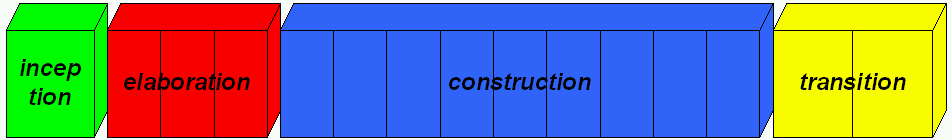
# Projektplanung

Agil (Scrum; extreme programming XP)  
Iterativ (RUP, Iterative Development, Spiral Development...)



RUP (Rational/IBM Unified Process):

4 Phasen x Iterationen à 2-3 Wo. Bei jedem Sprint soll der Gesamtwert für den Kunden steigen.

Elaboration: typ. 15-25% der Zeit. Construction: typisch deutlich über 50% der Zeit (gezählt ab Kickoff, wo Budget alloziert wurde).

Achtung: Prozent der Zeit ≠ Prozent der Kosten

Wichtig: Chief Architect von Kickoff bis mindestens Mitte Projekt involviert.

## Checkliste End of Elaboration

Requirements erfüllt? Scope abgedeckt (Use Cases)? UI Entwürfe? Prototype (möglichst alle Schichten)? Software Architektur Entwurf? Werkzeuge & Methoden definiert? Liste der Arbeitspakete.

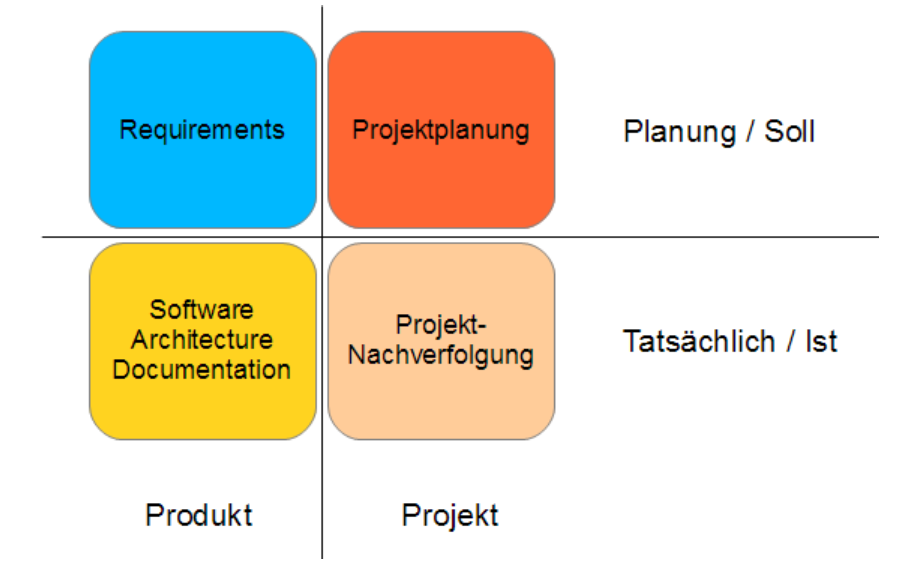
Use Cases beschreiben nur die Funktionalität. Nicht-funktionale Anforderungen ergänzen Use Cases und Domainmodell.

Anforderungen -> Arbeitspakete (JIRA Tickets)

Kurzbeschrieb, Akzeptanzkriterien, Aufwandschätzung, Prio, usw. Maximal halb so gross wie das, was eine Person in einer Iteration schafft.

Pro Iteration genug Arbeitspakete; eigenverantwortliche Zuteilung. Entwickler schätzen Aufwand, Kunde priorisiert.

