# Projektplanung

Unified Process Management: Inception, Elaboration (typ. 15-25%), Construction (min. 50%), Transition. Jeweils Iterationen à 2-3 Wo.

Bei jedem Sprint soll der Gesamtwert für den Kunden steigen.

Entwickler schätzen den Aufwand pro Arbeitspaket und der Kunde priorisiert den Arbeitspaket.

Dokumentation: Requirments (das wird der Kunde kriegen), Projektplanung (so gehen wir vor), Software Architecture Documentation (Bauplan), Projektnachverfolgung (hier stehen wir im Moment)

Achtung: Prozent der Zeit ≠ Prozent der Kosten. Meist gilt Pareto-Prinzip, d.h. 80 Prozent der Ergebnisse werden in 20 Prozent der Projektzeit erreicht. Die meisten komplexen Probleme tauchen erst am Ende der Projektzeit auf.

## Use Cases (Elaboration-Phase)

Use Cases beschreiben nur die Funktionalität.

Use Cases bzw. dann Use Case Diagramme helfen für Definition of Scope, denn man erhält gute Übersicht über gewünschten Funktionsumfang und kann entscheiden, was in scope bzw. out of scope ist. Gute Basis für System-Sequenzdiagramme.

Domain-Modell: Komplementär zu Use Cases. Beschreibt, was wir uns während der Laufzeit des Programmes merken, und was wir evtl. darüber hinaus speichern.

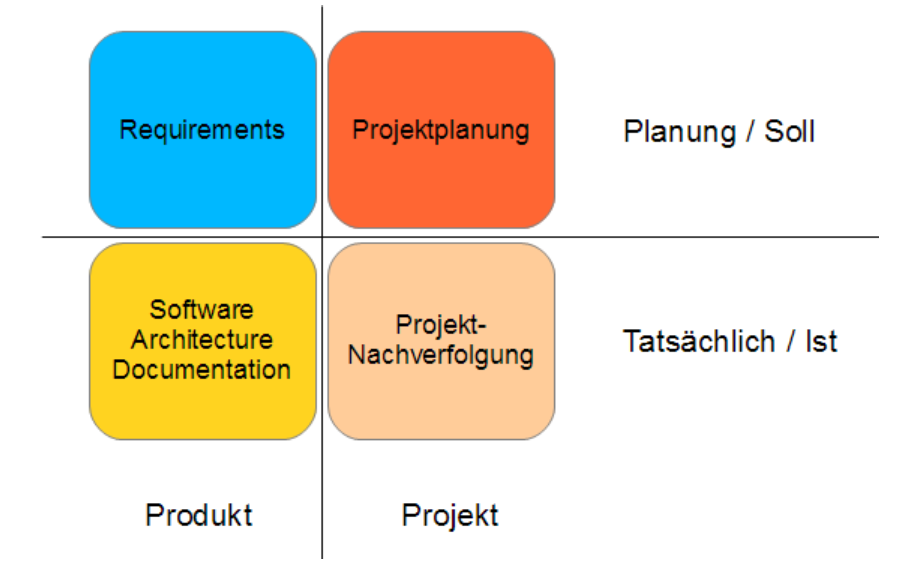
Use Cases = Dynamik | Domain-Modell = Persistenz

Gilt vor allem bei Kundenprojekten, aber weniger wenn man ein Produkt für 1 Mio. User entwickelt (Domain Driven Development).

## Checkliste End of Elaboration

Kunde ist verstanden, Werkzeuge bestimmt, Architektur bekannt, grober Entwurf des UI steht. Detailplanung für die nächsten zwei Iterationen. Requirements erfüllt? Scope abgedeckt (Use Cases)? UI Entwürfe? Prototype (möglichst alle Schichten)? Software Architektur Entwurf? Werkzeuge & Methoden definiert? Liste der Arbeitspakete.

Nicht-funktionale Anforderungen ergänzen Use Cases und Domainmodell.



4 Quadranten der Dokumentation

In der Mitte: Transiente Zwischenresultate (Arbeitspakete, User Stories, Issue Tracking, Automatic Build Results).

Vor *End of Elaboration* liegen die Haupt-Anstrengungen bei der Dokumentation darauf, zu zeigen, dass man den Kunden verstanden hat. Danach liegt Fokus Umsetzen der Lösung, d.h. die Dokumentation, die entsteht, ist hauptsächlich für die Entwickler.

# Scrum

Voraussetzungen: Team < 10 Personen; Projekt < 9 Monate. Alle an 1 Ort, Kunde immer verfügbar; Vertrauen des Kunden notwendig sowie seine Unterstützung für das Vorgehen. Team ist selbstorganisierend.

Product Owner: Der ‚Projektleiter’ im Scrum, priorisiert Backlog. Vertritt fachlicher Arbeitgeber -> Stakeholders. Passive Teilnahme am Daily Standup Meeting. Er ist nicht der Chef des Teams, entwickelt nicht mit und macht keine Anpassungen des Backlogs während eines Sprints.

