-Entwicklung nur bedingt anwendbar?

Im Software-Engineering ist der Aufwand eine Dokumentation (Source Code) zu erstellen viel höher, als beispielsweise einen Bauplan für ein Gebäude zu zeichnen. Der eigentliche Bauprozess geschieht in der Software-Entwicklung auf Knopfdruck und ohne menschliches zutun, während ein Gebäude lange und schwer von mehreren Menschen gebaut werden muss.

Welche besonderen Eigenschaften hat ein Cross Functional Team?

Ein sog. Cross-Functional Team besteht nicht nur aus Leuten, die Experten auf einem Gebiet sind, sondern an mehreren Teilbereichen des Entwicklungsprozesses beteiligt sind. So sind Programmierer nicht nur dafür zuständig, Source Code zu schreiben, sondern müssen diesen auch zum Teil testen und designen. Die Development Teams kümmern sich nicht nur darum, dass die Software entsteht, sondern auch teilweise um Analyse und Vermarktung. Dadurch wird das traditionelle Wasserfallsystem aufgelockert und dynamischer gestaltet.

Was versteht man unter Software-Engineering?

SE ist eine Ingenieursdisziplin, die sich mit der Produktion von Software beschäftigt und damit alle Phasen des Entstehungsprozesses abdeckt, von Requirements Engineering bis hin zur Wartung des Systems, nachdem es in Betrieb gegangen ist.
Als Ingenieursdisziplin bezeichnet man die Fähigkeit und Kunst aus der Wissenschaft, Mathematik, Gesellschaft und dem Markt Geräte, Gebäude oder System zu bauen, die zur Steigerung der Lebensqualität von Leuten herzustellen.

Was versteht man unter Software?

Ein Computerprogramm mit der dazugehörigen Dokumentation. Softwareprodukte können für einzelne Kunden oder für einen bestimmten Markt entworfen werden.

Was ist der Unterschied zwischen Software-Engineering und Computer Science?

CS beschäftigt sich mit Grundlagen und Theorie. SE beschäftigt sich mit Möglichkeiten, aus CS gewonnene Erkenntnisse praktisch umzusetzen und brauchbare Software zu produzieren.

Was ist ein System?

Ein System ist eine sinnvolle Kombination verschiedener, untereinander abhängiger Komponenten, die zusammenarbeiten um ein Problem zu lösen, oder eine Aufgabe zu erfüllen.

Was ist der Unterschied zwischen Software-Engineering und System-Engineering?

System-Engineering beschäftigt sich mit allen Aspekten der Systementwicklung (Computersysteme), von Hardware, über Software, bis hin zu Prozess-Engineering. Software-Engineering ist ein Teil dieses Prozesses.

Was versteht man unter Software Process?

Eine Ansammlung von Aktivitäten, deren Ziel die Entwicklung oder Erweiterung von Software ist.

In welchen Bereichen gehört angemessenes Verhalten zur täglichen Arbeit eines Software-Ingenieurs?

- Vertraulichkeit
  - Sie sollten immer die Vertraulichkeit bei ihren Kunden und Arbeitgebern beachten, egal ob sie eine Vertraulichkeitseinwilligung unterschrieben haben oder nicht.
- Kompetenz
  - Sie sollten niemals ihr Level an Kompetenz falsch repräsentieren und bewusst Aufträge annehmen, die ihre Fähigkeiten übersteigen.
- Geistige Eigentumsrechte
  - Sie sollten über die lokale Rechtslage im klaren sein, besonders was Copyrights und Patente angeht.
- Computermissbrauch
  - Sie sollten niemals ihr Wissen für schädliche Zwecke missbrauchen, egal ob sie auf der Maschine eines Kollegen ein Spiel spielen, oder wissentlich Malware oder Viren verbreiten.

Was versteht man unter einem Socio-Technical System (beschreiben Sie ein Beispiel)?

Ein sehr breites System, dass aus technischen und nicht-technischen Komponenten besteht. Sie besitzen definierte operationale Prozesse, die für gewöhnlich Menschen beinhalten. Charakteristiken eines ST-Systems sind:

- **Hohe Komplexität:** Es ist praktisch unmöglich das ganze System zu verstehen. Sie besitzen Eigenschaften, die erst zustande kommen und zum ganzen System gehören, z.b. Abhängigkeit.
- **Nicht-Deterministisch:** Gibt man ihnen spezifischen Input, produzieren sie möglicherweise nicht den selben Output. Das Verhalten des Systems hängt von menschlichen Operatoren ab.

Ein **Beispiel** wäre eine Krankenhausinfrastruktur. Leute mit unterschiedlichen Qualifikationen (Ärzte, Krankenschwestern, Helfer) tragen Daten ins System ein und andere Menschen produzieren Ergebnisse daraus (Berichte, Krankenakten, Überweisungen, etc...)

Was versteht man unter einem Legacy System (beschreiben Sie ein Beispiel)?

LS sind Socio-Technische computerbasierte Systeme, die in der Vergangenheit, unter Verwendung von alter, oder schon überholter Technologie, entwickelt wurden. Sie sind oft geschäftskritische Systeme, die weitergewartet werden, weil ein Ersatz durch ein neues System zu riskant wäre. Solche Systeme haben eine sehr lange Lebensdauer (mehrere Jahre bis hin zu Jahrzehnten).

Ein **Beispiel** wäre Microsoft Windows XP und der Internet Explorer 6.0. Viele Firmeninterne Applikationen laufen nur in diesem Browser.

Was versteht man unter einem Critical System (beschreiben Sie ein Beispiel)?

CS sind Systeme, in denen ein Systemfehler für wirtschaftlichen, physischen oder humanen Bereich ernste Gefahren darstellt. Unterbereiche sind:

- **Sicherheitskritische Systeme (Safety-Critical):** Systeme in denen Fehler Menschenleben, oder schwere Umweltbelastungen zur Folge haben können.
- **Missionskritische Systeme (Mission-Critical):** Systeme in denen ein Fehler eine zielgerichtete Aktivität (Mission) scheitern lassen kann.
- **Geschäftskritische Systeme (Business-Critical):** Systeme in denen Fehler hohe Kosten für ein Unternehmen bedeuten kann.

Beispiele:

- **Safety-Critical:** Lebenserhaltungssysteme in Krankenhäusern, Flugzeugsteuercomputer
- **Mission-Critical:** Weltraumfahrtskontrollsysteme
- **Business-Critical:** Online-Bezahl-Systeme

Was versteht man unter Cloud Computing (beschreiben Sie ein Beispiel)?

Ein neuer IT-Trend, in dem Berechnung und Datenspeicherung vom Desktop und Tragbaren Computern auf große Rechen- und Datenzentren ausgelagert wird. Computerarchitekturen sollten Moore's Gesetz von der steigenden Rechenleistung auf Anzahl von Prozessorkernen und Threads pro Chip umlegen. Die Industrie, sowie Akademien müssen Systeme und Services designen, die einen hohen Grad auf Parallelisierung ausbeuten. Softwarearchitektur für massive parallele und datenintensive Aufgaben wird an Popularität gewinnen. Die Daten werden an geographisch weit entfernten Punkten gespeichert und verarbeitet. Die Datenzentren sollten daher Integrität und Privatsphäre der Benutzer respektieren und einhalten. Die Rechenleistung muss „elastisch“ sein und sich schnell den Anforderungen anpassen können.

Beispiel: Dropbox (Daten im Internet speichern und auf jedem Gerät abrufen), Google Services (Dokumente bearbeiten, Termine managen, etc...)

Beschreiben Sie die Herausforderungen im Software-Engineering?

- **Komplexität:** Softwareentitäten sind viel komplexer für ihre Größe als jedes andere menschliche Konstrukt. Kein Teil ist wie das andere, über dem Statement-Level. Die Aufgabe des DevTeams ist es, eine Illusion von Einfachheit zu schaffen. Skalierung einer Software bedeutet nicht die Größe anzuheben, sondern den Funktionsumfang zu erweitern. In den meisten Fällen interagieren Elemente nicht linear untereinander. Aus dieser Komplexität resultieren auch Managementprobleme.
- **Konformität:** Software muss anpassungsfähig sein, weil sie als solche wahrgenommen wird. Sie muss mit anderen Benutzeroberflächen kompatibel sein. Dazu reicht kein umdesignen der Software selbst.
- **Änderbarbeit:** Software kann und muss leicht geändert werden können. Sie ist eingebunden in ein System aus Anwendungen, Benutzern, Rechten und maschinelle Vehikel, die sich ändern und damit muss auch die Software geändert werden.
- **Unsichtbarkeit:** Software ist unsichtbar und nicht visualisierbar. Sie ist kein Teil des Raumes. Sobald wir versuchen Software darzustellen, erhalten wir ein Diagramm chaotischer Struktur, welches ineinander übergreift. Die einzelnen Graphen des Diagramms symbolisieren den Kontroll- und Datenfluss, Abhängigkeitsmuster, Zeitliche Sequenzen, etc...

Was versteht man im Kontext von Software-Engineering unter „No Silver Bullet“?

Es gibt keinen technologischen Durchbruch, der magische Ergebnisse liefert, aber es gibt ein Versprechen auf kontinuierlichen Fortschritt.

Beschreiben Sie drei mögliche Ansätze, um die Effizienz in der Softwareentwicklung zu steigern?

- **Buy vs build:** Software für bestimmte Zwecke ist oft billiger zu erwerben, als herzustellen. Je mehr Benutzer eine Software hat, umso geringer sind die Pro-Kopf Kosten, da die Herstellung der Software viel kostet, die Verbreitung jedoch kaum.
- **Raffinierung der Anforderungen und schnelle Prototypisierung:** Der Kunde weiß nicht, was er will. Daher ist die wichtigste Aufgabe des Software-Engineers das exakte Herausfinden der Anforderungen an die Software und zu wissen, was genau man herstellen möchte. Software wird nicht gebaut, sie wächst.
- **Großartige Designer:** Softwaredesign ist ein kreativer Prozess, daher ist der Hauptgrund für die Entstehung guter Software die Leute die daran arbeiten. Der Unterschied zwischen einem normalen und einem großartigen Designer ist eine Zehnerpotenz.[1]

[1] *Vgl.: Schwarzl Patrick, SE_Kontrollfragen.pdf, ITM09*

Kapitel 2 - Requirements Engineering

Was versteht man unter Requirements Engineering?

Die Anforderungen an ein System werden festgelegt. Dazu gehören die erwünschten Eigenschaften, Restriktionen von Systemoperationen, sowie der Softwareentwicklungsprozess. RE ist die Kommunikation zwischen Kunden, Benutzer und Entwickler. Es ist kein rein technischer Prozess, da die Anforderungen von Benutzervorlieben, Abneigungen und Vorurteilen, sowie politischen und organisationellen Aspekten beeinflusst werden.

Lastenheft: Wenn eine Firma ein Softwareprojekt in Auftrag geben möchte, muss sie ihre Anforderungen an die Software abstrakt genug formulieren, ohne die Lösung dafür zu konkretisieren *(Solutlion-less-requirements).* Das ganze muss niedergeschrieben werden, sodass mehrere Teilhaber des Vertrages ihre eigenen Ideen und Vorschläge einbringen können, um die Spezifikationen des Systems zu verfeinern.

Pflichtenheft: Sobald das Lastenheft fertig ist, muss der Beauftragte eine detailliertere Systemdefinition für den Kunden schreiben, sodass derjenige die Software versteht und deren Funktion nachvollziehen und validieren kann.

Beschreiben Sie die beiden Levels von Requirements, für welche Lesergruppen sind diese Levels gedacht?

- **User requirements:** Sind statements in natürlicher Sprache mit Diagrammen, die beschreiben, welche Dienste das System ausführen wird, sowie die Einschränkungen, unter denen es funktionieren wird.
  - **Lesergruppen:** Systembenutzer, Manager von Kunden/Beauftragen, Systemarchitekten
- **System requirements:** Sind präzise Erklärungen über die Funktionen, Dienste und Einschränkungen des Systems Auskunft geben. Die System requirements sollten exakt definieren, was implementiert wird, da sie möglicherweise Teil des Vertrages zwischen Kunden und Entwickler sein können.
  - **Lesergruppen:** Systemarchitekten, Softwareentwickler

Beschreiben Sie die Klassifizierung der System Requirements? (Ganze Frage)

- **Functional requirements:** Statements von Diensten, die das System anbietet, wie es mit Inputs umgeht und wie es sich in verschiedenen Situationen verhält. Die Functional requirements statituieren auch, was das System nicht tun soll.
- **Non-functional requirements:** Sind die Begrenzungen der Dienste, die das System anbietet, wie zeitliche Beschränkungen, oder der Entwicklung und der Standards.
- **Domain Requirements:** Anforderungen der Domäne direkt. Sie bestimmen Verhalten und Einschränkungen des Systems und somit die funktionalen bzw. nicht-funktionalen Requirements.
  Problem dabei: Software-Engineers verstehen möglicherweise die Charakteristiken der Domäne nicht und können oft nicht sagen, ob ein Domain-Requirement ausgelassen wurde oder in Konflikt zu anderen steht.

Was versteht man unter einem Requirements Document?

Das RD oder Software Requirements Specification (SRS) ist die offizielle Angabe dessen, was die Systementwickler implementieren sollen.

Aufgrund der Diversität der unterschiedlichen Nutzer und Leser muss das Dokument ein Kompromiss sein zwischen:

- Die Anforderungen den Kunden verständlich machen
- Die Anforderungen präzise und im Detail für Entwickler und Tester erklären, sowie Informationen über mögliche Systemevolutionen beinhalten.

Was ist ein Stakeholder?

Eine Person oder ein Unternehmen, die/das in irgendeiner Weise am Projekt beteiligt ist, sei es Auftraggeber oder Benutzer.[2]

Was ist bei der Definition von nicht-funktionalen Anforderungen zu beachten?

Ein gängiges Problem bei der Definition von nicht-funktionalen Anforderungen ist, dass Benutzer diese Anforderungen oft als allgemeine Ziele anregen, wie etwa einfache Bedienung, einfache Konfiguration oder schnelle Reaktionszeit.

Beispiel: "*Das System soll für medizinisches Personal einfach zu bedienen sein und so organisiert sein, dass Benutzerfehler möglichst minimiert werden.*"

Besser: Medizinisches Personal soll nach vier Stunden Training in der Lage sein, das Programm mit all seinen Funktionen zu bedienen. Nach dem Training soll die durchschnittliche Anzahl an Fehlern pro Stunde, *von erfahrenen Benutzern*, zwei nicht übersteigen.

- **Geschwindigkeit**
  - Anzahl an ausgeführten Vorgängen / Sekunde
  - Aktualisierungszeit
  - Reaktionszeit
- **Größe**
  - Physikalische Größe (Mbytes)
  - Anzahl an benötigten Prozessoren
- **Einfachheit**
  - Einlernzeit
  - Anzahl an Hilfestellungen
- **Robustheit**
  - Zeit zu reinbeetriebnahme nach einem Fehler
  - Anzahl an Events die Fehler verursachen
  - Wahrscheinlichkeit von Datenverlust bei einem Fehler

[2] *Vgl. http://en.wikipedia.org/wiki/Requirements_analysis#Stakeholder_identification*

Was sind Domänen-Anforderungen?

Sind die Anforderungen an das System, die aus der jeweiligen Domäne (Beispiele: Rechtsdomäne, Kaufmannsdomäne, Materialverarbeitung, Handelsdomäne, etc...) stammen und die Charakteristiken und Beschränkungen jener reflektieren. Sie können funktionale oder nicht-funktionale Anforderungen sein.

Software-Engineers verstehen nicht immer die Charakteristiken jeder Domäne und können oft nicht genau sagen, ob eine Domänen-Anforderung erfüllt wurde, oder nicht.

Wodurch unterscheidet sich der Requirements Engineering Process in herkömmlicher und agiler Projektmethodik?

- **Traditionelle (Herkömmliche) Herangehensweise:** Für größere Projekte gibt es eine klar identifizierbare Requirements Engineering Phase vor der Implementation.
- **Agile Herangehensweise:** Anforderungen werden während der Entwicklung des Systems dynamisch eruiert.

