|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Projektplanung (Inception, Elaboration, Construction, Transition)** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Ablauf | | | | | | | Projektplanung/Requirements 🡪 Arbeitspakete, Bugs, SAD 🡪 Alles Archivieren und Ergebnis zeigen | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Dokumente | | | | | | | SRS(UC, NFR), SAD, Testprotokoll, Benutzerhandbuch, Installationsanleitung, IT Landschaft, Schnittstellen | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| End of Elaboration (Wendepunkt) | | | | | | | Kunde wurde verstanden: \*Requirements,: Scope mit UCs, Domain Model, NF Anforderungen \* Entwürfe des GUIs \* Architektur Entwurf mit Prototyp \* Entwicklungsumgebung \*Aufwandschätzung mit Arbeitspaketen | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Diagramme | | | | | | | Dienen zur Kommunikation (formaler als Worte), normiert (UML), maximal A3 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Merksatz | | | | | | | "So früh wie möglich so formal wie möglich" "So früh wie möglich so komplett wie möglich" | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| |  |  | | --- | --- | | Inception | Vision, Projekt-Eckwerte, IT-Landschaft, Annahmen&Einschränkungen | | Elaboration | Use Cases + Diagramm, Wireframes, NF Anforderungen, Domain Model, Glossar, Liste v. Stackeholder Architekturentscheidungen, Schichtendiagramm, Meilensteine, Testplan, Risikoliste, Zeitaufschreibung | | Construction | Deployment Diagram, Datenmodell, Metriken, Test Protokolle | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Requirements Best Practices**  (testbar und überprüfbar (min,max,avg) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1. Herausfinden der echten Kundewünsche durch kritisches Hinterfragen, Befragungen und Beobachtungen  2. Qualität als (NF-) Requirement aufnehmen (Performance, Scalability, Security, Robustness)  3. Mit Veränderungen umgehen. Software ändert sich ca. 2% pro Monat (**Design for Change**, Kurze Iterationen) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Design Best Practices** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1. DRY : Don’t Repeat Yourself : Repetition vermeiden, Keine Code/Documentation Duplication 🡪 Gefahr von Inkonsistenzen  2. Keine Kopplung zwischen konzeptionell unabhängigen Aspekten (Orthogonalität: Verschiedene Objekte sollen unabhängig sein)  3. Hierarchische Zerlegung in Komponenten, Packages. Abhängigkeiten zwischen Komponenten reduzieren. Zyklen vermeiden!  4. Design to Test : Testbarkeit hat Einfluss auf die Architektur | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Implementation Best Practices** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1. Fix broken windows : Probleme beheben wenn sie entstehen.  2. Refactor early and often : Bestehenden Code verbessern ohne die Funktionalität zu verändern. Liste von offenen Issues führen.  3. Bewusst programmieren : Vermeide Programming by Coincidence. Klares Ziel verfolgen. Annahmen mit Tests dokumentieren. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Verification Best Practices** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1. Tests : Früh, häufig und automatisch testen. Code Coverage überwachen. Realistische Testdaten verwenden. Integrationstests | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 2. Reviews in Sitzungen oder selbständig durchführen. Findings festhalten. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Projektmanagement** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Technical Debt** (Quick and dirty: Long after quick is gone, dirty remains.) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Möglichen Konsequenzen schlechter Umsetzung von Software. Den zusätzlichen Aufwand, den man für Änderungen einplanen muss. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1. Code-Smells Architektur-Smells 2. fehlende Unit Tests 3. Qualitätsmängel (Performance/Skalierbarkeit, Security Flaws…)  4. "Keine Zeit" → Prozesse werden nicht mehr beachtet (Keine Code Reviews mehr, Schätzung von Issues nicht mehr verifiziert…) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Die Baustelle der Programmierer für einen nicht-programmierenden Projektleiter** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| sichtbar: Backlog, Issue/Bug Tracking, Test-Vorlagen, Test-Protokolle | | | | | | | | | | | | | | teilweise sichtbar: Build Server Output  (Build Fails, Testabdeckung, metrics) | | | | | | | | | | | | | | | | | | nicht sichtbar: Programmcode, Versionkontrolle, Konfigurations-Dateien | | | | |
