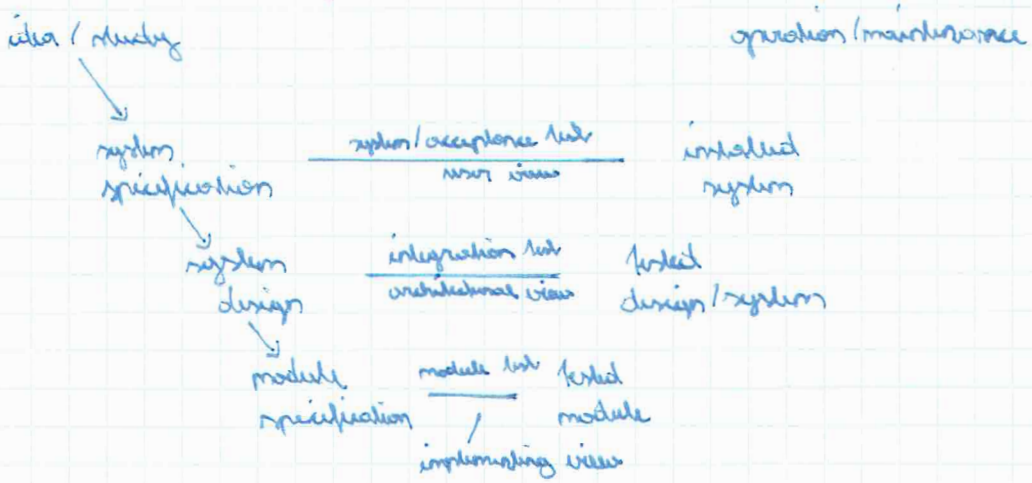
Software Qualitätssicherung

Planung und Sicherung von Qualität



Qualität: ... Kundenzufriedenheit
... Gesamtheit von Eigenschaften und Merkmalen eines Produkts, die sich auf die Eignung zur Erfüllung vorgegebener Anforderungen beziehen

- Termin- und Budgetgerecht erstellt
- für Anwender verwendbar
- für Professionellen verständlich und änderbar
- für Betreiber effizient und administrierbar

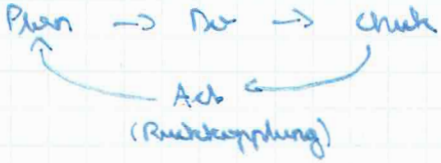
→ kann nicht im Nachhinein hinzugefügt werden

Qualitätssicherung: Durchführung von Verifikation und Validierung in jeder Phase der Software Herstellung

Qualitätsmanagement: Menge aller Aktivitäten, Vorgehensweisen, Techniken die sicherstellen, dass ein Softwareprodukt vorgegebene Standards erreicht oder übertrifft

Herstellung qualitativ hochwertiger Produkte erfordert:
- angemessener Softwareprozess und adäquate Vorgehensmodelle
- wirkungsvolle Methoden zur Produktherstellung
- wirkungsvolle Methoden zur Produkt- / Prozessverbesserung

Produkt / Prozessverbesserung: PDCA



Plan: Planen der nächsten Iteration
Do: Durchführung der geplanten Aktivität
Check: Überprüfung der Produkt/Prozessergebnisse
Act: Analyse der Ergebnisse als Feedback für nächste Planung

Erzeugung und Erhalt von Qualität

Qualitätsfaktoren

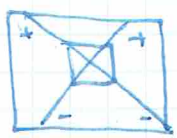
- Mehrwert vs Hygienefaktoren: Kunde kauft nicht bei Mangel welche Hygienefaktoren betreffen

Qualitätsmaßnahmen: das Richtige einfach machen, das Falsche erschweren

Qualitätsmodell nach McCall:
- Qualitätsfaktoren: beschreiben Verhalten des Systems
- Qualitätskriterien: Eigenschaften von Qualitätsfaktoren in Bezug auf Softwareentwicklung
- Qualitätsmetriken: Messungen welche definierte Aspekte der Qualitätsfaktoren beschreiben

- Messungen: direkt vs indirekt
  objektiv vs subjektiv
  quantitativ vs qualitativ