Was ist eine Machbarkeitsstudie?

Eine Beurteilung, ob das System sinnvoll und nützlich für das Geschäft ist.

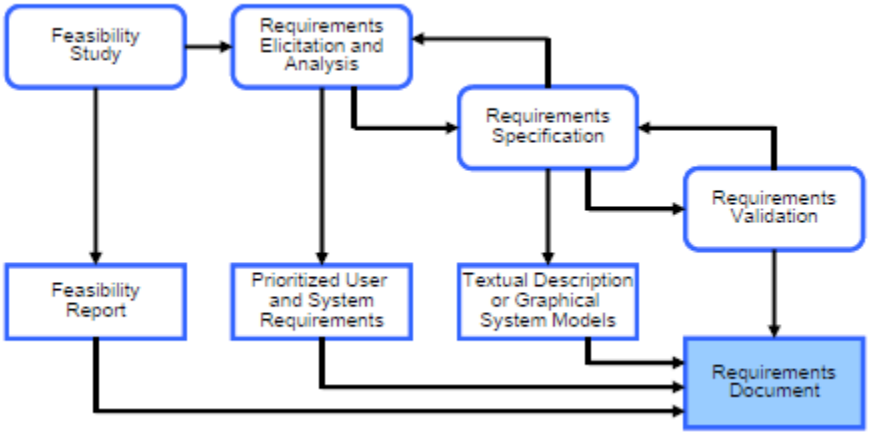
Die Resultate der Machbarkeitsstudie sollten Aussage darüber treffen, ob es Sinn macht, das Projekt weiter voranzutreiben, oder nicht. Resultate sollten folgende sein:

- Trägt das System zum Zweck der Organisation bei?
- Kann das System, im Rahmen der Zeitvorgabe und des Budgets, unter Benutzung aktueller Technologien, implementiert werden?
- Kann das System mit anderen benutzten System integriert und kombiniert werden?

Wenn diese Fragen mit Nein zu beantworten sind, hat das System keinen Wert für das Geschäft.

Skizzieren und erklären Sie den Requirements Engineering Process?

1. **„Entdeckung“ der Anforderungen:** Interaktion mit Stakeholdern, um ihre Anforderungen zu erfahren.
2. **Klassifizierung und Organisation:** Unstrukturierte Ansammlungen in zusammenhängende Gruppen (Cluster) organisieren.
3. **Priorisierung und Verhandlung:** Konfliktierende Anforderungen von mehreren Stakeholdern auflösen.
4. **Spezifizierung:** Formelle oder nicht-formelle Systemmodelle produzieren.



Welchen Prüfungen sollte ein Requirements Document unterzogen werden?

1. Kurze Sätze verwenden
2. Nie mehr als eine Anforderung pro Satz
3. Jargon, Abkürzugen und Akronyme vermeiden
4. Kurze Absätze verwenden
5. Wo möglich, Listen und Tabellen verwenden
6. Konsistente Terminologie verwenden
7. Wörter wie *soll*, *sollte* und *muss* regelmäßig verwenden
8. Keine verschachtelten Bedingungsklauseln verwenden
9. Aktiv statt passiv schreiben
10. Komplexe Beziehungen nicht in natürlicher Sprache ausdrücken
11. Keine anonymen Referenzen benutzen
12. Auf Rechtschreibung und Grammatik achten

Was versteht man unter Requirements Management?

RM ist das verstehen und kontrollieren von Änderungen in den Systemanforderungen. Zusätzlich müssen neue Anforderungen, die entstehen gedeckt werden.

Kapitel 3 – Architektur & Design

Was versteht man unter Software Design?

Die Essenz des Softwaredesigns ist es, Entscheidungen über die logische Organisation von Softwaresystemen zu treffen.

Erklären Sie drei Vorteile eines Design-Dokuments.

- **Stakeholderkommunikation:** Die Architektur ist eine hochwertige Präsentation des Systems, die als Diskussionsgrundlage für verschiedene Stakeholder dienen kann.
- **Systemanalyse:** Eine Systemarchitektur explizit zu machen bedarf einiger Analysen. Architekturentscheidungen haben einen großen Einfluss darauf, ob das System die kritischen Anforderungen wie die Performance, Verlässlichkeit und Wartbarkeit überhaupt erfüllen kann.
- **Breitflächige Wiederverwertbarkeit:** Systemarchitekturen sind kompakte Beschreibungen über die Organisation eines Systems und wie dessen Komponenten miteinander operieren. Sie sind oft die selben für Systeme mit ähnlichen Anforderungen.

Beschreiben Sie fünf non-functional Requirements, die für die Wahl der richtigen Software-Architektur wichtig sind.

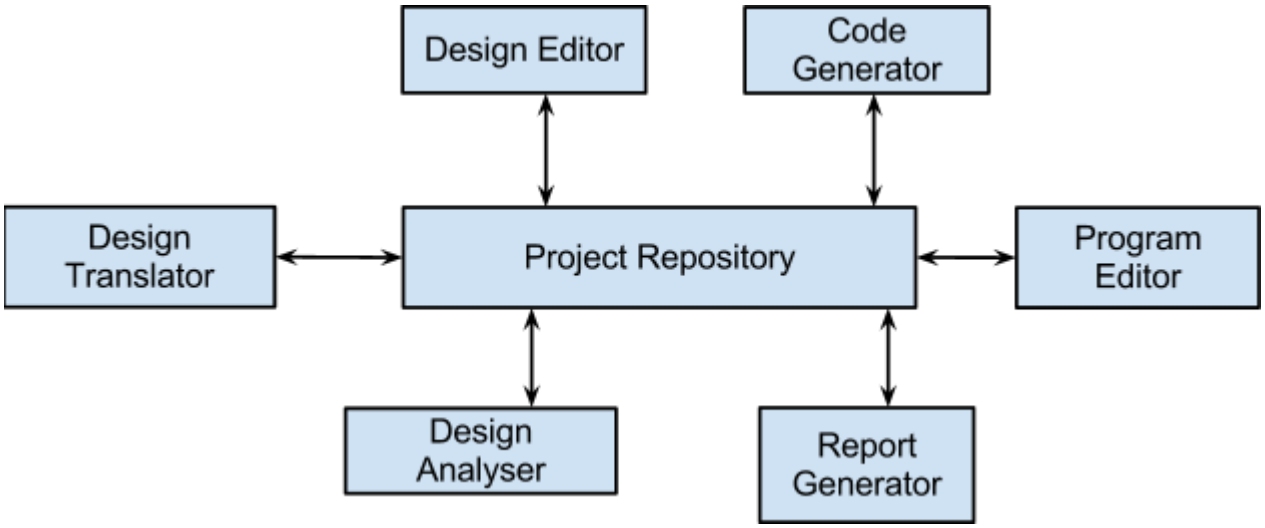
- **Performance:** Die Architektur sollte so designed sein, dass kritische Operationen in einer geringen Anzahl an Sub-Systemen lokalisiert sind, die möglichst wenig Kommunikation untereinander haben. (Large-grain vs. Fine-grain Komponenten)
- **Security:** Es sollte eine Schichtenstruktur für die Architektur genutzt werden, bei der die kritischten Teile in den innersten Schichten geschützt sein sollten. Diese innersten Schichten sollten auf höchstem Niveau auf Sicherheit validiert werden.
- **Safety:** Die Architektur sollte so angelegt sein, das sicherheitsrelevante Operationen in entweder einem einzelnen Subsystem, oder in einer geringen Anzahl von Subsystemen lokalisiert sind. Das reduziert die Kosten und Probleme der Safety-Validation.
- **Availability:** Die Architektur sollte so designed sein, dass sie redundante Komponenten enthält, sodass es möglich ist, Teile zu ersetzen oder zu erweitern, ohne das System anhalten zu müssen.
- **Maintainability:** Die Systemarchitektur sollte fine-grain Komponenten enthalten, die in sich abgeschlossen sind, sodass sie einfach ausgetauscht werden können.

Was ist ein Pattern(Entwurfsmuster)?

Pattern dienen zum Präsentieren, Teilen und zum Wiederverwenden von Wissen über Software Systeme. Sie sind im wesentlichen stilisierte, abstrakte Beschreibungen über “best practices”, die auf verschiedenen System geprüft und getestet wurden.

Skizzieren und erklären Sie das Repository Model, was sind die Vor- und Nachteile?

Alle Daten werden in einer zentralen Ablage verwaltet. Komponenten interagieren nicht direkt miteinander, sondern nur durch das Repository.



Vorteile:

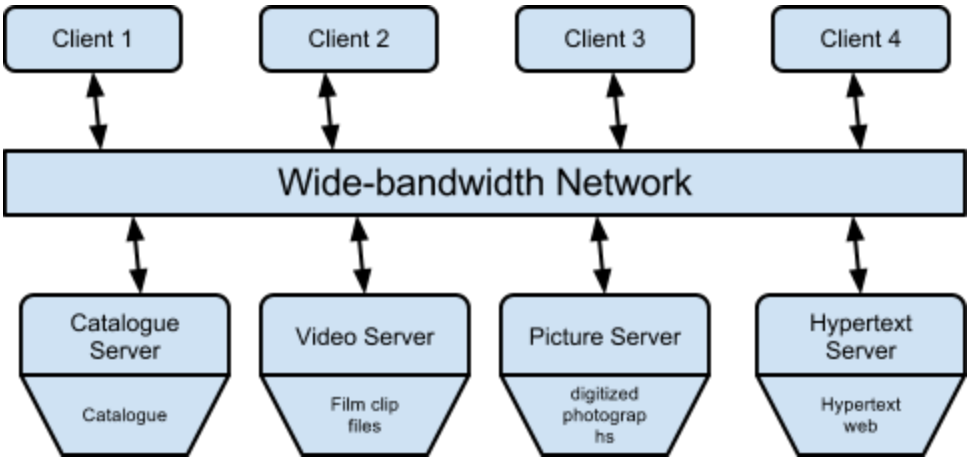
- Effizient größere Mengen an Daten zu teilen.
- Keine Notwendigkeit, Daten direkt zwischen den Sub-Systemen zu übermitteln.
- Komponenten sind unabhängig.
- Daten können einheitlich verwaltet werden.
- Das Interface der Verteilung ist sichtbar durch das Repository Schema. Außerdem entwickelt es neue Werkzeuge.

Nachteile

- Fehler im Repository befallen das gesamte System.
- Kommunikation ist durch das Repository ineffizient. Sub-Systeme müssen mit dem Repository kompatibel sein.
- Weiterentwicklung ist schwierig, da große Mengen von Informationen durch das Datenmodell generiert werden.

Skizzieren und erklären Sie das Client-Server Model, was sind die Vor- und Nachteile?

Die Funktionalität wird in Diensten organisiert, bei dem jeder Dienst von einem separaten Server zur Verfügung gestellt wird.



Vorteile:

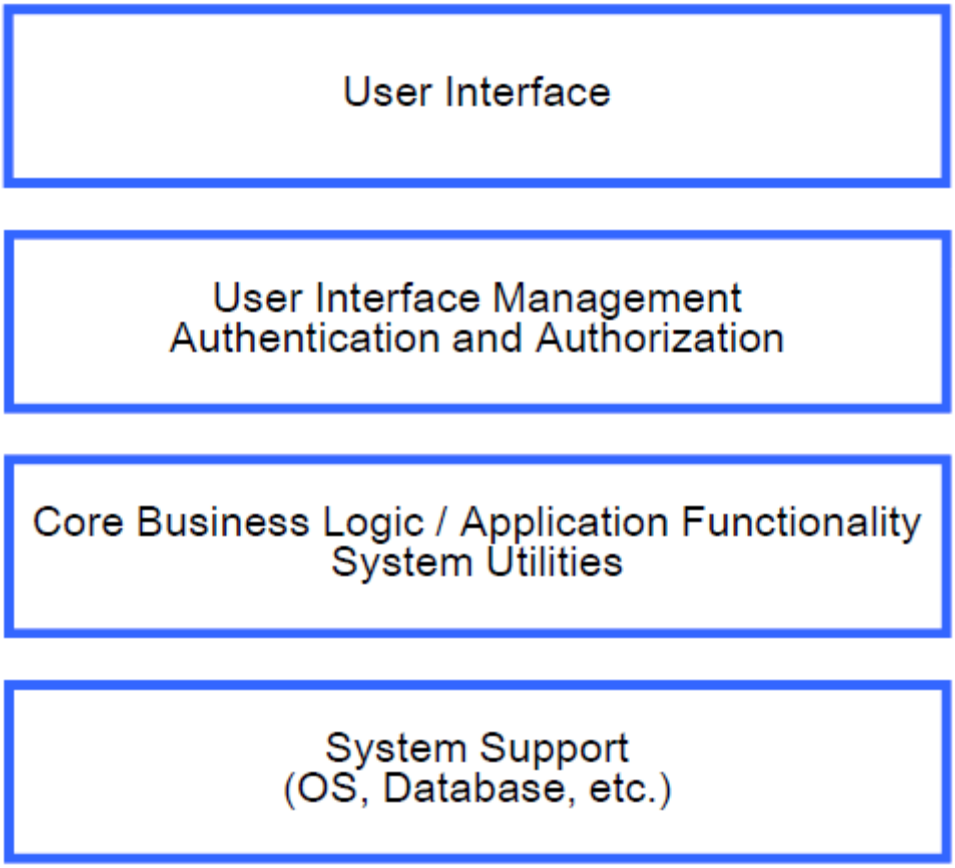
- Server können über dem Netzwerk verstreut sein
- Es ist leicht neue Server zu integrieren und sie in das System einzubauen oder Server upzugraden ohne, dass andere Teile des Systems davon beeinflusst werden

Nachteile

- Jeder Dienst ist ein einzelner Fehlerpunkt, der anfällig ist für Dienstverweigerung (denial of service) oder Serverversagen ist.

Skizzieren und erklären Sie das Layered Model, was sind die Vor- und Nachteile?

Das System ist in Schichten organisiert. Jede Schicht hat ihre eigene Funktionalität.



Vorteile:

- Leichtere Multi-Plattform-Implementationen.
- Leichtes Ersetzen von ganzen Schichten, solange die Schnittstellen erhalten bleiben.
- Redundante Einrichtungen können in jeder Schicht zur Verfügung gestellt werden um die Zuverlässigkeit des Systems zu erhöhen.

Nachteile:

- Klare Separation zwischen Schichten anzubieten kann schwierig sein, da höhere Schichten möglicherweise direkt mit Schichten interagieren müssen, die nicht unmittelbar unter ihnen liegen.
- Wegen vielen Interpretationsebenen einer Systemanfrage kann die Performance ein Problem werden.

Beschreiben Sie die vorgestellten Modular Decomposition Styles.

Sub-system Decomposition:

Nachdem eine Systemorganisation beschlossen wurde, müssen die Subsysteme in Module zerlegt werden. Eine gängige Methode ist die Objektorientierte Dekomposition, bei der das System in untereinander kommunizierende Objekte zerlegt wird.

Ein Objektorientiertes Modell strukturiert das System in einen Satz aus lose zusammenhängenden Objekten mit klar definierten Schnittstellen. Objekte rufen Dienste anderer Objekte auf. Eine Objektdekomposition beschäftigt sich mit Objektklassen, deren Attributen und Operationen. Objekte sind häufig Repräsentationen von Dingen aus der realen Welt sodass die Struktur leicht verständlich ist.