Scrum Master: Prozess Owner. Beseitigt Probleme, holt das Maximum aus dem Team heraus und schützt es vor unberechtigtem Eingreifen während des Sprints, Er ist nicht der Chef des Teams; keine Doppeltfunktion als Product Owner.

Sprint Planung: Definition Backlog, Kickoff von Sprint. Maximal 8h.

Sprint Review: Demo Ergebnisse letzter Sprint für Product Owner und Stakeholders. Maximal 1h. Anschliessend intern im Team diskutieren was besser/anders gemacht werden kann.

Story Points: Entsprechen nicht effektiven Arbeitsstunden, denn unterschiedliche Mitarbeiter haben auch unterschiedliche lange. Sind also proportionale Abstraktion, denn so haben alle die gleiche Basis.

Use Case/Epics -> User Story. User Story sollte nicht grösser sein als 70% dessen, was 1 Entwickler in 1 Sprint schafft. Durschnitt 25%. Pro Iteration genug Arbeitspakete; eigenverantwortliche Zuteilung.

# Aufwandschätzung

Keep it simple; Komplexität tief halten. Aufteilen, separat planen, ausführen und abliefern (siehe Story Splitting). Schätzen ist Erfahrungssache. Aufwand ist hauptsächlich bestimmt durch: Grösse (Funktionalität), Art der Software (Komplexität), Qualität der Mitarbeitenden.

Entwickler schätzen Aufwand pro Arbeitspaket, der Kunde priorisiert. Ideal Schätzung im Team, denn so gibt es genauerer Ergebnis dank mehreren Einflüssen. Team weiss am besten, wie lange Umsetzung dauert.

## Top-Down-Schätzung

Zu Beginn des Projekts. Benutzen nur ganz wenige globale Parameter und ein paar Formeln und Annahmen.

Vorgehen: 1. Modell nehmen. 2. Schätzung mit vorgegebenen Einstellungen durchführen. 3. Nach Projektende Soll & Ist vergleichen. 4. Modell kalibrieren.

## Bottom-Up-Schätzung

In Elaboration-Phase. Dafür braucht man alle Requirements sowie Architektur-Entwurf, dann Liste der Arbeitspakete. Alle Elemente einzeln schätzen.

## Function Points

Externe Inputs / Outputs / Queries und Tabellen zählen. Gewichten und zusammenzählen. Nachschauen in Tabelle und dadurch Anzahl Personentage / -monate herausfinden.

# Story Splitting

Was tun, wenn Projekt oder Arbeitspaket zu gross wird? Lösung: In Teilen ausliefern. Aufteilung nach Kunden-Domäne, nach Geschäftsprozessen (Use Cases), Rollen (Admin, Einzelkunde, Übersetzer, usw.) oder etwa Geografie. Zuerst Basis-Version, danach Ausbau in Schritten. Stichwort **M**inimum **V**iable **P**roduct.

Ideale Grösse von Paketen:

Durchschnitt: was 1 Person in 1/4 eines Sprints schafft.

Maximum: was 1 Person in ca. 70% eines Sprints schafft.

User Story ≠ Use Case. Story: Klein, umsetzungsorientiert, wenig Kontext. Case: Grösser, bilden oft Geschäftsprozesse ab, kundenorientiert (Aktoren und deren Motivation).

## Story Mapping

Kleinere User Stories zu grösseren Themenbereichen zusammenfassen. Besserer Überblick & Fokus auf einzelne Themenbereiche in Bezug auf Priorisierung und Fortschritt.

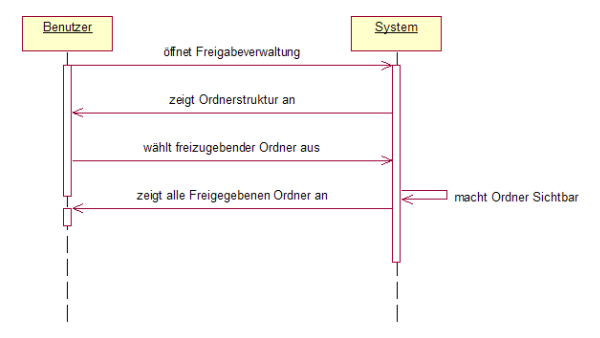
Epics (≈ Use Cases) > User Stories > Tasks

# Sequenzdiagramme

System-Sequenzdiagramm ≠ Sequenzdiagramm.

## System-Sequenzdiagramm

System-Operationen. Immer nur Aktor + System. System = Blackbox; abstrahiertes Systemverhalten.



## Sequenzdiagramm

Immer mehrere Handelspartner, System = Whitebox; zeigt Zusammenspiel mehrerer Klassen; von/nach Code generierbar.