Teufelsquadrat nach Snaid:



Fläche des Quadrats ändert sich nicht (= Produktinvariante)

SE Prozessmodelle und QS

- Konstruktive Qualitätsmaßnahmen:
  - technische Maßnahmen: Prinzipien, Methoden, Werkzeuge
  - organisatorische Maßnahmen: Vorgehensmodelle
  - menschliche Maßnahmen: Schulungen

- Analytische Qualitätsmaßnahmen: Prüfung/Bewertung der Qualität
  - statische Prüfungen: Reviews, Inspektionen
  - dynamische Prüfungen: Software Tests

V-Modell XT: eigenständiges, verpflichtendes Vorgehensmodell

RUP: ständiges Begleiten während gesamten Projektverlauf, Review bei allen Meilensteinen

Agile Entwicklungsprozesse: Reviews und Tests hier Bestandteil des Sprints

---

Prinzipien Qualitätsmanagement:
- Produkt / Projektabhängige Qualitätsplanung
- Mehraugenkonzepte bei Qualitätsprüfung
- Frühzeitige Erkennung und Behebung
- Bewertung der eingesetzten Maßnahmen

Prinzip der ständigen Verbesserung von Produkten und Prozessen

Qualitätssicherung & Fehlerreduktion

Zielsetzung: frühe Fehlererkennung
→ umso später ein Fehler erkannt wird, umso höher die Kosten
→ analytische Methoden in frühen Phasen der Softwareentwicklung
↳ Reviews / Inspektionen erfordern keinen ausführbaren Code:
können auf Anforderungsdokument / Designdokument angewandt werden

Reviews: formal geplanter / strukturierter Analyse- und Bewertungsprozess

→ qualitative Bewertung von Produkten / Prozessen welche Quantitativ nur schwer beurteilt werden können: Dokumente, Modelle
→ Beurteilung des Produkts und nicht des Autors
→ definierte Rollen mit definierten Aufgaben

→ mit Kunde:	→ ohne Kunde
SRR	MR
PDR	Inspektion
CDR	Code Walkthroughs
IPR	Technical Review

Review: Stärken / Schwächen des Prüfobjekts identifizieren
Inspektion: Schwere Defekte im Prüfobjekt identifizieren
Metriken ermitteln
Code Walkthroughs: Defekte / Probleme des Prüfobjekts identifizieren
Ausbildung von Benutzern / Mitarbeitern

Rollen: Moderator, Leser, Gutachter, Schreiber, Autor

Inspektionen: spezielle Reviews welche in frühen Phasen der Softwareentwicklung angewandt werden können

→ formale, effiziente, wirtschaftliche Methoden um Fehler in Design und Code zu finden
→ Lesetechniken unterstützen Inspektion durch systematisches Lesen
↳ Perspektiven, Rollen, Szenarien, Checklisten
→ Fehlerfindung in Anforderungs- und Designdokumenten

Lesetechniken zur Fehlerfindung:
- Ad-hoc: unstrukturiertes Lesen
- checklistenbasiertes Lesen: vordefinierte Fragestellungen
- Perspektivenbasiertes Lesen: Fehler aus individueller Sicht
- Usage Based Reading → Best Practice Inspektion
→ Fehler in wichtigen Anwendungsfällen zuerst finden
→ Überprüfung der für Use Case relevanten Teile

→ Ablauf Review: Planung: Objekte, Ziele, Teilnehmer, Ort / Zeit
Vorbesprechung: bei komplexen / neuen Produkten
Individuelle Vorbereitung
Reviewsitzung: Gemeinsames Lesen, Aufzeichnung Mängel
Nachbearbeitung (Rework): Mängel korrigieren
Bewertung (Follow Up): Überprüfen der behobenen Mängel

Anforderungen: Korrektheit, Vollständigkeit

funktionale Anforderungen: beschreiben was das System tut

nichtfunktionale Anforderungen: beschreiben wie das System etwas tut

→ Constraints: langfristige Einschränkungen durch Frameworks/ Architekturentscheidungen