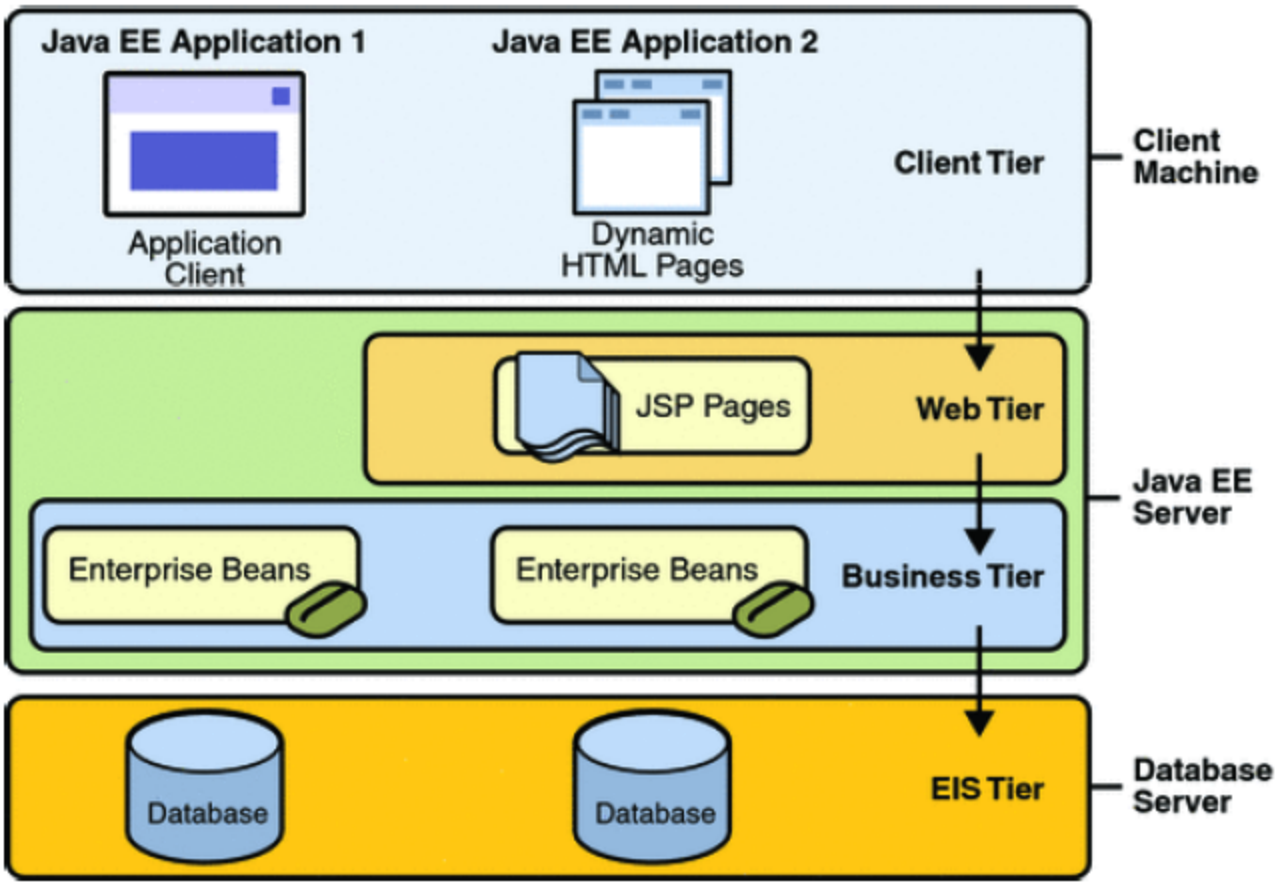
Beschreiben Sie die vorgestellten Control Styles.

- **Centralized control:** Eine Komponenten ruft Dienste von anderen Komponenten im System auf (Reflektiert die Methode/Prozedur/Subroutine Aufrufe in Programmiersprachen). → Ein System fungiert als System Controller und ist verantwortlich für das Management und die Ausführung von anderen Komponenten.
    - *Call-return model:* Die Kontrolle beginnt in der Spitze einer Subroutinenhierachie und durchläuft die unteren Ebenen des Baumes. Dieses Modell ist nur auf sequentielle Systeme anwendbar.
    - *Manager model:* Ein Manager kontrolliert das Starten, Stoppen und Koordinieren anderer Prozesse. Ein Prozess ist eine Komponente oder ein Modul, das parallel zu anderen Prozessen ausgeführt werden kann. Dieses Modell ist auf nebenläufige Systeme anwendbar (Event-loop model).
- **Event-based control:** Das System antwortet auf asynchrone Ereignisse, die entweder von Subsystemen oder anderen Komponenten in der Systemumgebung kommen. → Ereignisgesteuerte Kontrollmodelle werden durch extern erzeugte Ereignisse gesteuert. Der Unterschied zwischen einem Ereignis und simplem Input ist, das Timing des Ereignisses liegt außerhalb der Kontrolle des Prozesses, welcher das Ereignis handlen soll.
    - *Broadcast model:* Ein Ereignis wird an alle Komponenten gesendet. Jede Komponente, die darauf programmiert ist, das Ereingnis zu handlen, kann darauf antworten. Dieses Modell ist effektiv um Komponenten, die über viele Computer in einem Netzwerk verteilt sind, zu integrieren.
    - *Interrupt-driven model:* Dieses Modell wird in Real-Time Systemen verwendet, in denen externe Interrupts von einem Interrupt-Handler entdeckt werden. Diese werden einer anderen Komponente übergeben, die diese bearbeiten.

Nennen Sie fünf „Gerüche“ von „verrottender“ Software.

- **Rigidity (Starrheit):** Schwer zu verändern
- **Fragility (Zerbrechlichkeit):** Bricht an vielen Stellen nach einer einzelnen Veränderung
- **Immobility (Unbeweglichkeit):** Enthält nützliche Teile, die jedoch nur unter großem Aufwand und Risiko separiert werden können.
- **Needless Complexity (Unnötige Komplexität):** Software enthält Elemente, die momentan nicht nützlich sind.
- **Opacity (Transparenz):** Schwer verständlich.
- **Needless Repetition (Unnötige Wiederholungen):** Häufiges Cut&Paste.
- **Viscosity of software** - Hoch, wenn Methoden, die das Design erhalten schwieriger sind als Hacks.
- **Viscosity of environment** - Hoch, wenn die Entwicklungsumgebung langsam und ineffizient ist.

Was versteht man unter einer Distributed Multitiered Application im Kontext von JEE (Skizze und Erklärung)?

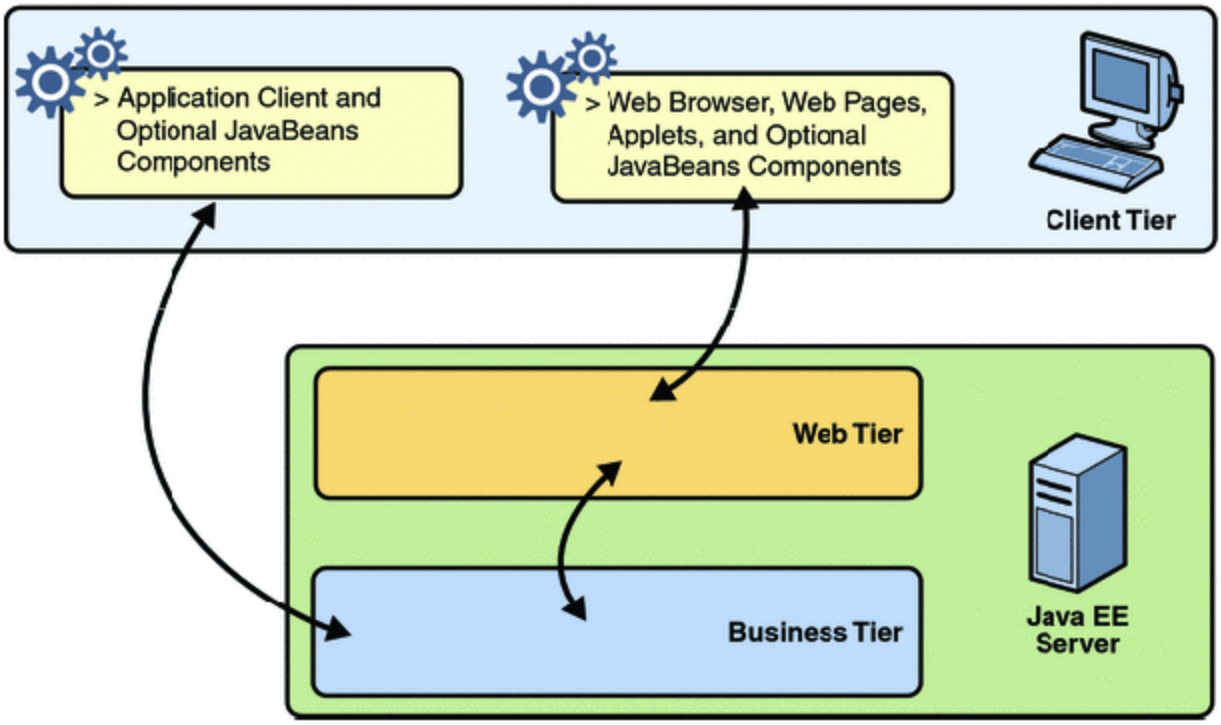


Die diversen Applikationen, die eine Java EE Applikation ausmachen sind auf verschiedenen Geräten installiert, abhängig von der Stufe in der Mehrstufigen Java EE Umgebung, zu der die Applikationskomponente gehört.

Java EE Applikationen werden generell als 3-Stufige Applikationen gesehen, weil sie über 3 Standorte verteilt sind: Clientrechner, die Java EE Servermaschine und die Datenbank- oder Legacymaschinen im Backend.

Welche Client-Typen werden von der JEE Architektur unterstützt (Skizze und

Erklärung)?



- **Web Client**
  Ein Webclient ist ein Thin-Client, der gewöhnlich keine Datenbankabfragen macht, keine komplexen Businessregeln ausführt, oder sich zu Legacy-Applikationen verbindet. Solche Heavyweight-Operationen werden an Enterprise Beans abgeladen, die am Java EE Server ausgeführt werden.

  Ein Webclient besteht aus zwei Teilen:

  - **Dynamische Websites**, die verschiedene Arten von Markup-Language beinhalten, die von Webkomponenten generiert werden, die in der Webstufe laufen.
  - **Webbrowser**, welche die Seiten, die vom Server empfangen werden darstellen.
- **Applets**
  Eine Website, die von der Webstufe empfangen wird kann ein eingebettetes Applet enthalten. Ein Applet ist eine kleine Clientapplikation, die in Java geschrieben ist und an der Java Virtual Machine im Webbrowser ausgeführt wird.
- **Applikations-Clients**
  Ein Applikations-Client läuft auf der Client-Maschine und stellt eine Möglichkeit für Benutzer bereit um Aufgaben zu bewältigen, die ein umfangreicheres Interface benötigen, als mit Markup-Languages angeboten werden kann. Applikations-Clients greifen direkt auf Enterprise Beans zu, die in der Business-Stufe laufen.

Kapitel 4 - Implementation

Was versteht man unter der Implementierung eines Software-Systems?

Implementation ist die Transformation eines Designmodells in ausführbaren Code.

Für die Ausführung dieses Softwareentwicklungsschrittes ist es essentiell die richtige(n) Programmiersprache(n) als Implementationsvehikel auszuwählen.
Die ausgewählte Programmiersprache muss dem Applikationsprogrammierer die Macht geben die Aufgabe in disziplinierter Art zum Ausdruck zu bringen.

Um die Anforderungen jedes Anwendungsgebietes zu erfüllen wurden tausende Programmiersprachen designed. Viel wurden nur zu Forschungszwecken entwickelt und andere mit dem Ziel des Produktionsgebrauchs. Manche Sprachen sind für allgemeine Zwecke, während andere für bestimmte Domänen entwickelt wurden.

Einige der am weitesten verbreiteten angewandten Programmiersprachen sind:

- C
- C++
- Java
- Perl
- Visual Basic
- Python
- C#

Beschreiben Sie drei mögliche Abstraktionsformen die von Programmiersprachen eingesetzt werden.

- **Control abstraction:** Kontrollabstraktion beschreibt eine klar definierte Operation, die Ausführungssequenzen von Statements (Actions) im Programm darstellt und intra-prozedurale Kontrollstrukturen wie **if-then-else, while** und **switch** Statements beinhaltet.
- **Procedural abstraction:** Prozedurale Abstraktion beschäftigt sich mit dem Zerlegen eines Programmes in Einheiten, die typischerweise spezielle Zwecke haben. PA schließt die Definition eines Interfaces und Implementation der Einheit ein. Einheiten setzen sich aus lokalen Daten und Code, der in der Lage ist auf nicht-lokale Daten zuzugreifen, zusammen. Indem man Implementation versteckt ist man besser in der Lage Implementation zu ändern. Dies warf die Idee des **Information Hiding** auf.
- **Data abstraction:** Abstrakte Datentypen geben Programmierern die Möglichkeit ihre eigenen Datentypen sicher zu definieren. Moderne Konzepte von abstrakten Datentypen beinhalten:
    - Einkapseln oder Einschließen der Daten und der damit verbundenen Operationen.
    - Den Datentyp benennen.
    - Die Operatoren mit Beschränkungen versehen.
    - Regeln, welche die Sichtbarkeit der Daten spezifizieren.
    - Spezifizierung von Implementierung trennen.

Wozu verwenden Programmiersprachen das Konzept der starken Typisierung?

Starke Typisierung wird momentan als fundamentales Feature moderner Programmiersprachen akzeptiert, egal ob durch Compilezeit- oder Laufzeitüberprüfung.
Starke Typisierung sichert die Verlässlichkeit von Code und bietet eine zusätzliche Schicht um ein Programm semantisch zu überprüfen.
Sowie typisierte Systeme für Programmiersprachen Generität und Polymorphismus entwickelten, wurden sie direkt mit dem Thema der **Software-Wiederverwendung** in Verbindung gebracht.

Welche 5 Konzepte kennzeichnen objekt-orientierte Programmiersprachen?

- **Abstraktion**
  - Eine Abstraktion kennzeichnet die essentiellen Charakteristiken eines Objektes, die es von allen anderen Arten von Objekten unterscheidet. So liefert sie klar definierte konzeptuelle Grenzen, die sich auf die Perspektive des Betrachters beziehen.
    Implementationen: Klasse, Interface, Konstruktor/Destruktor
- **Kapselung**
  - Kapselung ist der Prozess der Bereichsbindung von Elementen einer Abstraktion, die ihre Struktur und Verhalten darstellt.
    Implementationen: Interne Daten/Operationen, Zugriffsmodifikatoren
- **Modularität**
  - Modularität ist die Eigenschaft eines Systems, das in eine Ansammlung von zusammenhängenden und lose verbundenen Modulen zerlegt wurde.
    Implementationen: Package, Klasse + Interface
- **Hierarchie**
  - Hierarchie ist ein Reihen oder Ordnen von Abstraktionen. Die beiden wichtigsten Hierarchien in einem komplexen System sind:
    - seine Klassenstruktur (“ist ein”-Hierarchie)
    - seine Objektstruktur (“teil von”-Hierarchie)

    Implementationen: Einzelvererbung, Mehrfachvererbung, Aggregation, Komposition
- **Typisierung**
  - Typisierung ist das Erzwingen einer Klasse eines Objektes, sodass Objekte von anderen Typen nicht ausgetauscht werden können, oder bestenfalls auf sehr streng eingeschränkten Wege.
    Implementationen: Klasse, Dynamische/Statische Typisierung, Polymorphismus

Wozu wird in Programmiersprachen das Konzept der Exceptions eingesetzt?

Exceptions sind Features, die in Programmiersprachen eingebaut wurden, um dem Programmierer die Fähigkeit zu geben, zu spezifizieren, was passieren soll, wenn ungewöhnliche Bedingungen/ Situationen bei der Ausführung auftreten.

Statements werden in einem try-Block ausgeführt, welcher mit einem catch-Block assoziiert ist, der die Exceptions, die geworfen werden können, behandelt. Exceptions werden lokal, oder im ersten geeigneten typisierten Handler in der Call-Kette abgearbeitet.

Die Ansammlung an Exceptions, die von einer Funktion direkt oder indirekt geworfen werden können, kann als Teil der Funktionsdeklaration aufgelistet werden.

Was versteht man unter dem Begriff “Concurrency”?