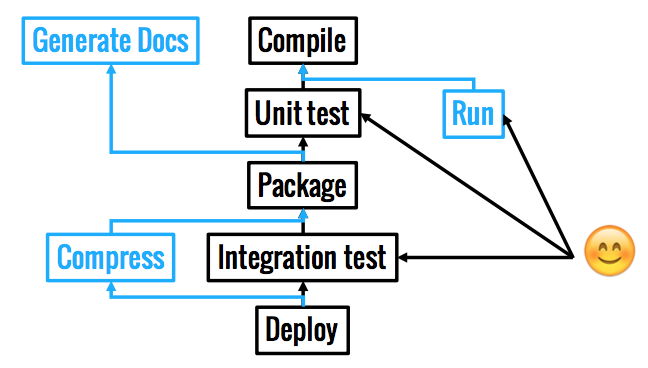
## ../../../../Desktop/Screen%20Shot%202016-08-27%20at%2015.08.

# Automatisierung

Build Script: Compile, unit tests, package, integration tests, deploy. Idealerweise mit 1 Befehl.

**C**omplete **R**epeatable **I**nformative **S**chedulable **P**ortable

Lösung: Build Tool. Build-Prozess als DAG modellieren.



Imperative Build Tools wie make, rake, Gulp sind zwar mächtig, aber wird schnell komplex und nicht gut wiederverwendbar.

Deklarative Builds mit Apache Maven oder Gradle. Zwar gutes Dependency Management, aber weniger flexibel.

## Continuous Integration

Ziel: Immer ein lauffähiges Produkt, häufige Commits in master.

Rules: Build automatisieren (schnell!), Build ist selbst-testend, kleine Commits, jeder Commit = Build. Nichts wichtiger als fehlgeschlagenen Build fixen.

Testumgebung möglichst wie Production.

Automatisierte Deployments, Test Coverage, Build History.

Tools: Jenkins, GitLab CI, Buildbot, Stride, MS Team Service, Travis CI

Vorteile / reduzierte Risiken: 1. Automatisierte und selbst-testende Builds eliminieren manuelle Build-Fehler. 2. Alle nötigen Scripts für Builds und Tests in VCS, damit haben alle die gleiche Basis. 3. Reduziert Fehler & erhöht Test Coverage, da jeder Commit Build (und somit Tests) auslöst. 4. Ermöglicht bessere Code Qualität, Entwickler können sich auf Kernaufgabe konzentrieren. 5. Automatisierte Deployments reduzieren manuelle Fehler. Mehr Sensitivität für Fehler.

# Software Engineering Practices

## Requirements Practices

Requirements herausfinden: Zusammenarbeit mit dem Benutzer; Kritisches Hinterfragen und Nachbearbeiten; Genügend abstrakt definieren; Verfolgung des Ursprungs.

Qualität als Requirement: Möglichst testbare Qualitätsanforderungen, basierend auf echten Anforderungen; wichtig aber auch schwieriger zu ermitteln.

Schlecht: "Stock orders shall be placed instantaneously".

Besser: "Stock orders should be placed within 100 ms after arrival.“

Nicht-funktionale Anforderungen: Wichtig, weil gerade Dinge wie Performance und Skalierbarkeit direkte Auswirkungen auf den Endbenutzer und Server- und Software-Architektur haben.

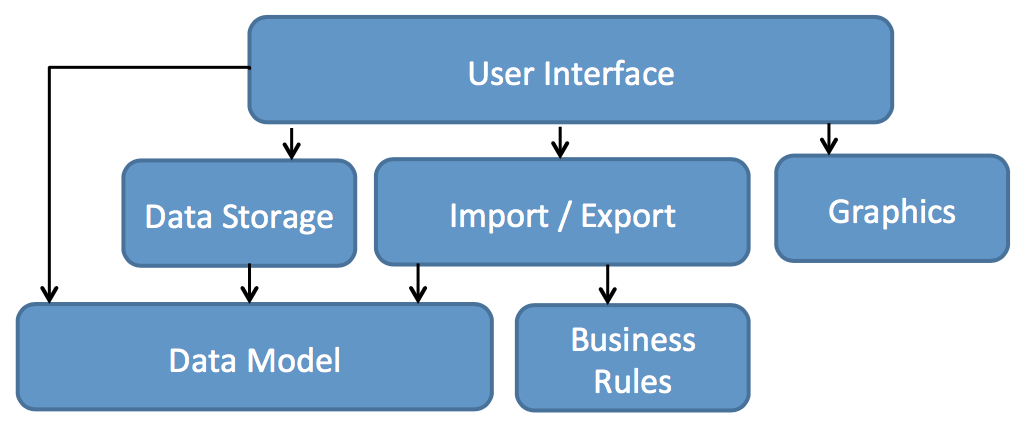
Ist Applikation langsam und nicht bedienbar, bringt auch die beste Architektur nichts.

Deal with Changes: Requirement-Änderungen antizipieren, bei Entwicklung berücksichtigen; kurze Iterationen, Qualität der Requirements danach prüfen (Change Assesment)

## Design Practices

**D**on’t **R**epeat **Y**ourself

Achieve Orthogonality: Hohe Kohäsion; keine Koppelung zwischen konzeptionell unabhängigen Aspekten. Einfacher zu verstehen & zu testen. Hierarchische Zerlegung in Komponenten: Aussen relevante Eigenschaften in Schnittstellen; Details ausblenden (Information Hiding); Reduktion der Kopplung; Acyclic Dependency Prinzip.