Klassifizierung durch FURPS:
- Functional
- Usability
- Reliability
- Performance
- Supportability

Reviews überprüfen Korrektheit + Konsistenz von Artefakten an verschiedenen Punkten des Entwicklungsprozesses:
→ Anforderungen gegen Prozessbeschreibung
→ hergestellte Dokumente gegen aktuell gültige Anforderungen

Von Anforderungen abgeleitete Modelle:
- Anforderungen → Modelle → Datenbank, Logik, GUI
  Anforderungen → Modelle → Review, Tests

UML Sichten auf Softwarestrukturen

- Konzeptionelle Sicht: Übersicht über Problembereich
  Domänenmodell
- Spezifizierende Sicht: Software Schnittstellen - keine Implementierung
  Komponentendiagramm
- Implementierende Sicht: Grundlage für Implementierung
  Klassendiagramm

→ Überprüfung Domänenmodell mit Anforderungen
→ Überprüfung Domänenmodell mit Anforderungstext
→ Überprüfung Diagramm mit übergeordnetem Diagramm

Software Testen – Theoretische Hintergründe

Ziel: Herstellung von qualitativ hochwertigen und testbaren Softwareprodukten

→ Anwendung von Software Prozessen
- wann muss was mit welcher Qualität fertig sein?

→ Konstruktive Methoden zur Herstellung der Produkte
- wie werden Produkte erstellt?

→ Analytische Methoden zur Verifikation und Validierung der Ergebnisse
- wie erfolgt Überprüfung der Qualitätseigenschaften

Testen: ausführen eines Programmes, mit der Absicht, Fehler zu finden

Fehler: Abweichung zwischen Programm und dessen Spezifikation
Programm tut nicht das, was Benutzer vernünftigerweise erwartet

Testen ≠ Debuggen: Lokalisierung und Entfernen von Fehlern
Testen ≠ Fehlerfreiheit: formaler Korrektheitsbeweis

Verifikation: Bauen wir das Produkt richtig?
Umsetzung gegen Spezifikation

Validierung: Bauen wir das richtige Produkt?
Umsetzung gegen Anforderung

Testen im Softwareentwicklungsprozess:

Testen in traditionellen Prozessen: V-Modell
→ erkennen von Fehlern in Implementierung (Verifikation, Validierung)

Testen in agilen Prozessen: SCRUM
→ engere Kopplung von Implementierung und Testen

→ Test driven Development: Think: Anforderungen
Red
Green
Refactor

Aufbau von Testfällen: Vorbedingung
Eingabewerte
Aktionen
erwartetes Ergebnis
} (V, E, A, R)

Test Abdeckung:
- Abdeckung aller Funktionen
- Abdeckung aller Eingabeklassen
- Abdeckung aller Ausgabeklassen

Äquivalenzklassenzerlegung

→ Zerlege Menge von Eingabedaten in Untermengen (Klassen), welche äquivalente Ergebnisse / Auswirkungen produzieren
Eingabevariablen: Explizite Parameter, implizite Parameter (Systemzustand)

→ Jede Klassenkombination sollte mindestens ein mal getestet werden

Grenzwertanalyse: Erweiterung der Äquivalenzklassenzerlegung für bessere Abdeckung

→ Grenzwerte werden als Klassenrepräsentanten gewählt

Value-Based Testing:

→ Requirement-Based Testing
→ Risk-Based Testing
→ Testcase selection Techniques

Test-Typen: Review
Funktionaler Test
Messen Test
Stress Test
Performance Test

Test Stufen:

→ Private Tests
→ Komponenten Tests: Modulimplementierung gegen Spezifikation
(Unit-Tests) genaue Lokalisierung, frühe Erkennung
→ Integrations Tests: Test der Interaktion zwischen Modulen
→ System Tests: Gesamtsystem gegen Anforderungen / Entwurf
→ Abnahme / Akzeptanz Test: System gegen Kundenanforderungen

Black Box Tests: basiert auf Spezifikation
Wissen über innere Struktur ignoriert
Äquivalenzklassen der Eingabedaten
Input-Output getrieben