Das anfängliche Gleichzeitigkeitskonstrukt war eine Co-Routine, welche die quasi-parallele Ausführung eines Programmes erlaubte. Dijkstra hat gezeigt, wie man **Semaphore** benutzt, um vielerlei Synchnronisationsprobleme zu lösen.

Ein **Monitor** ist ein Objekt oder Modul, das für die sichere Verwendung in mehr als einem Prozess/ Thread beabsichtigt ist. Seine Methoden werden mit wechselseitiger Ausschließung ausgeführt, also führt zu jedem Zeitpunkt maximal ein Thread eine seiner Methoden aus.

Java hat das Monitorkonzept erweitert, indem es Zugriff zu gemeinsam benutzten Daten mittels synchronosierten Methoden und Codeblöcken gekapselt hat.

Dijkstras essende Philosophen

Fünf stille Philosophen sitzen an einem runden Tisch vor jeweils einem Teller Spaghetti. Neben jedem Teller ist eine Gabel. Jeder Philosoph muss alternierend denken und essen. Ein Philosoph kann nur essen, wenn er jeweils eine Gabel links und rechts hält. Jeder Philosoph kann eine benachbarte Gabel aufheben, wenn sie verfügbar ist und sie wieder abstellen. Dies sind separate Aktionen: Gabeln müssen mit einzelnen Aktionen aufgehoben und abgelegt werden. Das Problem liegt darin, ein Benehmen zu designen sodass keiner der Philosophen verhungert, also dauerhaft zwischen Denken und Essen abwechseln kann.

Erklären Sie den Begriff "Visual Programming".

Das Ziel von visueller Programmierung ist es, *mittels Vorrichtungen, welche den mentalen Aufwand der benötigt wird, um die verfügbare Information zu begreifen und Sinn daraus zu schöpfen*, die kognitive Belastung von menschlichen Programmierern zu reduzieren.

Visuelle Programmierung bezieht sich auf jedes System, das erlaubt ein Programm zwei- oder mehrdimensional zu spezifizieren.

Viele dieser Systeme hatten Vorteile, die bei der Demonstration mit Spielzeugprogrammen spannend und intuitiv wirkten, aber verliefen sich in schwierige Probleme, als versucht wurde, sie zu realistisch-dimensionierten Programmen zu erweitern.

Visuelle Programmierung bewegte sich in zwei Richtungen:

- **Visuelle Programmierung durch Programmierungsumgebungen**
  Diese Richtung umfasst die Eingliederung von visuellen Ausdrucksformen für die spezifischen Teile der Entwicklung, in denen visuelle Ausdrucksformen offensichtliche Vorteile für menschliche Programmierer brachten, in integrierte Programmiersprachumgebungen.
- **Domänenspezifische Visuelle Programmiersprachen**
  Weil viele domänenspezifische Sprachen auf spezifisches Publikum gezielt waren sowie die spezifischen Applikationen, begannen diese Forscher sich mehr auf die Zielgruppen zu konzentrieren, für die diese Programmiersprachen gedacht waren (**end-user programming)**.

Erklären sie domänen-spezifische Sprachen (DSLs)?

Einen domänen-spezifische Sprache ist eine Programmiersprache oder Spezifikationssprache, die speziell dafür vorgesehen ist, eine bestimmte Problemdomäne, Problemrepräsentationstechnik und/oder eine bestimmte Lösungstechnik zu behandeln.

Beispiele zu DSLs:

- **DSLs von technischen Domänen:** HTML, Logo, SQL, etc.
- **DSLs von professionellen Domänen:** Zur Beschreibung von Telefonrechnungen, Versicherungsverträgen, Zustandsmaschinen in eingebetteten Systemen, etc.

im Gegensatz zu:

- Programmiersprachen für allgemeine Zwecke (wie C oder Java)
- Modelliersprachen für allgemeine Zwecke (wie UML)

Vorteile von DSLs:

- Es wird weniger Code benötigt um die Domänenaspekte zu beschreiben, daher bessere Lesbarkeit und einfachere Wartung
- Semantischer, daher sind Einschränkungen leichter zu analysieren
- Schnellere Implementation (falls ein Generator verfügbar ist)
- Bessere Implementationsqualität, da der Generator stets konsistenten Code erzeugt
- Transformation von DSL zu technischen Code kann verändert werden, ohne die Beschreibung der Domäne zu verändern zu müssen
- Trennung von Domäne und technischen Aspekten, daher Möglichkeit plattformunabhängiger Implementation
- Bessere Integration von Domänenspezialisten in den Entwicklungsprozess

Parserbasiertes Konzept

- Der Parser liest den Textbaum (DSL Programm) und generiert einen abstrakten Syntaxbaum (AST), der weiterverarbeitet wird um den Zielcode zu erzeugen.
- Standard-Editoren werden zur Modifizierung des DSL Programms verwendet (Konzept von Zeilen, Cut&Paste, etc.)
- Beispiel: Xtext
  - Eclipse Modeling Framework
  - Kreirt Parser, Klassenmodell aus AST, wertet Texteditor auf (Codevervollständigung, Syntaxhighlighting, etc.)

Projektierungskonzepte

- Der Code ist zu jeder Zeit im AST visuell erhalten
- Verschiedene Editoren verändern direkt den AST (Mix aus grafischen und textbasierten Editoren)
- Es besteht kein Bedarf für Code-parsing. So ist es einfach eine Sprache in eine andere einzubetten
- Beispiel: MPS von JetBrains
  - Beim erstellen einer Sprache werden die Regeln für Code-Editierung und -Rendering, sowie Sprachtypsystem und Einschränkungen definiert
  - MPS generiert Java, XML oder HTML Code

Beschreiben sie den Unterschied zwischen System- und Scripting-Languages.

In der Softwareentwicklung gibt es seit je her eine Dikussion, ob man Systemprogrammiersprachen wie C, C++ oder Java nutzen soll, oder Skriptsprachen wie Python, Perl und Tcl.

Systemprogrammiersprachen wurden entwickelt, um Datenstrukturen und Algorithmen von Grund auf zu bauen, beginnend mit den primitivsten Computerelementen wie Speicherbytes (Words of Memory).

Skriptsprachen sind designed um zu kleben: Sie gehen von der Existenz mächtiger Komponenten aus und sind nur darauf ausgelegt diese zu verbinden.

- Systemprogrammiersprachen wurden als Alternative zu Assemblersprachen vorgestellt
- Systemprogrammiersprachen unterscheiden sich von Assemblersprachen auf zwei Arten. Sie sind auf höherer Ebene und streng typisiert:
    - Der Begriff **"Höhere Ebene"** bedeutet, dass viele Details automatisch verarbeitet werden sodass Programmierer weniger Code schreiben können um die gleiche Arbeit zu verrichten.
    - In einer **streng typisierten** Sprache definiert der Programmierer, wie jede Einzelinformation genutzt wird und die Sprache verhindert, das sie auf andere Art verwendet wird.
- Skriptsprachen gehen davon aus, dass eine Ansammlung brauchbarer Komponenten bereits in anderen Sprachen existieren.
- Skriptsprachen werden manchmal als Klebesprachen oder Systemintegrationssprachen bezeichnet. Sie erlauben schnelle Entwicklung von Klebe-orientierten Applikationen.
- Um das Verbinden mehrerer Komponenten zu erleichtern tendieren Skriptsprachen dazu typenlos zu sein.
- Skriptsprachen sind oft string-orientiert (liefert eine einheitliche Represäntation vieler verschiedener Dinge)
- Skriptsprachen werden gewöhnlich interpretiert, während Systemprogrammiersprachen kompiliert werden.
- Skriptsprachen sind weniger effizient als Systemprogrammiersprachen.
- Ein typisches Statement in einer Skriptsprache führt hunderte oder tausende Maschineninstruktionen aus, während ein typisches Statement einer Systemprogrammiersprache etwa fünf Maschineninstruktionen ausführt.

Eine Skriptsprache ist kein Ersatz für eine Systemprogrammiersprache und umgekehrt.

Welche besonderen Eigenschaften haben funktionale Programmiersprachen?

Funktionale Programmierung ist ein Paradigma (Musterbeispiel), dass Berechnung als Evaluierung mathematischer Funktionen behandelt und statische und variable Daten vermeidet.
In der Praxis ist der Unterschied einer mathematischen Funktion und der Auffassung einer "Funktion", wie sie in der imperativen Programmierung verwendet wird der, dass imperative Funktionen Seiteneffekte haben können, die den Wert des Programmstatus ändern.
Funktionen sind Objekte erster Klasse, was bedeutet, sie können überall benutzt werden, wo Daten verwendet werden (Bsp.: Zuweisungen, Parameter).

In der funktionalen Programmierung wird das Konzept der Abschließung verwendet. Ein Abschluss ist eine Funktion mit einer referenzierten Umgebung für die nicht-lokalen Variablen dieser Funktion. Ein Abschluss erlaubt einer Funktion Zugriff auf Variablen, die ausserhalb ihres Anwendungsbereiches liegen.

Vorteile von funktionalen Sprachen:

- **Parallelisierung** - Ohne Seiteneffekte und unveränderbaren Daten ist Parallelisierung einfacher zu verwalten. (Mehrkernprozessoren, etc.)
- **Mächtig im Ausdruck** - Es ist einfacher (verglichen mit imperativen Sprachen) und eleganter manche Arten von Problemen zu lösen.
- **Gute Unterstützung für Listen und Collections** - Funktionen zum listen von Datenstrukturen ist einfach (Bsp.: Liste sortieren, Funktion auf alle Elemente einer Liste anwenden, Liste filtern, etc.)

Nennen Sie einige der bekanntesten Vertreter von funktionalen Programmiersprachen.

Multi-Paradigma-Sprachen

- Scala, Clojure, Ruby, Haskell, Lisp (weite Verbreitung in AI)
- Smalltalk, Java (anonyme Klassen vs. Abschlüsse)

Was versteht man unter Configuration Managmenet?

In der Softwareentwicklung passieren ständig Veränderungen, daher ist es absolut unerlässlich Veränderungsmanagement zu betreiben.

- **Interaktion zwischen Leuten** - Wenn zwei Leute an einem Komponenten arbeiten müssen ihre Veränderungen koordiniert werden.
- **Neueste Version** - Es muss sichergestellt sein, dass jeder Zugang zur aktuellsten up-to-date Version hat.
- **Rollback oder Tagging** - Wenn mit einer neuen Version etwas schiefgeht, muss man in der Lage sein zu einer alten funktionierenden Version zurückzukehren.

Fundamentale Konfigurationsmanagementsaktivitäten sind:

- **Versionsmanagament** - Verschiedene Versionen von Komponenten im Auge behalten
- **Systemintegration** - Definition welche Versionen von Komponenten für den Entwurf eines Systems benutzt wurden
- **Problemverfolgung** - Benutzern erlauben Bugs mitzuteilen mit Referenz auf die Version

Was sind die besonderen Eigenschaften von Host-Target Development?

Software wird auf einem Computer entwickelt (Host-Development-Platform), läuft aber auf einer separaten Maschine (Target-Execution-Platform).

Manchmal sind Entwicklungs- und Ausführungsplattform die selbe Maschine. Gewöhnlich sind sie jedoch verschieden sodass entwickelte Software auf die Ausführungsplattform verschoben werden muss oder Simulatoren auf der Entwicklungsplattform laufen müssen.

Die **Softwareentwicklungsplattform** soll eine Auswahl an Werkzeugen bereitstellen:

- Integrierten Compiler und syntaxgerichtetes Editiersystem
- Debuggingsystem für die Sprache
- Grafische Editierwerkzeuge (Bsp.: um UML Modelle zu bearbeiten)
- Werkzeuge zum Testen (Bsp.: JUnit)
- Projektunterstützende Werkzeuge

Was versteht man unter Open Source Development?

Open Source Development ist ein Ansatz in der Softwareentwicklung in dem der Source Code des Softwaresystems öffentlich zugänglich ist und Freiwillige eingeladen sind am Entwicklungsprozess teilzunehmen.

Seine Wurzeln liegen in der Free Software Foundation, die dafür eintritt, dass Source Code nicht geheim sein soll, sondern für jeden Benutzer einsichtig, damit jener Veränderungen jeglicher Art vornehmen kann. Open Source Software hat diese Idee erweitert, indem sie das Internet als Rekrutierungsmittel genutzt hat um eine viel größere Anzahl an freiwilligen Entwicklern anzuheuern. Viele davon benutzen den Code auch.

Das bekannteste Open Source Produkt ist das Linux Betriebssystem.

Die meisten erfolgreichen Open Source Produkte waren Plattformprodukte und weniger Applikationssysteme.

Welche Lizenzbedinungen kennen Sie aus dem Bereich "Open Source Licensing", und was sind deren besondere Kennzeichen?

- **GNU General Public License (GPL)** - Wenn GPL Open Source Software benutzt wird, muss sie auch Open Source zugänglich sein.
- **GNU Lesser General Public License (LGPL)** - Komponenten, die zu Open Source Code verlinken, müssen nicht Open Source sein, ausser wenn die Open Source Softwarekomponenten verändert wurden.
- **Berkley Standard Distribution (BSD) License** - Komponenten dürfen ohne Einschränkungen zu Software verlinkt oder verändert werden. Der Originalautor muss eingeräumt werden.

Kapitel 5 - Verification & Validation

Erklären Sie die Begriffe Verification und Validation.

Während und nach der Implementierung muss das Programm darauf geprüft werden, ob es die Spezifikationen und die Funktionalität bietet, welche die Leute, die dafür zahlen erwarten.
Diese Vorgänge nennt man **Verification und Validation (V&V)**:

- **Verification:** Bauen wir das Produkt richtig?
- **Validation:** Bauen wir das richtige Produkt?

Beschreiben Sie die beiden Ansätze zur Verification und Validation von Software-Systemen.

- **Softwareinspektionen, Peer Reviews (Beaufsichtigungsfeedback).**
  Softwareinspektionen analysieren und überprüfen die Systemrepräsentationen, wie das Requirementsdokument, die Designdiagramme und den Source Code. Softwareinspektion ist eine statische V&V Technik.
- **Software testing.**
  Softwaretesting beinhaltet eine Software mit Testdaten laufen zu lassen. Der Output und das Verhalten der Software wird geprüft um zu sehen, ob die Software genau das macht, was vorgesehen ist. Software testing ist eine dynamische V&V Technik.

Erklären Sie die Technik der Software Inspection, ihre Vorteile und die Möglichkeiten der Automatisierung.

Softwareinspektion ist ein statischer V&V Prozess, bei dem eine Software reviewed wird, um Fehler und Anomalien zu finden.
Gewöhnlich fokussiert sich die Inspektion auf den Source Code, aber jede lesbare Repräsentation der Software wie Anforderungen oder Design Modell können inspiziert werden. Man benutzt das Wissen übers System sowie Einsatzdomäne und die Programmiersprache, oder das Designmodell um Fehler zu finden.

Vorteile:

- **Fehlerabhängigkeiten**
  Während des Testens können Fehler andere Fehler verstecken. Eine einzelne Inspektion kann viele Fehler in einem System entdecken.
- **Testen unvollständiger Systeme**
  Unvollständige Versionen von Systemen können ohne zusätzliche Kosten inspiziert werden.
- **Testen von weiteren Qualitätsattributen**
  Eine Inspektion kann auch andere Attribute eines Programmes untersuchen, wie Einhaltung von Standards, Portabilität und Wartbarkeit. Man kann nach Ineffizienz, ungeeigneten Algorithmen und schlechtem Programmierstil ausschau halten, welche das System schwer wartbar machen können.

Automatisierte statische Analyse

Inspektionen werden häufig durch Checklisten, Fehler und Heuristiken getrieben, die gängige Fehler in verschiedenen Programmiersprachen identifizieren. Für manche Fehler und Heuristiken ist es möglich den Prozess der Checklistenüberprüfung zu automatisieren, was in der Entwicklung von automatischen statischen Analyzern für verschiedene Programmiersprachen resultierte. Statische Analyzer sind Softwarewerkzeuge, die den Soruce Code eines Programmes scannen und mögliche Fehler und Anomalien aufdecken.