Design to Test: Testbarkeit *vor* Entwicklungszeit betrachten; Testbarkeit hat Einfluss auf Architektur; Klare Interfaces und Contracts; evtl. Freiheitsgrade für Dependency Injection (Factory & Strategy Pattern, Faking); kann Design auch komplexer machen als nötig.

## Testing Vocabulary

Fake: Vereinfachte schnellere Implementirung (z.B. In-Memory-DB). Mock: Auf Testfall zugeschnitten; prüft Reihenfolge und Inhalt der erwarteten Aufrufe. Stub: Auf Testfall zugeschnittene Antworten. Dummy: Objekte, die nur herumgereicht, aber nie inspiziert werden.

## Implementation Practices

Fix Broken Windows: Probleme beheben, wenn sie entstehen (kleinere sofort, grössere markieren für später). Werkzeug: Refactoring.

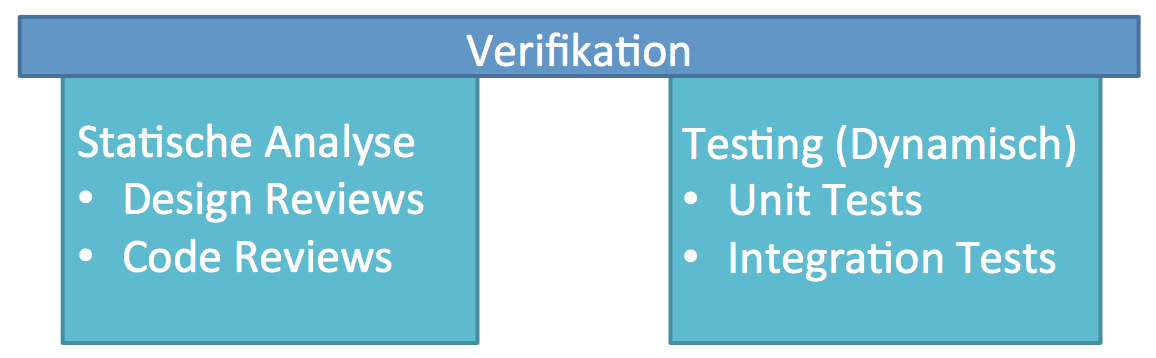
Refactoring early & often: Konstanter Verbesserungsprozess während Projekt. Liste von zu verbessernden Bereichen führen. Tests *vor* Refactoring haben. Mehrere kleine Schritte.

Program deliberately: Vermeide «Programming by coincidence (or luck)». Eingesetzte Technologien beherrschen, nur auf spezifizierte Features verlassen; korrektes Exception Handling; Crash early; Annahmen dokumentieren & mit Tests prüfen.

## Verification Practices

Test rigorously; TDD; hohe Code Coverage erzielen; für gefundene Bugs Tests schreiben. NB: Auch 100% Test Coverage bedeutet noch keine Korrektheit.

Perform reviews: Formal inspections (Reviews), zusammen oder einzeln, dokumentiert; Ziel: Bugs finden, keine Diskussionen über Architektur oder Autor o.ä.



# High Assurance Software Development

High assurance development means producing compelling evidence that a system meets specified requirements.

Reducing complexity of trusted code. Component-based architecture.

Identify security properties. Required to enforce security guarantees of system. These properties must hold even in the presence of an attacker.

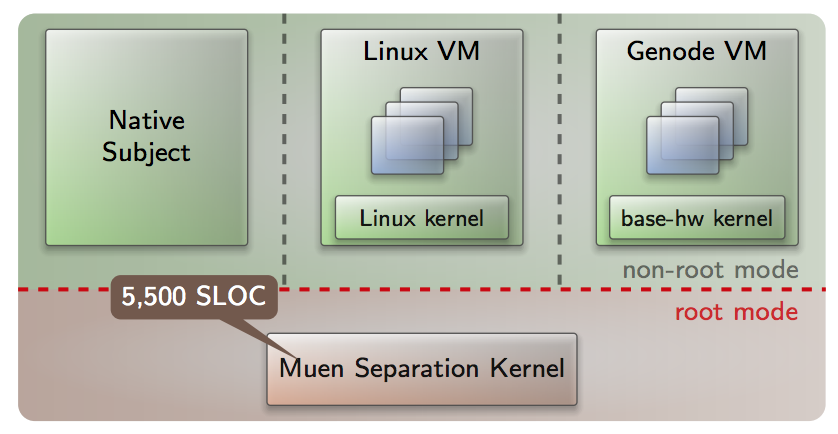
Threat model: Describe the capabilities of an attacker.

Trust: Any component/system that has the potential to break your security policy is by definition trusted. In contrast, trustworthy refers to whether that trust is warranted.