White Box Tests: basiert auf Code
Wissen über innere Struktur
Überdeckung von Knoten / Kanten / Pfaden
Logik getrieben

Software Testen – Praktischer Teil

Zerlegung in Teststufen ermöglicht frühe Prüfung von unterschiedlichen Teilen des Systems:

- Komponententest: Klassen, Objekte, Module
  Isolation der Komponenten vom Rest des Systems
- Integrationstest: Schnittstellen zwischen Komponenten
  Inkrementelle Integration vs Big Bang Integration
  Horizontale Integration vs vertikale Integration
- Systemtest: Verhalten des Gesamtsystems
  prüfen funktionale und nichtfunktionale Anforderungen abdecken
- Abnahmetest: Nachweisen der vereinbarten Leistungen
  von Kunden oder Benutzern ausgeführt (meistens)

→ Testabdeckung: Ziel von Verifikation ist möglichst hohe Abdeckung des Systems

- Identifizierung von Menge an Testfällen, die mit höchster Wahrscheinlichkeit Fehler finden
- strukturelle Abdeckung: White Box Tests
- funktionale Abdeckung: Black Box Tests

Hybrid Testing (basierend auf JUnit)

- Test phases: Set up, Exercise, Verify, Tear Down
- Benennungsschema: <unsere> - should <expected Result>
  → unsere Langfiles
- Lifecycle Methoden: @ BeforeClass } statisch
  @ AfterClass }
  @ Before
  @ After
- Assertions: statische Methoden, Hamcrest → Fluent Interface
- Testen von Exceptions: expected, fail
- Parameterized Tests
- Categories
- Rules
- Testreihenfolge

→ Best Practice: Einhaltung von Namenskonventionen
Klare Struktur Testvorbereitungen
Kurze Abhängigkeiten zwischen Testfällen
Eigene Logik testen, nicht Frameworks

→ Bad Practice: lange Tests, viele Assertions
Wiederholung von Vorbereitungen
Random Logik statt Äquivalenzklassen

Test Doubles

- Dummy Object: dienen als leere Methodenparameter
- Fake Object: ausführbare Implementierung liefern keine echten Daten
- Stub: liefert vordefinierte Daten
- Mock: liefert vordefinierte Daten überprüft Parameter eigene Funktionen

→ Bottom - Up Integrationstests
→ Top - Down Integrationstests: Mocken der unteren Schichten

Black Box Tests: Äquivalenzklassenanalyse
Grenzwertanalyse

Kontrollflussorientierte Testverfahren: basieren auf Quellcode
relevant auf Unit Test Ebene

→ orientieren sich an Kontrollflussgraphen des Programms

- Anweisungsüberdeckungstest: C_0
- Zweigüberdeckungstest: C_1
- Bedingungsüberdeckungstest: C_2
  einfacher Bedingungsüberd.: alle atomaren Entscheidungen true + false
  mehrfacher Bedingung.: alle Wahrheitswertkombinationen atomaren Entscheidungen
- Pfadüberdeckungstest: C_3
  Durchführung aller unterschiedlichen Pfade (Sequenz von Knoten)
  in Praxis kaum durchführbar

Automated Quality Assurance

... QA support engineers in evaluating products in every stage of software development and must be an integrated part of the development process

- DB Unit: automated database configuration
  set database to defined state before Test execution

- Stub- / Mock Objects: mimic behaviour of real object
  real object is difficult / expensive to create

- HTML Unit: emulate Browser behaviour
  model HTML documents which are returned from a server

- HTTP Unit: model HTTP protocol

- JMeter: make load and performance tests on servers (ftp, HTTP, JDBC)
  functional tests according to scripts

Code Quality Checks: hints about potential problems in code
can be embedded in build automation

- Checkstyle: JavaDoc Comments, Naming Conventions, Imports, Modifier order, Duplicate code

- Find Bugs: based on the concept of bug patterns
  close streams, useless assignments,
  Thread.start in constructor, Empty DB password

Automation: embed tasks in automated build-environment (ant, maven)