Die Phasen der statischen Analyse beinhalten:

- **Kontrollflussanalyse**
  Dieser Abschnitt identifiziert und hebt Schleifen mit mehreren Eintritts- und Ausganspunkten sowie unerreichbaren Code hervor. Unerreichbarer Code ist Code, der im Bereich eines Bedingungsstaments liegt, dessen Bedingung nie wahr sein kann.
- **Datengebrauchsanalyse**
  Dieser Abschnitt hebt hervor, wie Variablen im Programm genutzt werden. Es spürt Variablen auf, die ohne vorherige Initialisation benutzt werden und solche, die deklariert sind, aber nie benutzt werden.
- **Interfaceanalyse**
  Diese Analyse kann Konsistenzen von Operationsdeklarationen und deren Verwendung in einer schwach typisierten Sprache wie C überprüfen. Interfaceanalyse kann auch Prozeduren, die deklariert sind und die aufgerufen werden, oder Funktionsresultate, die nie benutzt werden aufspüren.
- **Pfadanalyse**
  Diese Phase der semantischen Analyse identifiziert alle möglichen Pfade durchs Programm und legt alle Statements, die in diesen Pfaden festgelegt werden, fest.

Erklären Sie die Technik des Software Testing, ihre Ziele und die Möglichkeiten der Automatisierung.

Test cases sind Spezifikationen über den zu testenden Input und den zu erwartenden Output des Systems sowie ein Statement darüber, was getestet wird. Testdaten können manchmal automatisch generiert werden. Automatische **Testcase-generierung** ist unmöglich. Ausführung des Tests kann automatisiert werden.Gründliches Testen, bei dem jede mögliche Programmausführung getestet wird, ist unmöglich.

Gründe für frustrierte Entwickler wegen Bugs, die den Tests entfliehen konnten:

- **Der Benutzer führte ungetesteten Code aus** - Wegen Zeitdruck ist es für Entwickler nicht ungewöhnlich ungetesteten Code herauszugeben.
- **Die Reihenfolge in der Statements ausgeführt werden** - Die Reihenfolge der Ausführungen ist im tatsächlichen Gebrauch anders, als im Test.
- **Der Benutzer hat eine Kombination von ungetesteten Eingaben angewandt** - Wegen der vielen Möglichkeiten an Eingabekombinationen müssen Tester schwierige Entscheidungen darüber treffen, welche sie testen und manchmal sind es die Falschen.
- **Die Operationsumgebung des Benutzers wurde nie getestet** - Man wusste vielleicht über die Umgebung bescheid, hatte aber keine Zeit diese zu testen. Die Zusammenstellung der Hardware des Benutzers, etc. wurde nicht nachgebaut.

Erklären Sie den Unterschied zwischen Defect und Validation Testing.

- **Validation Testing**
  Um dem Entwickler und dem Kunden zu demonstrieren, dass die Software die Anforderungen erfüllt.
  Dieses Ziel führt zu **Validationstesten**, bei dem man vom System erwartet, dass es die gegebenen Testcases, *welche die zu erwartende Nutzung des Systems reflektieren*, korrekt ausführt.
  Beim Validation Testing ist der Test dann erfolgreich, wenn das System sich korrekt verhält.
- **Defect Testing**
  Um Fehler oder Defekte zu entdecken, bei denen das Verhalten der Software inkorrekt, unerwünscht oder nicht konform der Spezifikation ist.
  Dieses Ziel führt zu **Defect Testing**, bei denen die Testcases darauf ausgelegt sind, Fehler aufzuzeigen.
  Beim Defect Testing ist ein Test dann erfolgreich, wenn ein Defekt aufgezeigt wird, bei dem sich das System inkorrekt verhält.

In welche drei Phasen kann Software Testing bei kommerzieller Software gegliedert werden und wodurch zeichnen sich diese Phasen aus?

- **Development testing**
  Das System wird während der Entwicklung auf Defekte getestet. Die Tester sind Systemdesigner und Programmierer.
- **Release testing**
  Ein separates Testteam testet eine vollständige Version des Systems, bevor es an Benutzer ausgeliefert wird. Das Ziel ist es zu überprüfen, ob das System die Anforderungen der Stakeholder erfüllt.
- **User testing**
  (Mögliche) Benutzer testen das System in ihrer eigenen Umgebung. Beim Akzeptanztesten testet ein Kunde formal das System um zu entscheiden, ob es vom Systemanbieter akzeptiert werden kann.

Was sind die Unterschiede zwischen Unit Testing, Component (Interface) Testing und System Testing?

- **Unit testing**
  Individuelle Programmeinheiten oder Objektklassen werden getestet. Es fokussiert auf das Testen der Funktionalität von Objekten oder Methoden.
- **Component (interface) testing**
  Mehrere individuelle Einheiten werden zu zusammengesetzten Komponenten integriert. Es fokussiert darauf Komponenteninterfaces (Schnittstellen) zu testen.
- **System testing**
  Manche oder alle Komponenten eines Systems werden integriert und das System wird als ganzes getestet. Es fokussiert auf das Testen von Komponenteninteraktion.

Beschreiben Sie die Unterteilung der unterschiedlichen Arten von Tests.

Dies ist ein Lösungsansatz von mir. Möglicherweise falsch, aber ich wüsste nicht, was da sonst passen könnte. - Ivan

- **Development Testing**
  - Unit Testing
  - Component (Interface) Testing
  - System Testing
- **Release Testing**
  - Black-Box Testing
  - White-Box Testing
- **Performance Testing**
- **User Testing**
  - Alpha Testing
  - Beta Testing
  - Acceptance Testing

Erklären Sie den Begriff Component Testing.

Softwarekomponenten sind häufig zusammengesetzte Komponenten, die aus mehreren interagierenden Objekten aufgebaut sind. Der Zugang zur Funktionalität dieser Objekte ist im Komponenteninterface definiert. Komponententesting fokussiert darauf, das Verhalten der Komponentenschnittstellen laut Spezifikation, zu beweisen.

Schnittstellenfehler in zusammengesetzen Komponenten sind möglicherweise nicht ausfindig zu machen, indem man die einzelnen Objekte testet, da diese Fehler sich erst aus der Interaktion zwischen den Objekten in der Komponenten ergeben.

Schnittstellenfehler sind eine der häufigsten Formen von Fehlern in komplexen Systemen. Diese Fehler unterteilen sich in 3 Klassen:

- **Interfacemissbrauch** - Eine aufrufende Komponenten ruft eine andere Komponenten auf und macht einen Fehler bei der Verwendung der Schnittstelle (Bsp.: Falscher Parametertyp übergeben, etc.)
- **Missverstehen von Interfaces** - Eine aufrufende Komponenten missversteht die Spezifikation der aufgerufenen Komponenten und macht Annahmen über deren Verhalten.
- **Timingfehler** - Diese kommen in Echtzeitsystemen vor, die geteilten Speicher (Shared Memory) oder Message-passing Interfaces nutzen. Der Produzent der Daten und der Konsument können in verschiedenen Geschwindigkeiten arbeiten.

Beschreiben Sie die vier Phasen einer Testsequenz.

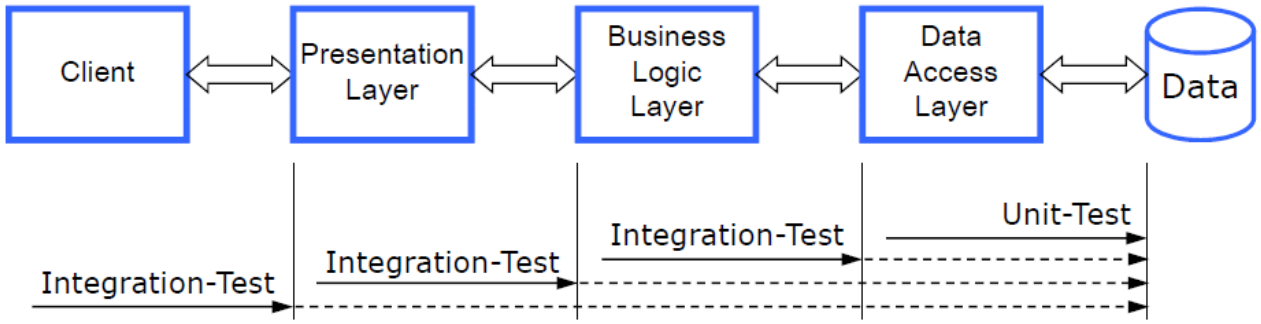
- **Setup** - Wir setzen die Testbefestigung (test fixture) auf, die vom SUT benötigt wird, um das erwartete Verhalten zu beobachten sowie alles, was wir brauchen um die Möglichkeit zu haben, das tatsächliche Ergebnis zu beobachten.
- **Übung (Exercise)** - Wir interagieren mit dem SUT.
- **Verifizieren (Verify)** - Wir tun alles, was notwendig ist um festzulegen, ob das erwartete Ergebnis erhalten wurde.
- **Abriss (Teardown)** - Wir reissen die Testbefestigung ab, um die Welt in den Zustand zurückzubringen, in der wir sie gefunden haben.

Erklären Sie die Begriffe SUT, DOC und Test Double.

SUT - System under Test. So heisst die getestete Komponente.
DOC - Dependent-on Componentent (Eine individuelle Klasse oder eine Large-grained Komponente, von der das SUT abhängt)
Test Double - Die DOC wird mit einer testspezifisch äquivalenten Komponente ersetzt, dem sogenannten Test Double oder Mock Object.

Stellen Sie Integrationstesting und Unit/Component Tests gegenüber (Skizze) und erläutern sie Vor- und Nachteile der beiden Vorgehensweisen.

Nähere Ausführungen erwünscht!



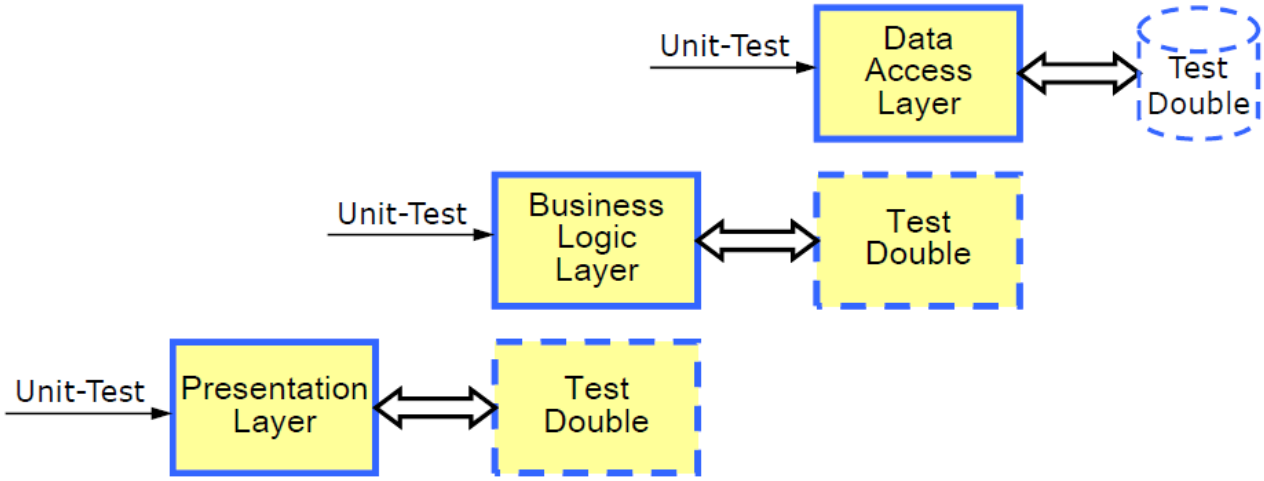
Integrationstesting ist eine Reihe von Einzeltests, die dazu dienen, verschiedene voneinander abhängige Komponenten eines Systems im Zusammenspiel zu testen. Jede einzelne Komponente hat ihren Modultest bereits erfolgreich bestanden.[3]

Vorteile:

- Zusammenspiel der Komponenten wird getestet.

Nachteile:

- Bei größeren Systemen nicht immer durchführbar, da Komponenten voneinander abhängen um getestet werden zu können.



Komponenten, die von anderen abhängig sind, können mit Hilfe von Test Doubles getestet werden.

Vorteile:

- Jede Komponente kann jederzeit getestet werden.

Nachteile:

- Zusammenspiel der Komponenten wird nicht getestet.

[3] *Vgl.: http://de.wikipedia.org/wiki/Integrationstest*

Was versteht man unter „Test-Driven Development (TDD)“? Erläutern Sie die Vorgehensweise bei TDD.

TDD wurde als Teil agiler Methodik vorgestellt und beschreibt einen Ansatz in der Programmentwicklung, bei dem man Testing und Codeentwicklung verschachtelt.
Essentiell entwickelt man den Code inkrementell zum Test für genau dieses Inkrement. Man geht erst zum nächsten Code weiter, wenn der bereits geschriebene seinen Test besteht.

TDD hilft Programmierern ihre Ideen, was ein Codesegment tatsächlich machen soll, zu verdeutlichen. Um einen Test zu schreiben, muss man verstehen, was angestrebt wird, weil dieses Wissen es einfacher macht den benötigten Code zu schreiben. Wenn man kein vollständiges Wissen oder Verständnis hat, wird TDD nicht weiterhelfen.

Vorgangsweise:

- **Identifizieren neuer Funktionalität** - In einigen Zeilen implementieren
- **Test schreiben** - Automatisierter Test. Ergebnisse sollen sofort berichtet werden
- **Test ausführen** - Zusammen mit allen anderen Tests. Anfangs wird er fehlschlagen, weil die Funktionalität noch nicht implementiert ist.
- **Funktion implementieren** - Refaktorieren, Verbessern und Ergänzen von Code bis die Tests bestanden sind.

Vorteile:

- **Codeerfassung** - Jedes geschriebene Codesegment sollte zumindest einen zugehörigen Test haben. Der Code wird getestet, während er geschrieben wird. Somit werden Defekte im frühen Entwicklungsprozess entdeckt.
- **Regressionstesting** - Eine Testfolge wird inkrementell mit dem Programm entwickelt. Man kann immer Regressionstests laufen lassen, um zu überprüfen, ob neue Bugs entstanden sind.
- **Vereinfachtes Debugging** - Wenn ein Test schiefgeht, sollte es offensichtlich sein, wo das Problem liegt. Der neu geschriebene Code muss überprüft und modifiziert werden.
- **Systemdokumentation** - Die Tests selbst stellen eine Form von Dokumentation dar, die beschreibt, was der Code machen sollte. Der Code kann durch das Lesen der Tests leichter verständlich sein.

Welche Arten von User Testing gibt es und wodurch unterscheiden sich diese?

- **Alpha testing** - Benutzer der Software arbeiten mit dem Entwicklerteam zusammen und testen die Software am Standort des Entwicklers.
- **Beta testing** - Eine Ausgabe der Software wird Benutzern zugänglich gemacht, die damit experimentieren und Probleme aufzeigen.
- **Acceptance testing** - Kunden testen das System informal um zu entscheiden, ob es aus den Händen der Systementwickler akzeptiert werden kann. Phasen von Acceptance testing:
    - Akzeptanzkriterium definieren
    - Akzeptanztest planen
    - Herleiten des Akzeptanztests
    - Akzeptanztest ausführen
    - Testresultate diskutieren
    - Ablehnen oder Annehmen des Systems

Was versteht man unter Test Case Design und welche Ansätze verwendet man dabei (Erklärung)?

Das Ziel des **Testcase-Design-Prozesses** ist es, eine Ansammlung von Tests zu erstellen, die Programmdefekte effektiv aufdecken können und zeigen, dass das System seine Anforderungen erfüllt. Um einen Testcase zu designen, wählt man ein Feature des Systems oder eine Komponente zum Testen aus. Danach wählt man eine Anzahl an Inputs, die das Feature ausführt und dokumentieren die zu erwartenden Outputs oder Outputbereiche.