4 Quadranten der Dokumentation



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

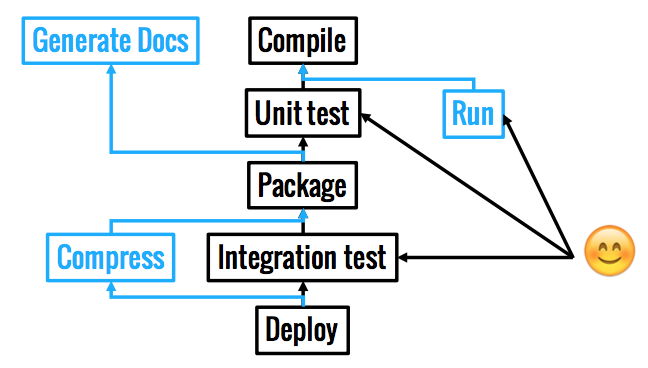
Vor *End of Elaboration* liegen die Haupt-Anstrengungen bei der Dokumentation darauf, zu zeigen, dass man den Kunden verstanden hat. Danach liegt Fokus Umsetzen der Lösung, d.h. die Dokumentation, die entsteht, ist hauptsächlich für die Entwickler.

# 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 Dep. Mgmt, aber weniger flexibel.

## Continuos Integration

Rules: Build automatisieren (schnell!), Build ist selbst-testend, kleine Commits, jeder Commit = Build.

Testumgebung möglichst wie Production.

Automatisierte Deployments, Test Coverage, Build History.

# Software Engineering Practices

## Requirements Practices

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

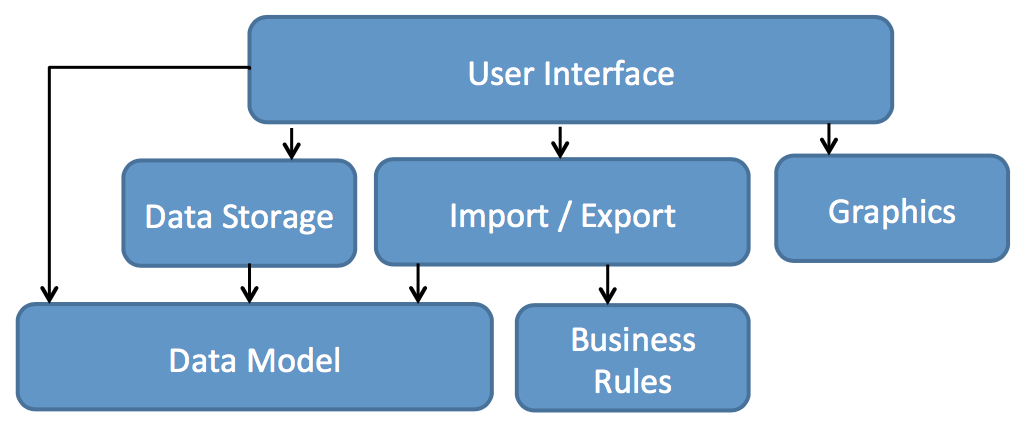
Qualität als Requirement: Möglichst testbare Qualitätsanforderungen, basierend auf echten Anforderungen; wichtig aber auch schwieriger zu ermitteln.

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.

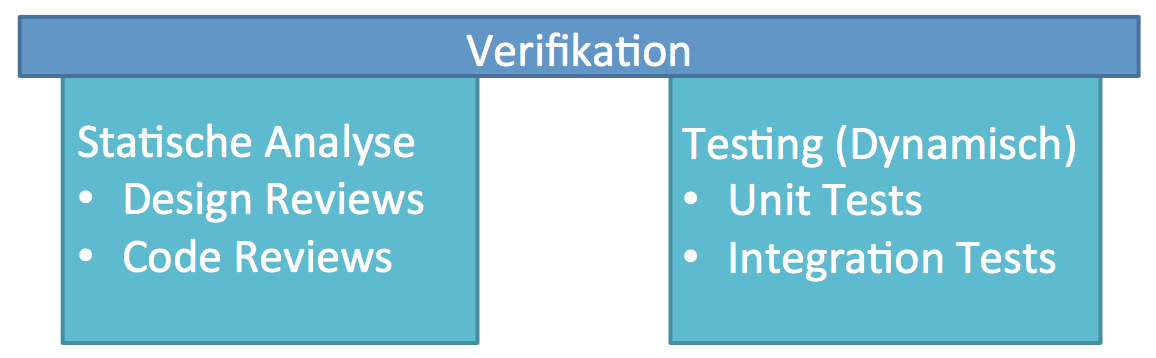
Refactoring early & often: Konstanter Verbesserungsprozess während Projekt. Liste von zu verbessernden Bereichen führen. Tests *vor* Refactoring haben. Mehrere kleine Schritte.