| **Unsichtbares sichtbar machen** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| für Kunden und Management: Burn-Down Chart, Story Map mit Farben, Trends von Metriken  für Entwickler: zusätzlich Backlog, Metriken, Testabdeckung, SonarQube  für IT Infrastruktur: Dashboards mit Ressourcen-Verbrauch (Disk, CPU...), Log-Auswertungen visualisiert (Trends sind besser) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Projektautomation** Compile (alle 1’), Unit test (3’), Package/bundle (1h), Integration Test (1h), Deploy (?h-y) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **CRISP** | | Complete/Repeatable/Informative/Schedulable/Portable (Flexilibity (z.B. exec only Test), Performance, Extensilibity) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Build Scripts** | | **Vorteile:** Automated, Repeatable, Schedulable **Nachteile:** non-interactive, schwierige Maintenance, Platform-dependent | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **GNU Make** | | Imperativ, plattformabhängig, kein Dep. Mgmt, Datei wird in Targets aufgeteilt, welche parallel laufen können | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Apache Ant** | | Imperativ, xml-based scripting, portalbe, complex, kaum reuse, viel copy-paste | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Maven** | | Deklaritv (pom.xml), vorgegebene Lifecycles und Dateistruktur, Dep. Mgmt | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Continuous Integration:** Fortlaufendes Erstellen von Software aus mehreren Komponenten (Basis für Continuous Delivery) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Workflow** | | Compile > Unit Test > Package > Integration Test > Deploy (*Comit early and often, make small changes*) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Vorteile** | | Weniger Integrationsprobleme (Merge Conflict, Hardware Probleme), Automatisches Testausführung, Einchecken nur möglich wenn wenn Code ohne Error und Warnungen, Generieren und Darstellen von Metriken (Sonarqube), Bugs werden schneller gefunden, Visualisierung des Projekts mit Metriken, Versionierung von Builds, | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Best Practice** | | Single SRC Repo with Git Flow, Build automatisieren (Maven), Auto. Testing (Unit&Integration), At least daily commits, Ever Commit Builds, Fast build (<10m), Test on clone of Prod, Download latest Exec, Reports, Auto. Deployment | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Features** | | Quality Analysis (Metrics, Static analysis, Test coverage, build history) IDE Integration, Review Functionality | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **C. Delivery** | | Beim Continuous Delivery wird die geteste Software in sehr kurzen Zyklen auf den produktiven Server deployed. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **SCRUM** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Voraussetzungen | | | | | | Kunde weiss noch nicht genau was er will, kein klar definierter Scope/eher experimentell, ganzes Team an einem Ort, Team deckt alle Fähigkeiten ab, häufige Kommunikation, Kunde im Team/ständig verfügbar. Issues werden eigenverantwortlich zugeordnet, Max 10 Personen Teams | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Gefahren/Nachteile | | | | | | Kein Checkpoint End of Elaboration, “übertrieben agil” (kein Plan, keine Architektur), höhere Abstraktionsstufe zu User Stories im Backlog fehlt (NFR, Use Cases, Domain Modell etc) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| PO (commited) | | | | | | Schnittstelle zum Kunden, Kennt das Buisness | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| PL (involved) | | | | | | Stundenkontrolle, IST/SOLL Vergleich, Risiko Beobachtung, Stakeholderanalyse, Verhandelt mit Kunden | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Sprints | | | | | | Dauern 2-3 Wochen. Inkrementelle Ablieferung mit **Feedback vom Kuden**. Bei jedem Sprint soll der Gesamtwert für den Kunden steigen. 🡪 Minimum Viable Product(MVP). | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Arbeitspakete | | | | | | Arbeitspakete im Sprint Backlog habe eine Grösse von durchschnittlich 2-4 Arbeitstagen. Entwickler schätzen, Kunde priorisiert! | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Story Points | | | | | | Schätzungen auf Studen sind schwierig und basieren oft auf der Erfahrung des schätzenden Entiwkcler.  