Reducing complexity of trusted code through a component-based architecture:

Split system in trusted/untrusted functionality; specify component interfaces and resources; reuse existing code for untrusted parts; maximize assurance of trusted components; employ solid separation. Shift complexity out of trusted components.

Separation Kernel: Kernel must guarantee component separation. Static partitioning and isolation of resources.



Identify security properties. Example: Only trusted components have access to sensitive data.

Threat Model: Describe the capabilities of an attacker. Example: The attacker can execute arbitrary commands in untrusted components

# Test Driven Development

Erster Schritt: RED. Man schreibt einen ersten Test, der fehlschlägt. Kompiliert und zeigt deutlich an, dass Funktionalität noch nicht implementiert wurde.



Zweiter Schritt: GREEN. Einfachen Code schreiben, damit Test nicht mehr fehlschlägt. So schlicht wie möglich, überhaupt nicht komplex.

Dritter Schritt: REFACTOR. Wichtigster Schritt. Produktiven Code verbessern, ohne Funktionalität zu beeinträchtigen. Green macht Code richtig für den Kunden, Refactoring macht ihn richtig für den Entwickler.

# Error Handling Design

Strategie wählen & konsistent verfolgen. Exceptions, Assertions, Logging.

Defensive Programming: Systematische Fehlerprüfung und –behandlung. Sanitize all input. Ungültige Zustände systematisch abfangen (Bsp: Default Case bei Switch).

Exceptions: Für mögliche produktive Fälle, sicherheitsrelevante Fehler. Assertions (-ea Flag): Für «Debug Mode» aka Programmierfehler, die nie auftreten sollten. Für Postconditions sowie Preconditions interner Quellen. Kein ausführbarer Code in Assertions!

Konservative Fehlerbehandlung: Error Handling Prozedur aufrufen, Meldung anzeigen, Shutdown.

Optimistisch: Neutrales Resultat, Warnung loggen.

Korrektheit (niemals ungenaues Resultat liefern) vs. Robustheit (niemals abstürzen)

Lokale Behandlung: Nur wenn Fall lokal abschliessend entscheidbar. Globale Behandlung: Wenn nicht lokal behandelbar; wenn auf höherer Systemebene relevant. Keine ungültigen Zwischenzustände hinterlassen (z.B. finally Block, Locks & Ressourcen freigeben, etc.).

Achtung vor möglichen Fehlern in Fehlerbehandlung; einfach halten, auch testen.

Logging: Diagnostische Zwecke, Frameworks & verschiedene Log-Levels nutzen.

Error Handling Policy klar definieren: Welche Eingaben sind erlaubt, wie geht System mit unerlaubten Eingaben um? Exception Policy: Benutzen? Wenn ja, wie? Assertions Policy: Benutzen? Wenn ja, für was? Wann einschalten? Logging Policy: Benutzen? Wie detailliert? Welche Levels werden benutzt?

Assertions: Für Programmierfehler, die nie auftreten sollen. Kann man auch nur als formale Kommentare betrachten. Abschalten in Production und Nutzen in Tests ist wie Schwimmübung auf Land mit Schwimmweste und dann ohne Schwimmweste ins Wasser gehen…

# Design by Contract

Beziehung zwischen Komponentendurch Verträge geregelt. Nützlich, wenn pragmatisch angewandt. Contracts für Interface-Methoden angeben. Deklarativ statt imperativ wie Unit Tests.

Vorteile: Fehler werden früher gefunden, Schuldiger schneller klar, Tools: C4J (Contracts for Java), Confoja.

Preconditions: Muss vor dem Aufruf erfüllt sein; Aufrufer ist verantwortlich. Nie in Methode selbst prüfen!

Postconditions: Nach dem Aufruf gültige Bedingungen; Implementation der Methode ist verantwortlich. Unterscheiden zwischen Normal und Exceptional Postconditions, wenn Methode Exceoptions schmeisst. Nur Post Conditions für Rückgabewerte.

Postconditions, die für alle Methoden gelten, als Klasseninvariante definieren (DRY). Gelten immer, d.h. nach Aufruf jeder Methode.