Es gibt verschiedene Ansätze für Testcase Design:

- **Anforderungsbasiertes Testen**
  Die Anforderungen sollten so geschrieben werden, dass Tests erstellt werden können, die einem Beobachter zeigen, dass die Anforderungen erfüllt wurden.
  Anforderungsbasiertes Testen ist ein **systematischer Ansatz** im Testcase Design, bei dem man für jede Anforderung eine Ansammlung an Tests bedenkt. Man sollte Rückverfolgbarkeitsaufzeichnungen der anforderungsbasierten Tests aufbewahren, *die den Test mit der spezifischen Anforderung*, die getestet wird, verbinden.
  Anforderungsbasiertes Testen ist eher **Validation Testing** als Defect Testing, da man versucht zu demonstrieren, dass die Anforderungen im System ordentlich implementiert wurden.
- **Richtliniengesteuertes Testen**
  Hier werden Richtlinien zu Rate gezogen, um Testcases auszuwählen.
  Diese Richtlinien reflektieren Erfahrungen mit den Arten von Fehlern, die Programmierer oft machen, wenn sie Komponenten entwerfen.
  Allgemeine Richtlinien:
    - Wählen sie Inputs aus, die das System zwingen alle Fehlermeldungen auszugeben
    - Entwerfen sie Inputs, die den Inputbuffer zum Überlaufen bringen
    - Wiederholen sie den gleichen Input oder die Reihe an Inputs mehrmals
    - Erzwingen sie das erzeugen von ungültigen Outputs
    - Erzwingen sie, dass Rechenergebnisse zu groß oder zu klein sind
- **Teiltesten (Partition Testing)**
  Die Input-Daten und Output-Ergebnisse eines Programms fallen oft in eine einzelne von vielen verschiedenen Klassen (**äquivalente Partitionen**), die einheitliche Charakteristiken

haben wie z.b.: positive Zahlen, negative Zahlen, etc.
Ein semantischer Ansatz zu Testcase-Design basiert auf der Identifikation aller Partitionen einer Systemkomponente. Sobald eine Ansammlung an Partitionen identifiziert wurde, kann man Testcases aus jeder Partition auswählen.
Eine gute Daumenregel für Testcaseauswahl ist es, Grenzwerte der Partitionen und Werte, die nahe am mittigsten Wert liegen, zu wählen.

- **Strukturelles Testen**
  Dieser Ansatz beschreibt die Herleitung der Testcases aus dem Wissen über die Struktur der Software und deren Implementation. Das Verständnis des Algorithmus' einer Komponente kann bei der Identifizierung weiterer Partitionen und Testcases weiterhelfen. Diesen Ansatz nennt man manchmal **white-box testing.**
- **Pfadtesten**
  Ist eine strukturelle Teststrategie, deren Zweck es ist, jeden individuellen Ausführungspfad einer Komponente oder eines Programms in Anspruch zu nehmen. Weiters werden alle Bedinungsstatements auf wahr und falsch getestet.
  Pfadtesttechniken werden am meisten während Komponententesting verwendet.

Kapitel 6 - Software Evolution

Erklären Sie das Grundkonzept der Software Evolution.

Softwareentwicklung kommt nicht nach der Auslieferung eines Systems zum Stillstand, sondern **setzt sich durch die ganze Lebenszeit des Systems durch**. Sobald sich Software in Gebrauch befindet, kommen neue Anforderungen hinzu und bereits existierende Anforderungen ändern sich. Teile der Software müssen möglicherweise modifiziert werden, um Fehler zu korrigieren, mit einer neuer Plattform kompatibel zu sein und die Performance zu steigern, oder andere nicht-funktionale Anforderungen zu erfüllen.

Beschreiben Sie die unterschiedlichen Arten der Software Maintenance.

- **Maintenance um Softwaremängel zu reparieren**
  - Codierungsfehler sind für Gewöhnlich sehr billig zu beheben
  - Designfehler sind teurer, da möglicherweise einige Komponenten umgeschrieben werden müssen
  - Anforderungsfehler sind durch das möglicherweise sehr umfangreiche Systemredesign am teuersten zu beheben.
- **Maintenance um Software an andere Operationsumgebungen zu adaptieren**
  - Diese Art von Wartung ist notwendig, wenn sich Aspekte der Systemumgebung wie Hardware, Plattformbetriebssystem oder Unterstützungssoftware verändern. Das Applikationssystem muss so modifiziert werden, dass es mit diesen Umgebungsveränderungen umgehen kann.
- **Maintenance um Systemfunktionalitäten hinzuzufügen oder zu modifizieren**
  - Diese Art von Wartung ist notwendig, wenn sich die Systemanforderungen aufgrund von organisationellen oder geschäftlichen Veränderungen verändern. Der Maßstab der Veränderungen der Software ist oft viel größer, als für andere Wartungstypen.

Stellen Sie die Kosten für die Software Entwicklung und Wartung gegenüber (Erklärung).

Der wichtigste Punkt ist jener, dass Systemfehler zu beheben nicht die teuerste Wartungsaktivität ist. Das System weiter zu entwickeln, dass es mit neuen Umgebungen oder veränderten Anforderungen zurecht kommt, konsumiert wesentlich Wartungsaufwand.

Die Wartungskosten bestehen aus:

- 17% Systemfehlerbehebung
- 18% Softwareanpassung und -adaption
- 65% Hinzufügen oder Veränderungen von Funktionalitäten

Wartungskosten als Teil von Entwicklungskosten variieren von einer Applikationsdomäne zur anderen:

- Die Wartungskosten für Business-Applikationssysteme sind weitläufig mit Systementwicklungskosten vergleichbar.
- Für eingebettete Real-Time-Systeme können Wartungskosten das bis zu vierfache der Entwicklungskosten ausmachen.

Daher senkt Arbeit, die während der Entwicklung gemacht wird, *um die Software leichter verständlich und veränderbar zu machen*, vorraussichtlich die Wartungskosten.

Erklären Sie die Unterschiede zwischen Software Development und Maintenance.

- **Teamstabilität**
  Nachdem ein System abgeliefert wurde, ist es normal für ein Team, dass es sich auflöst und die Leute an anderen Projekten arbeiten. Das neue Team, oder die Individuen, die für die Wartung des Systems verantwortlich sind, verstehen das System, oder den Hintergrund für Systementscheidungen nicht. Es ist sehr viel Aufwand notwendig, um das existierende System zu verstehen, bevor man Veränderungen implementieren kann.
- **Vertragsverantwortung**
  Der Vertrag zur Wartung des Systems ist gewöhnlich ein anderer, als der Entwicklungsvertrag. Der Wartungsvertrag wird möglicherweise einer anderen Firma übergeben als der Entwicklerfirma. Das bedeutet, dass es für das Entwicklerteam keinen Anreiz gibt, die Software so zu schreiben, dass sie einfach zu ändern ist.
- **Mitarbeiterfähigkeiten**
  Wartungsmitarbeiter sind oft relativ unerfahren und unvertraut mit der Applikationsdomäne. Wartung hat ein schlechtes Image unter Softwareingenieuren. Weiters können alte Systeme in veralteten Programmiersprachen geschrieben sein.
- **Programmalter und Struktur**
  - Mit zunehmendem Alter eines Programms tendiert dessen Struktur zu degradieren sodass es schwerer zu verstehen und modifizieren ist.
  - Manche Systeme wurden ohne moderne Software-Engineering-Techniken entwickelt.
  - Sie wurden möglicherweise nie gut strukturiert und eher für Effizienz optimiert als für Verständlichkeit.
  - Systemdokumentation ging möglicherweise verloren oder ist inkonsistent.
  - Alte Systeme waren vielleicht nie Thema von Konfigurationsmanagement, was dazu führt, dass viel Zeit verschwendet wird um die richtige Version der zu verändernden System zu finden.

Erklären Sie den Begriff System Re-Engineering, was sind die Vorteile im Vergleich zu einer Neuimplementierung.

Software Re-Engineering beschäftigt sich mit der Re-Implementierung von Legacy-Systemen um diese besser Wartbar zu machen. Es hat zwei große Vorteile zur Neuimplementierung:

- **Reduziertes Risiko**
  Es steckt sehr hohes Risiko im Neu-Entwickeln von Business-kritischer Software.
  Es können Fehler in der Systemspezifikation gemacht werden, oder es kann Entwicklungsprobleme geben. Verzögerungen die Software einzuführen können zu Geschäftsverlust und Extrakosten führen.
- **Reduzierte Kosten**
  Die Kosten von Re-Engineering sind signifikant geringer als die Entwicklungskosten für neue Software.

Beschreiben Sie die Aktivitäten beim Software Re-Engineering.

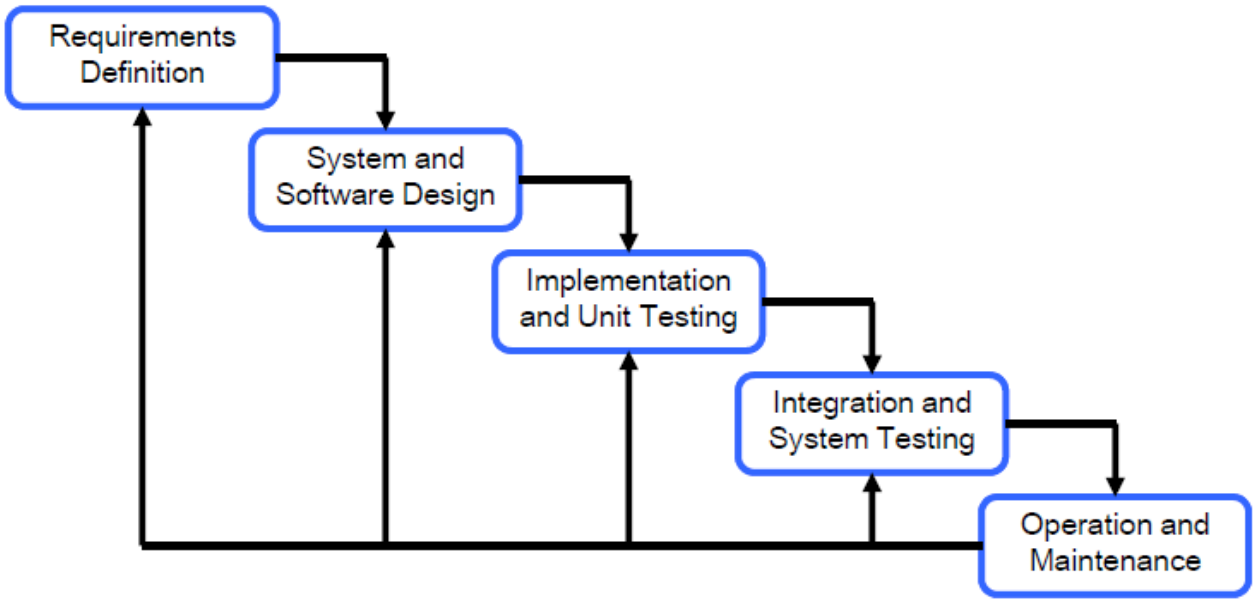
- **Source Code Übersetzung**
  Das Programm wird aus einer alten Programmiersprache zu einer modernen Version der selben Sprache oder einer anderen Sprache konvertiert.
- **Reverse Engineering**
  Das Programm wird analysiert und es werden Informationen daraus extrahiert. Das ist bei der Dokumentation dessen Organisation und Funktionalität behilflich.
- **Verbesserung der Programmstruktur**
  Die Struktur des Programms wird analysiert und modifiziert, um es leichter lesbar und verständlich zu machen.
- **Modularisierung des Programms**
  Zusammengehörige Teile des Programms werden zusammengruppiert und Redundanzen werden entfernt, wo es zweckmäßig ist.
- **Daten Re-Engineering**
  Die Daten, die vom Programm bearbeitet werden, werden verändert, um die Programmänderungen zu reflektieren.

Beschreiben Sie die unterschiedlichen Strategien bei der Legacy System Evolution.

- **Wegwerfen des kompletten Systems**
  Wenn das System keinen tatsächlichen Beitrag zum Geschäftsprozess liefert.
- **System unverändert lassen und mit der Wartung weitermachen**
  Wenn das System immer noch benötigt wird und einigermaßen stabil ist und die Benutzer relative wenig Änderungsanfragen machen.
- **Das System re-engineeren**
  Wenn die Systemqualität durch regelmäßige Veränderungen verschlechtert wurde und weitere Änderungen am System noch notwendig sind.
- **Das System teilweise oder vollständig durch ein neues ersetzen**
  Wenn andere Faktoren wie neue Hardware bedeuten, dass das alte System nicht weiteroperieren kann, oder wenn fertige Systeme die Entwicklung des neuen Systems zu akzeptablen Kosten erlauben.

Kapitel 7 - Software Process Models

Beschreiben Sie das Waterfall Model (Skizze und Erklärung) sowie die Vor- und Nachteile.



Das erste Modell des Softwareentwicklungsprozesses ist von allgemeineren Systemengineeringprozessen abgeleitet. Wegen des Stufenförmigen Aufbaus ist dieses Modell als Wasserfallmodell oder Software-Lify-Cycle benannt.

Seine fundamentalen Entwicklungsaktivitäten sind:

- **Anforderungsanalyse und Definition**
  Die Dienste, Einschränkungen und Ziele des Systems werden mit durch konsultieren der Systembenutzer festgelegt. Sie werden im Detail definiert und dienen als Systemspezifikation.
- **System- und Softwaredesign**
  Der Systemdesignprozess unterteilt die Anforderungen in Hardware- oder Software-Systeme. Es baut eine allumfassende Systemarchitektur auf.
  Softwaredesign bezieht die Identifikation und Beschreibung der fundamentalen Softwareabstraktionen und deren Beziehungen mit ein.
- **Implementation und Unit Testing**
  In dieser Phase wird das Softwaredesign als Ansammlung von Programmeinheiten realisiert. Unit Testing bedeutet zu verifizieren, dass jede Einheit ihre Spezifikation erfüllt.
- **Integration und System Testing**
  Die einzelnen Programmeinheiten oder Programme werden als komplettes System integriert und getestet, um sicherzustellen, dass die Softwareanforderungen erfüllt wurden. Nach dem Testen wird die Software an den Kunden ausgeliefert.
- **Operation und Wartung**
  Das System wird installiert und in praktischen Betrieb genommen. Wartung beinhaltet die Korrektur von Fehlern, die in den vorherigen Phasen des Lebenszyklus nicht entdeckt wurden, das Verbessern der Implementation von Systemeinheiten und Erweitern der

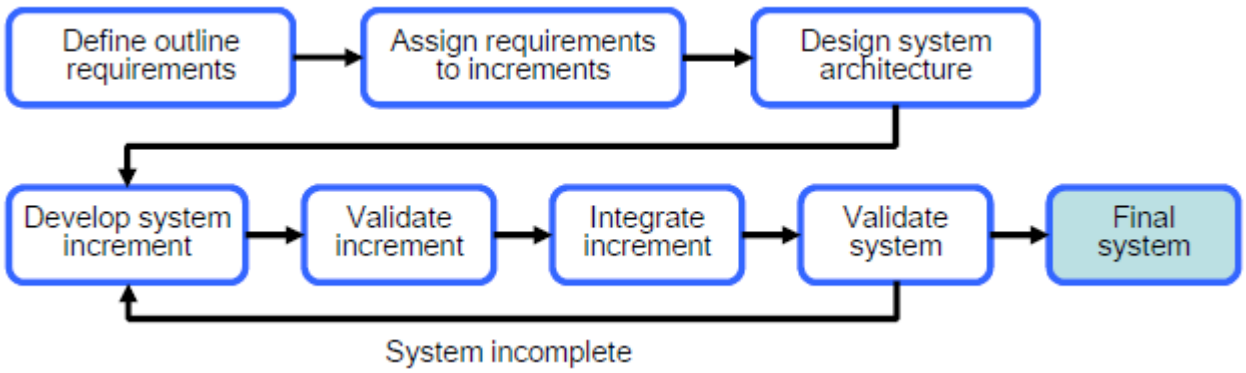
Systemdienste, wenn neue Anforderungen entdeckt werden.