Story Points sind eine relative Grösse über die Geschwindigkeit des Teams (Erfahrung über das gesamte Team) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Product Backlog | | | | | | Bald zu bearbeitende Stories sind detailiert beschrieben | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Impediments | | | | | | Hindernis, die nicht innerhalb des Scrum Teams beseitigt werden kann. Gründe/Auslöser ausserhalb Team | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Story Splitting von Projekten und Arbeitspaketen** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Kein Projekt über eine Million Budget, Kein Projekt länger als 9 Monate! (Max 10Pers im Team) 🡪 ansonsten splitten nach Kriterien | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Arbeit im Team aufteilen** 🡪 Architektur dem gesamten Team klar und Kunde vollumfänglich verstanden (*Erst nach End. Elaboration*) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Kunden-Domäne** | | | | Einzelne Teilbereiche umsetzen | | | | | | | | | | | | | | | **Geschäfts-Prozessen** | | | | | | | | | | | Einezlne Use-Cases definieren | | | | | | |
| **Domain Model** | | | | Konzept-Gruppen einzeln umsetzen | | | | | | | | | | | | | | | **geografisch** | | | | | | | | | | | z.B einzelne Kantone einzeln umsetzen | | | | | | |
| **Rollen** | | | | Teillieferungen für Spezielle Rollen (z.B Produktmanager, Einzelkunde, Marketing) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Aufteilung von zu grossen Arbeitspaketen (Aufteilung im Kleinen) 🡪 Erst einmal alles «nice to have» weglassen. (MVP) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Ideale Grösse von Arbeitspaketen** | | | | | | | Durchschnitt | | | | | was 1 Person in ¼ eines Sprints schafft | | | | | | | | | | | | | | | | | | | | | | | | |
| Maximum | | | | | was 1 Person in 50%-70% eines Sprints schafft | | | | | | | | | | | | | | | | | | | | | | | | |
| Fallstrike | | | | | Zu generisch, auch unproduktive Tätigkeiten einplanen (Doku, Server aufsetzen) | | | | | | | | | | | | | | | | | | | | | | | | |
| **User Stories** | | | | | | | sind umsetzungsorientiert, sind sehr kurz, oft kein Kontext, gruppiert in Epics (10-15 US), unterteilt in Subtasks | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| As an [actor], I want [action] so that [business value] | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Bestandteile: ID, Titel, Beschreibung, Akzeptanzkriterien, Story Points, Kunden Prio, gel. Stunden, Status, Kat. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Use Cases (Sea Lvl)** (Funktionale Anforderungen) | | | | | | | Bei Use Cases geht es darum, Interaktionen von Nutzern mit einem System so zu beschreiben, dass das Ziel der Nutzer klar wird. Nutzer können dabei sowohl Menschen als auch andere technische Systeme sein. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| sind kundenorientiert ohne Blick auf Implementierung, sind sehr detailiert (fully dressed), Bilden oft ganze Geschäftsprozesse ab, Beschreiben den Kontext, wiederspiegeln die Benutzersicht. (ca. 8-12 Epics = 1UC) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Aufwandschätzung** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Erfolg ist extrem grössenabhängig, vA wegen Scope Creep, mehrere Mitarbeiter, ungenaue Angaben. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Brooks Law | | | | Adding man power to a late project, makes it later (Einarbeitungszeit 2-3 Monat) 🡪 besser spliten und separat Dev | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Einflussfaktoren | | | | **1.** Grösse des Projekts und geforderte Funktionalität, **2**. Art und Komplexität, **3**. Qualität der Mitarbeiter | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Nicht lineare Faktoren. | | | | Kommunikation (n\*(n-1) /2, Requirements, Architecture, Management / Planung, Anforderungsanalyse, Testing | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Falsche Schätzungen | | | | politische Vorgaben dominieren ≠ realistische Vorgaben, zuviel Optimismus, fehlende Sorgfalt/mangelnde Qualität, Komplexität und unerwartete Nebeneffekte durchkreuzen Pläne, fehlende Erfahrung. Deshalb: Schätzen->Ist-Zahlen ermitteln -> Soll/Ist Vergleich | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Top Down** | | | | Gut für Schätzungen am Anfang des Projekts. Benutzen nur ganz wenige globale