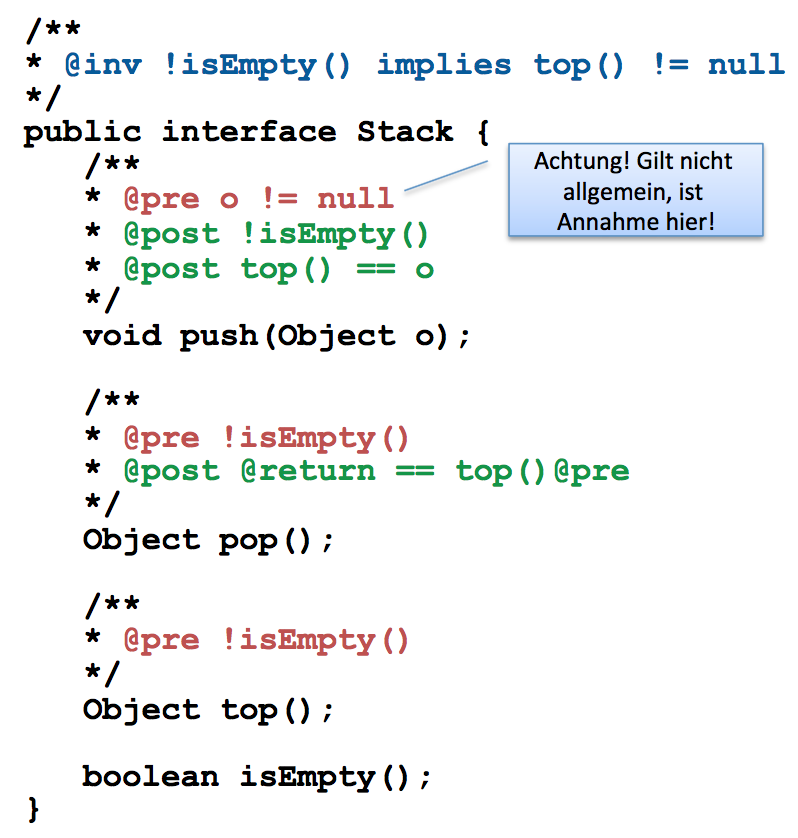
Vererbung: Subtyp darf Preconditions nur lockern, Postconditions und Invariante nur verschärfen.

Pure Method: Methode wird ohne Side Effects ausgeführt.

TDD mit Contracts: Modular, lose Kopplung, kein unnötiger Code, zur Laufzeit überprüfbare Spezifikation.

Es reicht aus, nur die wichtigsten Kernklassen mit Contracts zu schützen. Die meisten Bugs sind auf Verletzungen von Invarianten und Preconditions zurückzuführen. Grund waren falsche Annahmen über Nutzung der Klassen.

DBC ist gute Dokumentation; sorgt für klare Verantwortlichkeiten; gute Ergänzung zu Unit Tests.



Contracts dürfen nur Pure Methods benutzen. Die Pure Methods müssen identifiziert und als solche deklariert werden (z.B. via @Pure).

# Usability Testing

Software testen, nicht den User. Kurze Einführung; Aufgaben stellen; eventuell zwischendrin Fragen stellen.Nicht helfen, nur beobachten.

Während Construction-Phase, idealerweise mehrmals.

# Metriken

## Source Code Metriken

Zyklomatische Komplexität: M = E – N + 2P

E = Anzahl Kanten (Edges), N = Anzahl Nodes, P = Anzahl verbundener Komponenten (i.d.R. 1). Einfacher: Anzahl if / while / for / case + 1 = min. notwendige Tests für 100% Coverage. Sollte < 10 sein.

Umfangmetriken: Anzahl Codezeilen.Metriken für Kohäsion und Koppelung: NOC: Number of Classes.DIT: Depth of Inheritance Tree; wie tief wird vererbt. Normal 2-3 aber nicht mehr als 4 Stufen.

NOI: Number of Interfaces. NOM: Number of Methods (ideal 5-10).

TLOC: Total Lines of Code. NORM: Number of Overridden Methods; je mehr desto komplexer der Code. LOCOM: Lack of Cohesion; wie viele Methoden benutzen die Felder, je mehr desto höher die Kohäsion. (Anzahl Methoden – Mittelwert von Anzahl Methoden die auf Attribut zugreifen) / (Anzahl Methoden – 1).

Instability:Hat viel Abhängigkeiten zu anderen Klassen aber nicht viel ist von meiner Klasse abhängig. Stabil = viel ist von mir Abhängig aber weniger Abhängigkeiten zu anderen Klassen.

Tools: metrics2, Eclipse MEtrics, Structure 101, STAN, SonarQube, Scrutinizer.

# Performance Profiling

'Premature Optimizations' sind kontraproduktiv. Zuerst messen, dann optimieren.

Source Code Instrumentation: NetBeans, Remote Profiling von JVM: JProfiler, New Relic.

Last-Messungen (Black Box) z.B. mit JMeter.

Wiederholte Messungen durchführen, für Antwort auf Fragen à la "Sind die typischen Anwortzeiten immer noch etwa gleich?". Lösbar mit Headless Browser; Selenium.

Monitoring mit Dashboards: 3 Zielgruppen: Business, Application, Server Infrastructure.

Application Telemetry: Möglichst viele Daten an Dashboard senden (nicht loggen). Beispiel: Flugzeug sendet in Echtzeit an Boden.

# Software-Architektur

Architektur ist die Summe der Design-Entscheide, die von grosser Tragweite sind, und die länger leben. Beispiele: Interfaces zwischen Subsystemen, Datenstrukturen, Exception Handling, usw.

Systemübergreifende Überlegungen; Separation of concerns.

SW-Architektur-Dokumentation beschreibt, *wie* System so gebaut wurde und *warum*. In Requirements/Analyse steht, *was* Kunde wollte. Somit grob 2 Teile: Umfeld und Randbedingungen, sowie technische Struktur.