Im Prinzip ist das Resultat jeder Phase eines oder mehrere Dokumente, die abgesegnet werden (signed off). Die nächste Phase sollte nicht beginnen, ehe die vorherige Phase abgeschlossen wurde. In der Praxis überlappen sich diese Phasen und versorgen sich gegenseitig mit Informationen.
Aufgrund der Produktions- und Absegnungskosten dieser Dokumente sind Iterationen sehr teuer und beinhalten signifikante Nacharbearbeitung. Daher ist es nach einer geringen Anzahl an Iterationen normal Teile der Entwicklung, wie die Spezifikationen, einzufrieren und mit den späteren Entwicklungsphasen weiterzumachen.

Die **Vorteile** des Wasserfallmodells sind, dass nach jeder Entwicklungsphase Dokumentationen produziert werden, die in andere Engineering-Modelle passen.
Das **große Problem** ist, dass die Aufteilung des Projektes in eindeutige Phasen unflexibel ist.

Das Wasserfallmodell sollte nur dann verwendet werden, wenn die Anforderungen sehr gut verstanden wurden und die Wahrscheinlichkeit großer Veränderungen dieser während der Systementwicklung sehr gering ist.

Beschreiben Sie das Incremental Delivery Model (Skizze und Erklärung) sowie die Vor- und Nachteile.



In einem inkrementellen Entwicklungsprozess identifizieren die Kunden in Umrissen die Dienste, die das System bieten soll. Danach wird eine Anzahl an Inkrementierungen definiert, wobei jedes Inkrement eine Teilmenge der Systemfunktionalität zur Verfügung stellt. Während der Entwicklung kann weitere Anforderungsanalyse für spätere Inkremente erfolgen, aber Änderung von Anforderungen für das aktuelle Inkrement werden nicht akzeptiert. Sobald ein Inkrement fertiggestellt ist und geliefert wird, können Kunden es in Einsatz nehmen.

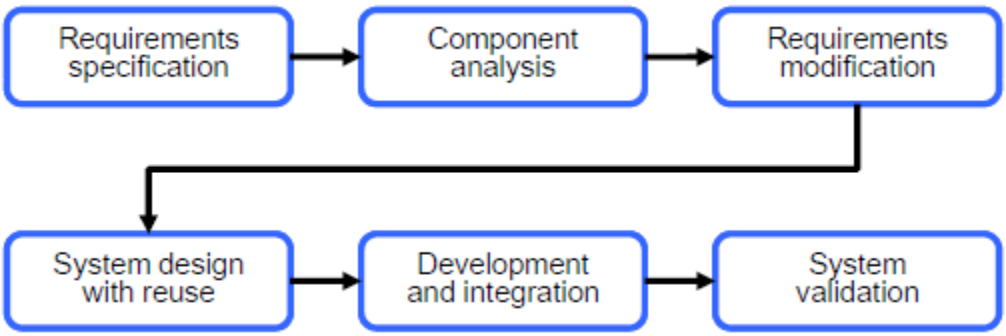
Vorteile:

- Kunden müssen nicht darauf warten, dass das ganze System geliefert wird bevor sie Wert daraus ziehen können.
- Kunden können die frühen Inkremente als Prototypen einsetzen und Erfahrung gewinnen, welche die Anforderungen für spätere Inkremente beeinflusst.
- Es herrscht ein geringeres Risiko, dass das gesamte Projekt scheitert.
- Da die Dienste mit höchster Priorität als erste geliefert werden und spätere Inkremente in diese integriert werden, ist es unvermeidlich, dass diese wichtigsten Systemdienste am meisten getestet werden.

Nachteile:

- **Inkremente sollten relativ klein sein** und jedes Inkrement soll irgendeine Funktionalität liefern. Es kann schwierig sein, die Anforderungen des Kunden in Inkremente der richtigen Größe zu planen.
- Die meisten Systeme benötigen eine Ansammlung von grundlegenden Einrichtungen, die von verschiedenen Teilen des Systems benutzt werden. Da Anforderungen nicht im Detail definiert werden bis ein Inkrement implementiert werden soll, kann es **schwer sein gemeinsame Einrichtungen** zu identifizieren, die von allen Inkrementen benötigt werden.

Beschreiben Sie das Component-Based Development Model (Skizze und Erklärung) sowie die Vor- und Nachteile.



In den meisten Softwareprojekten gibt es Softwarewiederverwendung. Dieser wiederverwertungsorientierte Ansatz stützt sich auf eine große Basis aus wiederverwertbaren Softwarekomponenten und einige integrierte Programmiergerüste für diese.

Die initialen und die Validationsphasen sind vergleichbar mit anderen Prozessen, aber die dazwischenliegenden Phasen unterscheiden sich:

- **Komponentenanalyse**
  Angesichts der Anforderungsspezifikationen wird eine Suche nach Komponenten unternommen, die diese Spezifikationen implementieren. Gewöhnlich gibt es keine perfekte Übereinstimmung und die Komponenten, die möglicherweise genutzt werden, bieten nur einige der benötigten Funktionalitäten.
- **Anforderungsmodifikation**
  Die Anforderungen werden unter Verwendung der Komponenten, die entdeckt wurden, analysiert. Sie werden dann modifiziert, um die vorhandenen Komponenten zu reflektieren.
- **Systemdesign mit Wiederverwertbarkeit**
  Das Programmiergerüst (Framework) wird entworfen oder ein bereits vorhandenes wird benutzt. Unter Umständen muss neue Software entworfen werden, wenn keine wiederverwertbaren Komponenten verfügbar sind.
- **Entwicklung und Integration**
  Bestehende und neu entwickelte Komponenten werden integriert um das neue System zu erstellen. In diesem Modell ist Systemintegration ein Teil des Entwicklungsprozesses und weniger eine separate Aktivität.

Vorteile:

- Die Menge an Software, die geliefert wird, wird reduziert. Das reduziert Kosten und Risiken.
- Gewöhnlich führt das zu schnellerer Entwicklung der Software.

Nachteile:

- Anforderungskompromisse sind unvermeidbar und das führt möglicherweise zu einem System, das die Bedürfnisse des Benutzers nicht befriedigt.
- Ein Teil der Kontrolle über die Systemevolution geht verloren, da neue Versionen der wiederverwertbaren Komponenten nicht unter der Kontrolle der Organisation stehen, die sie benutzt.

Erklären Sie das Manifest für agile Softwareentwicklung und geben Sie zwei Beispiele für agile Vorgehensmodelle an.

Das Manifest setzt sich aus vier Grundsätzen zusammen.

- **Individuen und Interaktion** vor Prozess und Werkzeugen
  - Menschen sind die wichtigste Zutat für Erfolg.
  - Ein Team aus durchschnittlichen Programmierern, die gut untereinander kommunizieren ist erfolgsversprechender als eine Gruppe aus Superstars, die nicht als Team arbeiten können.
  - Ein Team aufzubauen ist wichtiger, als eine Umgebung aufzubauen.
- **Funktionierende Software** vor umfangreicher Dokumentation
  - Es ist für ein Team immer eine gute Idee ein Begründungs- und Strukturdokument zu schreiben, aber dieses muss kurz sein.
  - Zuviel Dokumentation ist schlechter als zu wenig.
  - Kein Dokument produzieren, ausser es ist unmittelbar und ausschlaggebend.
- **Zusammenarbeit mit Kunden** vor Vertragsverhandlungen
  - Erfolgreiche Projekte benötigen Kundenfeedback auf regelmäßiger Basis.
  - Der Kunde arbeitet eng mit dem Team zusammen, anstatt nur vom Vertrag abhängig zu sein.
- **Auf Veränderungen reagieren** vor Planbefolgung
  - Die Businessumgebung kann sich leicht verändern, was die Anforderungen an die Software ändert.
  - Kunden ändern die Anforderungen manchmal ab nachdem sie das System in Funktion sehen.

Beispiele

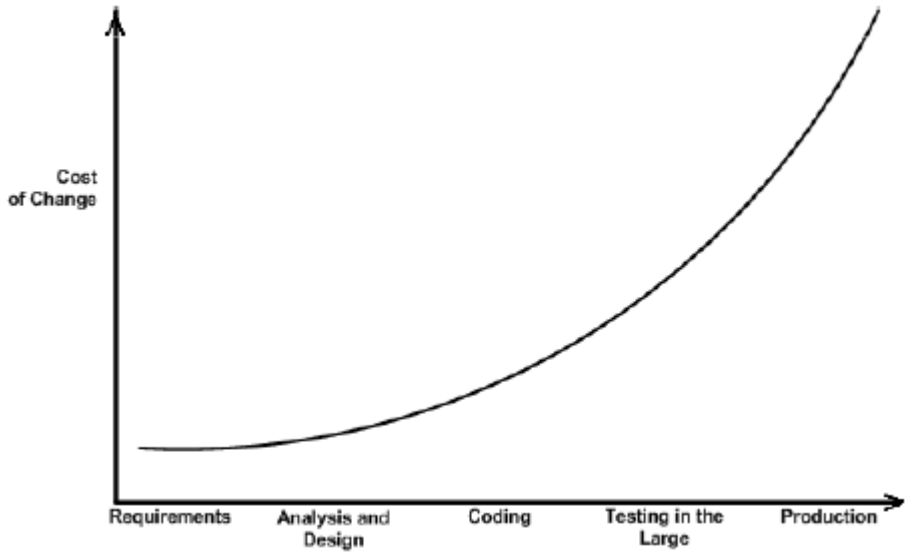
- **Extreme Programming**
  Viele der XP-Praktiken sind alte, bewährte und geprüfte Techniken. XP belebt diese wieder und webt sie zu einem zusammenwirkenden Ganzen zusammen, bei dem jede durch die anderen verstärkt wird und durch die Werte Zweck verliehen wird.
- **Scrum**
  Scrum konzentriert sich auf die Managementaspekte der Softwareentwicklung und teilt diese in 30-Tage-Iterationen (sprints) auf. Es wendet auch engere Überwachung und Kontrolle durch Daily-Scrum-Meetings an.

Erklären Sie die Bedeutung der vier Variablen in Projektmanagement. Welche Variable wird in XP aktiv beeinflusst (warum)?

- **Kosten**
  Kosten sind auf viele Arten die Variable, die am meisten einschränkt.
- **Zeit**
  Die Zeitvariable ist oft nicht in der Hand der Projektmanager aber in der Hand des Kunden.
- **Qualität**
  Jeder will gute Arbeit leisten und Leute arbeiten viel besser, wenn ihnen das Gefühl gegeben wird, dass sie gute Arbeit leisten.
- **Umfang**
  Dies ist die wichtigste Variable in der Softwareentwicklung. Weder die Programmierer, noch die Geschäftsleute haben mehr als eine vage Ahnung darüber, was an der entwickelten Software wertvoll ist. Wenn man den Umfang aktiv managed, kann man Manager und Kunden mit Kosten-, Qualitäts- und Zeitkontrolle versorgen.

In XP wird die Umfangsvariable (Scope) aktiv beeinflusst indem man so eine gewisse Kontrolle über Kosten, Qualität und Zeit hat.

Wie kann die Kurve „Cost of Change“ flach gehalten werden (Skizze und Erklärung).

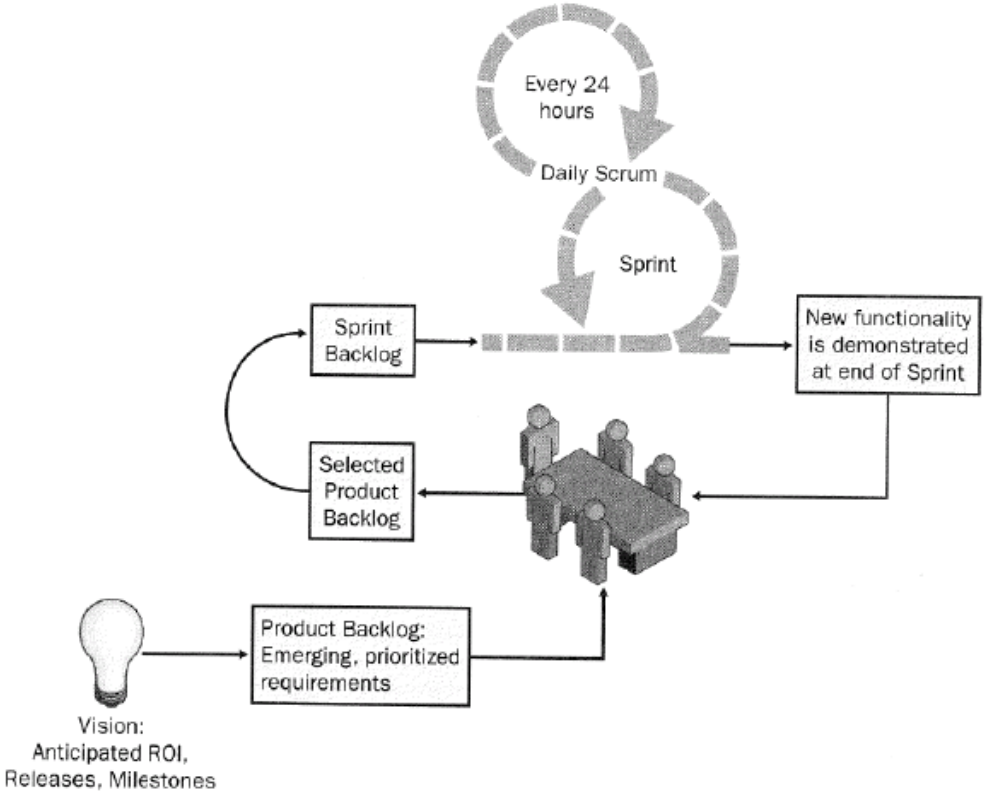


- **Simples Design** ohne extra Designelemente - Keine Ideen, die nicht sofort genutzt werden, aber später genutzt werden sollen.
- **Automatisierte Tests** sodass wir zuversichtlich wissen, ob wir das existierende Verhalten des Systems verändert haben.
- **Viel Übung in der Modifikation des Designs** sodass wir keine Angst davor haben, wenn die Zeit kommt das System zu ändern.

Beschreiben Sie die unterschiedlichen XP Practices.

- **Das Planungsspiel**
  Indem man geschäftliche Prioritäten und technische Schätzungen kombiniert, bestimmt man den Umfang des nächsten Releases schnell. Sobald die Realität den Plan überholt, wird der Plan upgedatet.
- **Kleine Releases**
  Ein simples System wird schnellstens in Produktion gebracht und neue Versionen werden in kurzen Abständen herausgegeben.
- **Metaphern**
  Die ganze Entwicklung wird mit simplen verteilten Geschichten darüber, wie das System funktioniert, geleitet.
- **Simples Design**
  Das System soll zu jedem Zeitpunkt so einfach wie möglich designt sein. Zusätzliche Komplexität wird entfernt.
- **Testing**
  Programmierer schreiben stetig Unit Tests, die fehlerfrei ablaufen müssen, damit die Entwicklung weitergeht. Kunden schreiben Tests, die demonstrieren, dass Features fertig sind.
- **Reafctoring**
  Programmierer restrukturieren das System ohne dessen Verhalten zu ändern. Ziel ist es Duplikationen zu entfernen, Kommunikation zu verbessern, zu vereinfachen oder Flexibilität hinzuzufügen.
- **Paarprogrammierung**
  Der gesamte Produktionscode wird von je zwei Programmierern an einem Computer geschrieben.
- **Kollektiver Besitz**
  Jeder kann jeden Code im System zu jeder Zeit ändern.
- **Kontinuierliche Integration**
  Das System wird, jedes Mal wenn eine Aufgabe beendet ist, mehrmals am Tag gebaut (build).
- **40-Stunden Woche**
  Es wird nicht mehr als 40 Stunden die Woche gearbeitet. Überstunden werden in der Folgewoche nicht mehr gemacht.
- **Kunde im Entwicklungsteam**
  Es wird ein echter, aktiver Benutzer ins Team integriert, der vollzeitverfügbar ist um Fragen zu beantworten.
- **Codingstandard**
  Programmierer schreiben den gesamten Code nach Regeln, die Kommunikation im Code fördern.