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.



# 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?

# Design by Contract

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

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.

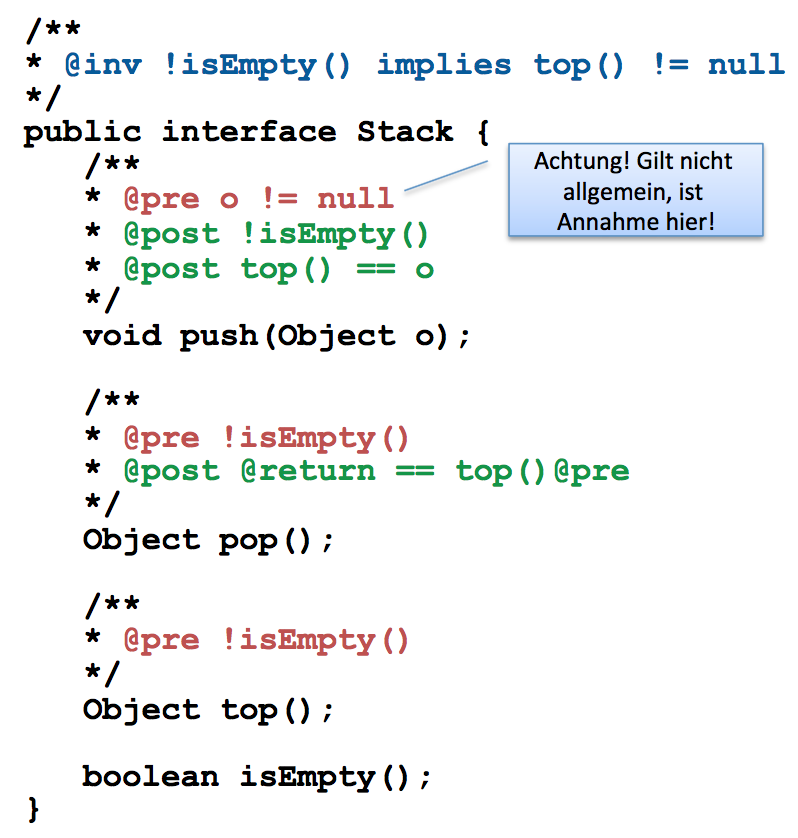
Postconditions, die für alle Methoden gelten, als Klasseninvariante definieren (DRY). Gelten immer, d.h. nach Aufruf jeder Methode.

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

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.



# Usability Testing

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

# 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. **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.

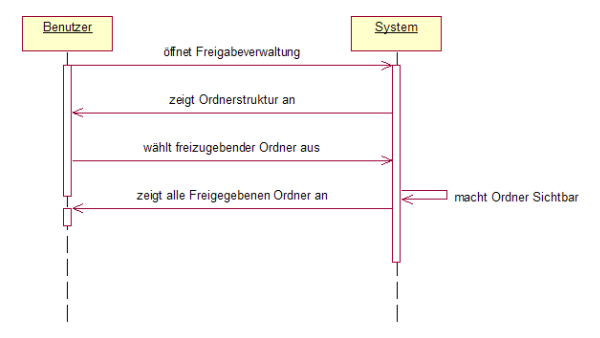
Epics (≈ Use Cases) > User Stories > Tasks

# 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.

# Metriken

Komplexität: M = E – N + 2P; wobei E = Anzahl Kanten (Edges), N = Anzahl Nodes, P = Anzahl verbundener Komponenten (i.d.R. 1).

Einfacher, falls sich Edges nicht überkreuzen: Anzahl “Regionen” zählen, in die der Hintergrund durch die Kanten unterteilt wurde. Oder: Anzahl if / while / for + 1.

Faustregel: Methoden mit Komplexität > 10 genauer anschauen.

# 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.

# 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.

# Code Reviews

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

Einheitlicher Coding Style, sowie Metriken & Coverage Reports helfen bei Code Reviews. Unterschiedliche Formalitäten: Pair Programming, Pull Requests, Grouo Reviews, Formal Inspection.