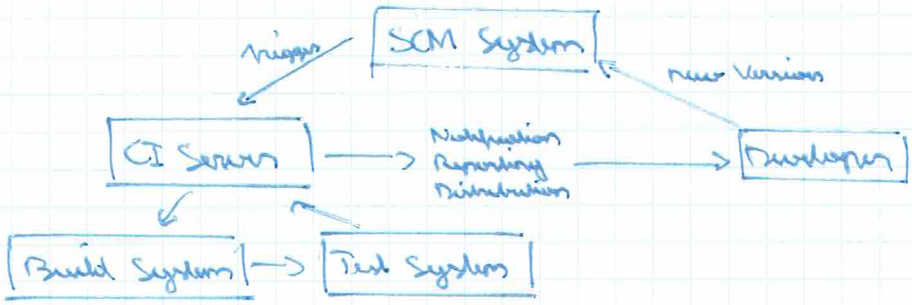
- execute tests as often as possible
- create automated reports and notifications
- integrate development tools as good as possible
  -> build automatisation
  -> Reporting
  -> Code Management
  -> Bug Tracking

Continuous Integration: Apache Continuum

-> build automation like maven is great, but developer centric
-> server based automated integration and execution of tests

- invocation based on events or schedules
- build using build system
- execution of tests
- creation of reports
- sending of notifications

-> best practices: single source repository
build automation, keep build fast
self testing build
test in clone of production environment
transparency: latest binary, build status



-> Documentation and Reporting are integral aspects of automated testing: generic formats (XML, HTML), integration in documentation system

-> Issue Tracking: many problems are found by humans
users should be able to add issues
- trac

- Trac integration: post-commit hooks: Trac ticket ID in commit msg
Report on SVN commits
notify developers

Refactoring: change inner structure of system, but keep external behaviour

-> cleaning up code: no bug fixing
<> understandability, reusability
-> systematic approach to enhance code quality

Bad Smells: indications that there might be design issues
<> duplicated code, long methods, long parameter list, large classes

- encapsulate fields
- extract method
- move method
- replace temp with query: method call instead of temp variable

Software Tester im agilen Umfeld

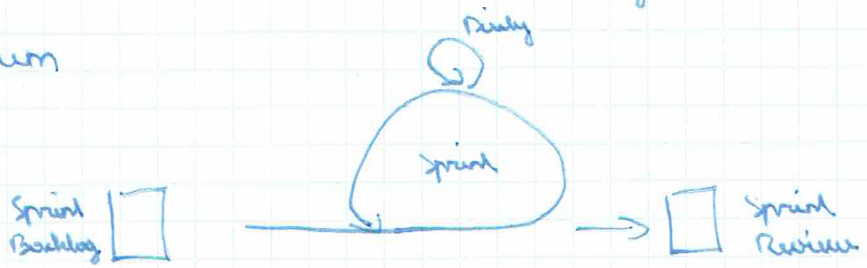
... Tester in agilen Projekten arbeiten anders als Tester in klassischen Projekten. Sie sind gleichberechtigte Mitglieder des Teams (Whole-Team-Approach) und kommunizieren von Beginn an mit dem Team

Agile Manifesto:
- Individuen und Interaktionen sind wichtiger als Prozesse und Werkzeuge
- Funktionierende Software ist wichtiger als umfassende Dokumentation
- Zusammenarbeit mit dem Kunden ist wichtiger als Vertragsverhandlungen
- Reagieren auf Veränderungen ist wichtiger als das Befolgen eines Plans

Kanban: Optimierung des Arbeitsflusses innerhalb einer Wertschöpfungskette

- Kanban Board: Visualisierung der Wertschöpfungskette
- Work-in-Progress Limit: Limitierung der gleichzeitig zu erledigenden Arbeiten. Limit je Bearbeitungsstation oder im gesamten Board
- Lead Time: Senkung der durchschnittlichen Bearbeitungszeit
  → Kontinuierlicher Fluss durch gesamte Wertschöpfungskette

Scrum



Sprint Retrospektive

- Produkterweiterung / Inkrement
- Product Backlog
- Sprint Backlog
- Definition of Done
- Timeboxing
- Transparenz: Daily