Beschreiben Sie den Scrum Flow (Skizze und Erklärung).



- Ein Scrum-Projekt beginnt mit einer Vision des Systems, dass entwickelt werden soll.
- Der PO formuliert einen Plan um die Vision umzusetzen: das **Product Backlog** (Liste von funktionalen und nicht-funktionalen Anforderungen). Dieses wird priorisiert.
- Die ganze Arbeit wird in **Sprints** erledigt. Jeder Sprint ist eine Iteration aus 30 Kalendertagen.
- Jeder Sprint wird mit einem **Sprint-Planungs-Meeting** begonnen, bei dem der PO und das Team sich zusammensetzen und gemeinsam Ziele des nächsten Sprints ausarbeiten. Sprint-Planungs-Meetings können nicht länger als **8 Stunden** dauern.
- Weil das Team selbst dafür verantwortlich ist die eigene Arbeit in Aufgaben einzuteilen, braucht es einen Plan um den Sprint zu beginnen - das **Sprint Backlog**.
- Jeden Tag trifft sich das Team zu einem **15 Minütigen Daily Scrum Meeting**, bei dem jedes Teammitglied folgende Fragen beantwortet:
    - Was hast seit dem letzten Meeting am Projekt gemacht?
    - Was hast du zwischen jetzt um dem nächsten Meeting vor zu machen?
    - Welche Probleme hindern dich daran deine Verpflichtungen gegenüber dem Sprint und diesem Projekt einzuhalten?
- Am Ende des Sprints wird ein **Sprint-Review-Meeting** abgehalten. Dies ist ein **4-Stündiges** Meeting, bei dem das Team dem PO und den Stakeholdern präsentiert, was während des Sprints entwickelt wurde.
- Nach dem Sprint-Review-Meeting und vor dem nächsten Sprint-Planungs-Meeting hält der SM ein **Sprint-Retrospective-Meeting (ca 3 Stunden)** mit dem Team. Der SM ermutigt das Team seinen Entwicklungsprozess zu verbessern um ihn effektiver und erfreulicher für den nächsten Sprint zu machen.

Beschreiben Sie die unterschiedlichen Scrum Rollen (Aufgaben und

Tätigkeiten).

- **Product Owner**
  - Der PO ist dafür verantwortlich die Interessen von jedem, der am Projekt und dem resultierenden System beteiligt ist, zu repräsentieren.
  - Der PO erstellt die gesamten initialen Anforderungen, das Return on Investment (ROI)-Ziel und die Releasepläne.
  - Die Liste der Anforderungen nennt man **Product Backlog**.
  - Der PO ist dafür verantwortlich das Product Backlog zu benutzen um sicherzustellen, dass die wertvollste Funktionalität zuerst produziert wird (durch regelmäßige Priorisierung des Product Backlogs).
- **Team**
  - Das Team ist dafür verantwortlich die Funktionalitäten zu entwickeln.
  - Teams managen und organisieren sich selbst und sind Cross-Functional.
  - Teams sind dafür verantwortlich herauszufinden, wie man das Product Backlog in einer Iteration in eine Funktionalitätssteigerung umwandelt.
  - Alle Teammitglieder sind gemeinsam für den Erfolg jeder Iteration verantwortlich.
- **Scrum Master**
  - Der SM ist für den Scrumprozess verantwortlich.
  - Der SM ist dafür zuständig, jedem, der am Projekt beteiligt ist, Scrum zu erklären.
  - Der SM ist dafür verantwortlich Scrum so in die Unternehmenskultur zu integrieren, dass es passt und trotzdem die erwarteten Vorteile bringt und dass jeder die Scrum-Regeln und Praktiken ausübt.

Beschreiben Sie die unterschiedlichen Scrum Meetings (wann finden sie statt und welchen Zweck haben sie).

- **Sprint-Planungstreffen**
  Es dient der Planung der Arbeiten für den anstehenden Sprint. Der Product Owner erklärt dem Team das Sprintziel. Dabei werden auch Prioritäten gesetzt sowie die Kriterien für die Abnahme der Anforderungen am Ende des Sprints.
  Das Team zergliedert die Anforderungen in die erforderlichen Aktivitäten und schätzt ab ob die Anforderungen überhaupt im Zeitraum fertig gestellt werden können.

- **Daily Scrum**
  Es ist ein Treffen von max. 15 Minuten. Es dient zur Selbstorganisation des Teams und findet jeden Tag zur gleichen Zeit und am gleichen Ort statt.
  Folgende Fragen werden von jedem Mitglied beantwortet:
  a. Was habe ich seit dem letzten Daily Scrum erledigt?
  b. Was nehme ich mir bis zum nächsten Daily Scrum vor?
  c. Welche Herausforderungen sind aufgetreten?

- **Sprint-Review**
  Findet am letzten Tag des Sprints statt. Das Team stellt seine Arbeitsergebnisse dem Product Owner zur Abgabe vor. Es kann nur „abgegeben“ werden, was vorständig und fehlerfrei umsetzt wurde. Beim Sprint-Review mit seinem Check-Charakter wird in besonderem Maße sichtbar, was bereits erarbeitet wurde und wo möglicherweise noch Anpassungsbedarf besteht.

- **Sprint-Retrospektive**
  Findet am Ende des Sprint-Zyklus statt. In der Sprint-Retrospektive wird die Zusammenarbeit der Beteiligten nun offen und ehrlich reflektiert. Das Team spricht Probleme konkret an, lastet sie aber einzelnen Mitarbeitern nicht an.
  Vergleichbar ist die Sprint-Retrospektive mit einer Einsatz-Nachbesprechung bei der Feuerwehr oder einem Debriefing beim Militär. Wie das Review-Meeting hat auch die Retrospektive vor allem einen Check-Charakter, hier ist der Fokus jedoch nicht auf das Produkt, sondern auf den Prozess und die Zusammenarbeit gerichtet. Die Erkenntnisse aus dem Sprint-Review und der Retrospektive gilt es dann zu nutzen, um Verbesserungsmaßnahmen einzuleiten.

Geben Sie die wesentlichen Eigenschaften von Kanban an.

- **Produktionskette**
  Die gesamte Produktionsketten wird mit allen Schritten (Anforderungsanalyse, Implementation, Dokumentation, etc.) für alle Teilnehmer visualisiert.
- **Kanban Board**
  Alle Prozessschritte werden in Spalten geschrieben. Alle Aufgaben stehen auf Karten (Tickets), die sich über die Zeit hinweg von Spalte zu Spalte bewegen.
- **Work in Progress (WiP)**
  Die maximale Anzahl an Tickets, die an einer Station verarbeitet werden dürfen ist begrenzt. Wenn beispielsweise Implementation auf zwei Tickets begrenzt ist, darf Implementation kein weiteres Ticket annehmen, auch wenn Anforderungsanalyse ein Ticket bereitstellen kann.
- **Pull-System**
  Aufgrund der Limitation der Tickets pro Station entsteht ein Zugsystem. Eine Station bekommt von der anderen Arbeit anstatt seine Arbeit zur nächsten Station weiterzuleiten.

Nennen Sie einige Unterschiede zwischen Kanban und Scrum.

Kanban	**Scrum**
Iterationen sind optional	Iteration mit gleicher Länge sind verpflichtend
Priorisierungen sind optional	Alle Einträge des Backlogs müssen priorisiert werden
Das Kanban Board wird dauerhaft aufrechterhalten	Das Scrum Board wird am ende des Sprints gelöscht
Das Kanban Board kann von mehreren Teams mitbenutz werden	Das Scrum Board gehört nur einem Team
Es gibt keine speizifischen Rollen	Rollen (PO, Team, Scurm Master) sind definiert
Neue Anforderungen können jederzeit hinzugefügt werden (wenn die Kapazität vorhanden ist)	Keine neuen Anforderungen, solange der Sprint läuft
Schätzungen sind optional	Schätzungen sind verpflichtend
Kein Diagramm notwendig	Burn-Down-Chart muss geführt werden
Keine Einschränkungen der größe der Anforderungen	Anforderungsgrößen müssen in eine Iteration passen

Kapitel 8 - Unified Modeling Language

Was ist UML?

Die Unified Modeling Language ist eine Familie grafischer Notationen, die von einem einzelnen Meta-Modell gestützt werden, welche in der Beschreibung und dem Design von Softwaresystemen dienen.

- UML ist ein offener Standard, der von der **Object Management Group (OMG)** kontrolliert wird.
- Programmiersprachen sind auf keinem **Abstraktionslevel, das hoch genug ist** um Diskussionen über das Design zu ermöglichen.

Wie werden die Diagramm-Typen gegliedert?

UML2 beschreibt 13 Diagrammtypen, die in folgender Art und Weise klassifiziert werden können:

- **Strukturdiagramme**
  - Klassendiagramm
  - Objektdiagramm
  - Komponentendiagramm
  - Zusammengesetzes Strukturdiagramm
  - Einsatzdiagramm (Deployment)
  - Packagediagramm
- **Verhaltensdiagramme**
  - Aktivitätsdiagramm
  - Anwendungsfalldiagramm (Use Case)
  - Zustandsdiagramm
  - Interaktionsdiagramm
    - Sequenzdiagramm
    - Kommunikationsdiagramm
    - Interaktions-Übersichts-Diagramm
    - Timingdiagramm

Nennen Sie 5 der wichtigsten UML-Diagramm-Typen.

Mein persönlicher Lösungsansatz. Bitte ausbessern, wenn falsch.

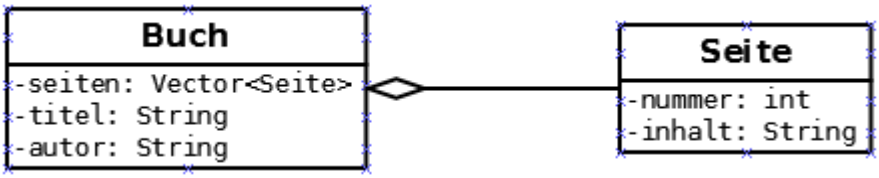
- Klassendiagramm (**Class** diagram)
- Squenzdiagramm (**Sequence** diagram)
- Zustandsdiagramm (**State** diagram)
- Package-Diagramm (**Package** diagram)
- Einsatzdiagramm (**Deployment** diagram)

In welchen Bereichen kann UML sinnvollerweise eingesetzt werden?

- **UML als Entwurf**
  - Entwickler benutzen UML um einige Systemaspekte besser zu kommunizieren.
  - Da Skizzieren nicht formal und dynamisch ist, aber auch sehr schnell geht, kann man es leicht gemeinsam betreiben. Ein gemeinsames Medium ist eine weisse Tafel.
  - Die meisten UML Diagramme, die in Büchern zu finden sind, sind Entwürfe.
- **UML als Bauplan**
  - Die Idee ist es, dass ein Designer einen detaillierten Bauplan entwickelt, der für Programmierer als Leitfaden zur Codierung dient.
  - Das Design soll hinreichend vollständig sein sodass alle Designentscheidungen darauf aufliegen.
  - Die Inspiration zu diesem Ansatz sind andere Ingenieursdisziplinen, in denen professionelle Ingenieure Baupläne zeichnen und diese Baufirmen zum Aufbauen übergeben.
- **UML als Programmiersprache**
  - Entwickler zeichnen UML Diagramme, die direkt in ausführbaren Code kompiliert werden und das UML der Source Code wird.
  - Eine der interessanten Fragen zu UML als Programmiersprache ist, wie man Verhaltenslogik modelliert.
  - **Modelgetriebene Architektur (Model Driven Architecture MDA)** ist ein Standardansatz UML als Programmiersprache zu nutzen (auch von der OMG kontrolliert).

Zeichnen Sie ein Klassendiagramm für eine Klasse ‚Buch', die ein oder mehrere Objekte der Klasse ‚Seite' als Referenzen hält.

Möglicherweise fehlerhaft! Bitte unbedingt überprüfen und geg. ausbessern!



Kapitel 9 - Anhang

Ad Kapitel 2 - Was ist ein Stakeholder?

Falls das irgendwo anders noch reinpasst, bitte dorthin kopieren. Warscheinlich passt das noch gut zur Definition von nicht-funktionalen Anforderungen (Darauffolgende Frage).

System Requirements von Stakeholdern zu verstehen ist, aus mehreren Gründen, keine leichte Aufgabe:

- **Systemimpedanz:** Skateholder wissen oft nicht, was sie vom Computersystem wollen. Sie verlangen unrealistisches, weil sie nicht wissen, zu was das System fähig sein kann.
- **Domänenimpedanz:** Stakeholder beschreiben Anforderungen in ihrer eigenen Sprache und mit implizitem Wissen aus ihrer Arbeit.
- **Stakeholderimpedanz:** Verschiedene Stakeholder haben verschiedene Anforderungen. Es ist Aufgabe des Requirements Engineers die Gemeinsamkeiten zu finden.
- **Politische Impedanz:** Politische Faktoren können die Anforderungen an das System beeinflussen. Ein Manager möchte durch das System oft mehr Einfluss im Unternehmen.
- **Ökonomische/Business Impedanz:** Diese Umgebungen sind dynamisch und ändern sich möglicherweise während des Analyseprozesses. Neue Anforderungen kommen plötzlich von neuen Stakeholdern zum vorschein.

Ad Kapitel 2 - Was ist bei der Defintion von nicht-funktionalen Anforderungen zu beachten?

Dieser Abschnitt gehört nicht direkt zu nicht-funktionalen Anforderungen, sondern ist ein Auszug aus dem Volere Template (Vorlage für Requirements Document) und passt somit eher zur Frage “Was versteht man unter einem Requirements Document?”.

- **Look and Feel:** Aussehen des Systems und das Gefühl damit zu arbeiten soll zufriedenstellend und motivierend sein.
- **Usability:** Es soll weitestgehend selbsterklärend und intuitiv zu bedienen sein.
- **Performance/Throughput/Capacity/Security:** Das System soll schnell sein, effektiv und sicher.
- **Operational Requirements:** ?
- **Maintenance and Migration:** Das System soll leicht und gut wartbar sein und auch einfache Möglichkeiten bieten, es auf andere Hardware zu migrieren.
- **Zugangsbeschränkungen (Access):** Benutzer sollten nur so viele Rechte haben, wie sie auch tatsächlich benötigen.
- **Kulturelle und Politische Anforderungen:** ?
- **Rechtliche Anforderungen:** Das System soll den Rechtsvorschriften und der lokalen Gesetzeslage entsprechen.

Ad Kapitel 5 - Beschreiben Sie die Unterteilung der unterschiedlichen Arten von Tests.

Dies ist keine direkte Antwort auf die Frage, sondern eine Erklärung einer Unterkategorie von Tests. Da dies nirgendwo spezifisch gefragt ist, ist es vermutlich unwichtig.

Unit Testing
Ist das Testen von Programmkomponenten, wie Methoden oder Klassen. Tests sollten Aufrufe dieser Routinen mit verschiedenen Parametern sein und sollten alle Features des Objektes umfassen.

Man sollte:

- **Alle Operationen**, die mit dem Objekt verbunden sind testen
- **Allen Attributen**, die dem Objekt zugehörig sind, **Werte zuweisen und diese überprüfen**
- **Das Objekt in alle möglichen Zustände setzen**, also alle Ereignise, die Zustandsveränderungen einleiten können simulieren

Unit Testing ist ein Defect Testing Prozess. Es hat also zum Ziel, Fehler und Defekte in den Komponenten aufzudecken. Unit Tests verbessern das Design, reduzieren die Debuggingzeiten und erlauben Refactoring.
Unit Tests verhalten sich wie ausführbare Dokumentation.
Unit Testing sollte immer, wenn möglich, mittels eines Testautomationsframeworks automatisiert werden, besonders, wenn es für **Regressionstesting** benutzt wird, also das Wiederholen vorheriger Tests, um zu überprüfen, dass durch die Änderungen im Programm keine neuen Bugs entstanden sind.