# Reviews

Reviews sind sehr kosteneffizient; man findet schnell viele Fehler. Zudem lernt Team Code besser kennen und programmiert besser.

Vorteile: Einheitlicher Coding Style, sowie Metriken & Coverage Reports helfen bei Code Reviews. Unterschiedliche Formalitäten: Pair Programming, Pull Requests, Grouo Reviews, Formal Inspection, Design Review. Prinzipiell immer Peer Reviews (Arbeitskollegen), nicht von oben herab (Audits).

Requirements Review: Requirements sollen Ansprüchen von QA und Testing genügen und Knowhof-Transfer ermöglichen. Beispiel-Frage: Stimmen Activity-Diagramme mit Use Cases überein?

Architektur Review: Nicht-funktionale Anforderungen prüfen und ob Doku mit tatsächlicher Implementation übereinstimmt. Beispiel-Frage: Wie skalieren, wenn Besucherzahl plötzlich viel höher?

# Fehlertheorie

Zufällige Fehler (eher Hardware);

Systematische Fehler (eher Software).

## Fehler verhindern

Statische Techniken: Code Review, Design Review, Coding Guidelines, Formale Methoden.

Dynamische Techniken: Testen, Bounds Check

Prozess Basierte Techniken: Pair Programming, Pull Request

Ursachen für Fehler: Feature creep, Komplexe Designs, Zuerst ausliefern dann Fehler beheben Mentalität. Mangel an Sicherheit, Kein klarer Entwicklungsprozess, schlechte Tools, fehlende Finanzielle Mittel.

# Code Smells

## Black Sheep (8, 16)

Klasse oder Methode passt nicht in Hierarchie, weil in vielen Ansichten anders als Basisklasse.

## Comment

Kommentare nur für Infos brauchen, die der Code (+ Tests) nicht selbst schon aussagt. Sollen nicht sagen, was der Code macht, sondern z.B. warum es so ist.

## Conditional Complexity (18, 19)

Zu viele (verschachtelte) Conditions machen den Code schwer lesbar.

## Dead Code

Nicht erreichter Code, der problemlos gelöscht werden kann.

## Duplicated Code (8, 9)

Gleicher / sehr ähnlicher Code an mehreren Stellen.

## Feature Envy (8, 16)

Verantwortlichkeiten am falschen Ort, z.B. wenn eine Methode zu sehr von einem anderen Objekt abhängig ist. Sollte dann eher Methode der Klasse dieses Objekts sein.

## Inappropriate Intimacy (15, 16)

Klassen-Interna unnötigerweise nach aussen sichtbar; *public* statt *private*. Oder eng gekoppelte Klassen (bidirektionale Abhängigkeit).

## Inefficient Name

Namen entspricht nicht dem, was die Methode wirklich macht.

## Large Class (7, 9)

Zu viele Methoden und Instanzvariablen, somit zu viele Zeilen Code. Macht mehr als nur 1 Job (Single Responsibility Principle verletzt).

## Lazy Class (11)

In etwa Gegenteil von Large Class: Sehr klein, macht zu wenig; könnte genauso gut Teil einer anderen Klasse sein. Ausser sie ist neu und „wächst“ noch.

## Long Method (8, 26)

Nicht schnell und einfach erkennbar, *was* eine Methode *wie* macht. Schwer zu lesen und zu testen. Kurz gesagt, Methode macht zu viel.

## Long Parameter List (8, 13)

Zu viele Parameter, schwierig zu lesen, mühsam zu implementieren.

## Magic Numbers

Scheinbar beliebige Zahlenwerte verteilt in Codebasis. Lösung: Konstante einführen und diese gut benennen.

## Middle Man

Wenn eine Klasse nichts Anderes macht als Arbeit an eine andere Klasse zu delegieren. Entfernen, Methoden direkt aufrufen.

## Oddball Solution (aka Inconsistent Solution) (8, 9)

Wenn ein Problem an mehreren Orten unterschiedlich gelöst wird. Etwa wegen Ignoranz oder zu wenig Refactoring.

## Primitive Obsession (13, 27, 28)

Zu viele low-level Methoden und und übermässige Nutzung primitiver Datentypen. Beispiel: *indexOf()* statt *foo.contains()*.

## Refused Bequest (9, 22)

Wenn eine Subklasse nicht das Interface seiner Elternklasse unterstützt. Beispiel: Method Overriding mit einer „Do nothing“ Implementation.

## Side Effect (8, 17)

Methoden die nicht nur das Thun was man von Ihnen erwartet. Beispiel: next() iteriert *und* returned.

## Shotgun Surgery / Solution Sprawl (15, 16)

Lösung für ein Problem ist über zu viele Klassen verteilt.

## Speculative Generality (4, 8, 11)

Code zu abstrakt oder gar nicht benutzt, weil man denkt, es wird in Zukunft gebraucht. Beispiel: Abstrakte Klasse oder Interface, obwohl es nur 1 implementierende Klasse gibt.