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 tasächlicher Implementation übereinstimmt. Beispiel-Frage: Wie skalieren, wenn Besucherzahl plötzlich viel höher?

# Aufwandschätzung

Keep it simple; Komplexität tief halten. Faustregel: Kein Projekt Machen über 9 Monate oder 1 Million. Aufteilen und separat planen, ausführen und abliefern (siehe Story Splitting). Schätzen ist Erfahrungssache. Aufwand is hauptsächlich bestimmt durch: Grösse (Funktionalität), Art der Software (Komplexität), Qualität der Mitarbeitenden.

## 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.

# Code Smells

## Duplicated Code

Entweder exakt gleicher Code an mehreren Stellen oder sehr ähnlicher Code, welcher dasselbe macht.

## 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

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

## Feature Envy

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

Klassen-Interna unnötigerweise nach aussen sichtbar. *public* statt *private*.

## Large Class

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

## Lazy Class

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

Wenn man nicht schnell und einfach erkennen kann, was eine Methode macht. Schwer zu lesen und zu testen.

## Long Parameter List

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

Ersetzen durch Parameter Object (DateRange statt *start* und *end*) oder in mehrere Methoden aufsplitten.

## Magic Numbers

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

## Middle Man

Wenn eine Klasse nichts anderes macht als Arbeit an eine andere Klasse zu delegieren.

## Oddball Solution (aka Inconsistent Solution)

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

## Primitive Obsession

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

## Refused Bequest

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

## Shotgun Surgery / Solution Sprawl

Lösung für ein Problem ist über zu viele Klassen verteilt

## Speculative Generality

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

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.

# Refactoring

## Change Method Signature

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

## 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.

## Collapse Hierarchy

Basisklasse mit Unterklassen vereinigen wenn sie sich nur geringfügig unterscheiden.

## Decompose Conditional

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

und else-Block.

## Encapsulate Field

Getter & Setter mit privatem Attribut statt public Attribut.

## Extract Interface

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

## Extract Method, Inline Method

Mache aus Fragment eine Methode, Name kennzeichnet Zweck.

## Extract Subclass/Superclass

Zwei Klassen haben ähnliche Eigenschaften? In neue Basisklasse verlagern. Funktionalität wird nur von einigen Instanzen verwendet? Neue Subklasse einführen.

## Extract Variable, Inline Temp

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

## Introduce Null Object

Es finden wiederholt Überprüfungen auf null statt.

Ersetze null mit einem speziellen Null Object.

## Move Creation Knowledge to Factory

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

## 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.

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

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

## Rename Method/Field/Parameter

Methode umbennen, damit Name besser den Zweck beschreibt.

## Replace Conditional Logic with Strategy

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

## Replace Constructor with Factory Method

Die Erzeugung eines Objektes ist komplex? Ersetze Constuctor mit Factory Method.

## Replace Exception with Test

Eine Methode wirft eine Checked Exception aufgrund

einer Bedingung, die der Aufrufer überprüfen könnte.

Der Aufrufer soll die Bedingung prüfen und die Exception

verhindern.

## Replace Inheritance with Delegation

Wenn Subklasse Interface der Basisklasse nicht unterstützen will.

## Replace Magic Number with Symbolic Constant

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

## Replace Nested Conditional with Guard Clauses

Geschachtelte Bedingungen machen Normalablauf

unübersichtlich. Verwende für Spezialfälle Guards und entsprechende Methoden.

## Replace State-Altering Conditionals with State

Wenn Conditionals, welche Zustandsübergänge des Objekts kontrollieren, zu komplex sind. State-Klassen einführen, welche die verschiedenen States und die Transitions zwischen ihnen handhaben.

## Replace Temp with Query

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

## Replace Type Code with Class/Subclass (Polymorphism)

Klasse verwendet einen Typcode, der das Verhalten der Klasse

nicht beeinflusst. Ersetze den Typcode durch eine Klasse (enum). Ersetze durch Subklasse, wenn Typcode Verhalten beeinflusst.

## Replace Type Code with State/-Strategy

Klasse verwendet einen veränderlichen Typcode, der das

Verhalten der Klasse beeinflusst. Ersetze den Typcode durch ein

State/Strategy Objekt.