Rollen:
- Scrum Master
- Product Owner
- Entwicklungsteam

→ Scrum Prozess sieht keine Tester vor
- Tester ist Teil der Entwicklung (Unit Tests, TDD)
- Testen geschieht durch Product Owner / Kunde
  - nach jedem Sprint

User Stories ... Anforderungen aus Sicht der Fachbereichsvertreter
funktionale + nichtfunktionale Anforderungen
Abnahmekriterien

→ User Story Card: Conversion: wie wird Software gemacht
Confirmation: Abnahmekriterien

→ Wer? (user role), Was? (goal), Warum (reason)
as a user role, i want to goal, so that i can reason

Iteration Zero

- Identifikation Projektumfang → Product Backlog
- Erstellung initiale Systemarchitektur
- Planung, Installation notwendiger Werkzeuge
- Erstellung initiale Teststrategie
- Definition von Metriken, Definition of Done

→ Aufwandsabschätzung: Abschätzen der Komplexität durch Planning Poker
→ Story Points (Fibonacci), T-Shirt größen

→ Iterationsplanung: Priorisierung offener Stories im Product Backlog
Diskussion über Scope des Sprints
→ komplettes Team

↳ Aufgaben Tester: Risikoanalyse User Stories
Abnahmetests User Stories
Schätzung Testaufwand
Identifizierung funktionale + nichtfunkt. Anforderungen
Unterstützung bzgl. Test-Automatisierung

→ Burndown Chart: X-Achse: zeitlicher Verlauf
Y-Achse: verbleibender Aufwand
Gegenüberstellung: Ideal Tasks Remaining
Actual Tasks Remaining

Sprint Ende:

→ Sprint Review: Demo vollständiger Stories gegenüber Stakeholder, Product Owner
Formale Abnahme der Stories

→ Retrospektive: Was war erfolgreich?
Was kann verbessert werden?

Continuous Integration: Integration von Inkrement mit funktionierender Software voraus

→ Geänderte Komponenten werden regelmäßig zusammengeführt (nightly)

- Statische Codeanalyse
- Kompilieren
- Unittests, Integrations + Systemtests
- Deployment auf Testumgebung
- Report erstellen

Test-Automatisierungspyramide:

Business-centric	GUI Tests	Selenium
	Akzeptanz Tests	Cucumber
Developer-centric	Unit / Komponenten Tests	x Unit

Agile Testmethoden:

- Test Driven Development
- Abnahmekriteriengetriebene Entwicklung: Abnahmekriterien und Tests werden mit User Story gemeinsam erstellt
- Verhaltensgetriebene Entwicklung: Erwartetes Verhalten wird in Given-When-Then Format definiert. Daraus wird ausführbarer Testcode generiert

Agile Testpraktiken:

- Pairing: gemeinsame Bearbeitung von Aufgaben
- Inkrementelles Test Design: Testfälle aus User Stories ableiten. Mit einfachen Tests beginnen, darauf aufbauend komplexe Tests
- Mind Mapping: Darstellung von Teststrategien / Daten

Abnahmekriterien

- Definition von detaillierten, testbaren Abnahmekriterien für User-Story

→ Conditions of Satisfaction: Broad Terms
Acceptance Criteria: Further refined
Examples: Actual Scenarios or Data
Executable Examples: Ready to Automate

} Stufen von Akzeptanzkriterien

Exploratives Testen: wichtig da begrenzte Zeit für Testanalyse und begrenzte Genauigkeit der User Stories

→ erfahrungsbasiertes Testen, risikobasiertes Testen, anforderungsbasiertes Testen
→ Ausdrucksfähigkeit, Analysefähigkeit, genaue Überprüfung

Werkzeuge für...

- Aufgabenmanagement und Nachverfolgung: Stories, Aufgabentickets
- Kommunikation und Informationsweitergabe: Email, Wiki
- Build und Distribution
- Konfigurationsmanagement: manuelle Tests, VCS
- Testentwurf, Implementierung, Durchführung:
  → Testfallmanagement
  → Testdatenverwaltung, -verwendung
  → Testdurchführung