## Switch / Case (8, 12, 27, 28)

Wenn gleiches Switch Statement mehrmals vorkommt, ist polymorphe Lösung mit Subklassen statt switch zur Unterscheidung oft besser geeignet.

Auch: Zu viele else if Clauses, denn diese sind i.d.R. zu wenig kompliziert für einen Conditional Complexity Smell.

## Temporary Field (9, 12)

Fields, die nur zeitweise gebraucht warden, sonst aber leer/unbenutzt sind. Zeichen dafür, dass Klasse zu viel macht.

# Refactoring (\* = Refactoring to Pattern)

1. Change Bidirectional Association to Unidirectional

Assoziation nur noch in eine Richtung.

1. Change Method Signature

Methode braucht mehr Infos: Parameter hinzufügen. Parameter nicht länger gebraucht: Entfernen. Evtl. Parameter Object nutzen.

1. Change Value to Reference (oder umgekehrt)

Klasse hat Value Object als Instanzvariable, ist aber ein Domainobjekt; bzw. Klasse hat Reference Object als Instanzvariable, welches klein und immutable ist. Value Objekt zu Reference Objekt (oder umgekehrt) machen.

1. Collapse Hierarchy

Basisklasse mit Unterklassen vereinigen wenn sie sich nur geringfügig unterscheiden.

1. Decompose Conditional

Kompliziertes if-else. Extract Method für Bedingung, then-Block

und else-Block.

1. Encapsulate Field

Getter & Setter mit privatem Attribut statt public Attribut.

1. Extract Interface

Mehrere Clients verwenden gleiches Subset eines Interfaces? Subset in eigenes Interface extrahieren.

1. Extract Method, Inline Method\*

Mache aus Fragment eine Methode, Name kennzeichnet Zweck. Aus einer langen Methode mehrere kleine machen.

1. Extract Class/Subclass/Superclass

2 Klassen mit ähnlichen Eigenschaften? In neue Basisklasse resp. In Subklasse verlagern, wenn nur von einigen Instanzen verwendet.

Oder allgemein neue Klasse.

1. Extract Variable, Inline Temp

Speichere Ausdruck (oder Teile) in temporärer Variable, Name erklärt Zweck. Inline Temp: Umgekehrt, da sich Variable nicht lohnt.

1. Inline Class

Klasse macht zu wenig, Existenz nicht berechtigt. In andere Klasse integrieren.

1. Introduce Null Object\*

Es finden wiederholt Überprüfungen auf null statt. Ersetze null mit einem speziellen Null Object.

1. Introduce Parameter Object

Mehrere Parameter ersetzen durch Parameter Object (z.B. DateRange statt *start* und *end*)

1. Move Creation Knowledge to Factory\*

Code zum Instanzieren einer Klasse ist über mehrere Klassen verteilt? Eine einzige Factory-Klasse einführen.

1. Move Field (incl. pull up/push down)

Eine andere Klasse nutzt ein Attribut mehr als die eigene. Erzeuge ein neues Attribut in der Ziel-Klasse und passe die Nutzer an.

1. Move Method (incl. pull up/push down)

Feature Envy. Verschiebe Methode in andere Klasse. Delegiere in alter Methode an neue Methode oder lösche sie.

1. Rename Method/Field/Parameter

Methode umbennen, damit Name besser den Zweck beschreibt.

1. Replace Conditional Dispatcher with Command\*

Command für jede Aktion erstellen und in Collection ablegen. Commands darin suchen statt Conditional Logic.

1. Replace Conditional Logic with Strategy\*

Condition entscheidet, wie etwas berechnet wird? Strategy für jede Variante einführen und Berechnung an Strategy-Instanz delegieren.

1. Replace Constructor with Factory Method

Wenn Erzeugung eines Objektes zu komplex ist.

1. Replace Exception with Test

Aufrufer soll Bedingung prüfen, damit keine Exception nötig ist.

1. Replace Inheritance with Delegation

Wenn Subklasse Interface der Basisklasse nicht unterstützen will.

1. Replace Magic Number with Symbolic Constant

Ersetze den Literal mit einer symbolischen Konstante, deren Name die Bedeutung widerspiegelt.

1. Replace Nested Conditional with Guard Clauses

Geschachtelte Bedingungen machen Normalablauf unübersichtlich. Verwende für Spezialfälle Guards und entsprechende Methoden.

1. Replace State-Altering Conditionals with State\*

State-Klassen einführen, welche die verschiedenen States und die Transitions zwischen ihnen handhaben.

1. Replace Temp with Query

Extrahiere Ausdruck in Methode, ersetze Referenzen auf Variable durch Methodenaufruf.

1. Replace Type Code with Class/Subclass (Polymorphism)

Ersetze Typcode durch Enum-Klasse bzw. Subklasse, wenn Typcode Verhalten beeinflusst.

1. Replace Type Code with State/-Strategy

Klasse verwendet veränderlichen Typcode, der Verhalten der Klasse beeinflusst. Ersetze Typcode durch State/Strategy Objekt.