Parameter und ein paar Formeln und Annahmen (COCOMO und Funtional Points) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Bottom Up** | | | | Alle Requirements müssen bekannt sein- Gantt Charts, Jira Redmine | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Cocomo:** Constructive Cost Model | | | | model that allows one to estimate the cost, effort, and schedule when planning a new SE activity. **Parameter**: Project size, Complexity, Analyst Capability, Programmer Capability, Time Constraints, Fluktuation  Personen Monate PM = 3.2 \* (KDSI) ^ 1.05 (KDSI: K delivered source instructions, 1000 Zeilen Code) Entwicklungszeit = 2.5 \* PM ^ 0.38 **Vorgehen:** 1. Schätzung abgeben, 2. Projekt durchführen, 3. Parameter justieren | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Function Points** | | | | External Inputs : 4 External interface files: 7 Logical internal tables: 10  External outputs: 5 External Queries: 4  zB 25 Eingabemasken, 5 interface Files, 15 Reports, 10 external queries, 20 DB-Tabellen  25\*4+5\*7+15\*5+10\*4+20\*10 = 450 Function Points -> in Tabelle & Formeln nachschlagen = 82 PersonenMte | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **LoC pro Monat** | | | | 80-150 bei schwierigen Echtzeit Projekten, 300-500 im Schnitt mit guten nicht zu grossen Team, bis ca. 1000 LOC nur mit kleinen Spitzenteams- über 1500 LOC ist unglaubwürdig (Pfusch, viel Copy/Paste, generiert) **Ohne Erfahrungswert**: 400 DLOC pro Monat und Entwickler | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **DLoC** | | | | Delivered Lines of Code: Ohne Testcode, ohne Prototypen, hohe Qualität des Codes, kein Copy Paste | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Verlauf des Arbeitsfortschritts** (Sigmoid Kurve 🡪 **80/20 Regel**) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1. Schwieriges wird hinausgeschoben, 2. Funktionen verursachen an anderen Orten neue Probleme, 3. Neue Kundenwünsche kommen auf, 4. Fehleinschätzungen, 5. einfaches stellt sich schwierig heraus, 6. Refactorings | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Reviews** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Relative Kosten für Fehlerbehebung** | | | | | | | | | | | | | | | | | | **Regeln für Code Reviews**:  **1**. Vorbereiten / Einladen (inkl. Link zu Req. Specs, Docus)  **2**. Vorbedingungen checken (Guidelines, Code kompilliert?, Doku up to date?) **3.** Code Review durchführen (Zeile für Zeile, sachlich bleiben, max. 2-3h)  **4**. Nacharbeiten **5**. Allfällige Nachkontrolle  **Wichtigste Punkte:**  - Verständlichkeit (alle sollten Code verstehen)  - Namenswahl (Methoden, Variablen, packages) - Code Smells  - Übereinstimmung mit Architektur-Ideen und Diagrammen | | | | | | | | | | | | | | | | | | |
| **Vorteile** | | | | | | | | **1**. Fehler werden früh gefunden 🡪 Kosten **2**.Wissensaustausch über Architektur **3**. Team wird besser (Code) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Rollen** | | | | | | | | Modeartor, Author, Andere Entwickler (Peers), Note Taker | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Regeln** | | | | | | | | 3-5 Personen, Alles protokollieren, Author abwechseln, 500LoC = ca. 2h, NRF nicht vergessen | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Requirement Review** | | | | | | | | - Wissens-Transfer vom Kunden zum Entwickler | | | | | | | | | | | | | | | | | | | | | | | - Wichtig für Kostenschätzungen | | | | | |
| **Architektur Review** | | | | | | | | - Nicht funktionale Anforderungen - Kern-Charakteristiken müssen sichtbar sein - Architektur-Doku und Implementation müssen übereinstimmen - Szenarien durchspielen (z.B. für Erweiterbarkeit) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Metriken** Metriken + Belohnung = Desaster 🡪 Reviews wichtig! | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Projekt besser zu Verstehen, Verbesserungspotential sehen und die Qualität zu erhöhen (Technical Debt besser einschätzen) Zukünftige Projekte besser abschätzen, Effekt von Anpassungen bewerten, Kosten von Projekten zu sehen | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Indikatoren | | Grösste SRC-Files (LOC), Cyclomatic Complexiy, Code Guidelines Komformität, Number of Commits, Gefährliche Konstrukte, Duplicate Code, Kopplung/Kohäsion, Test Coverage, Lesbarkeit, Separation of Concerns, Gutes Naming | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Typen | | Umfangsmetriken (Anzahl Codezeilen, Anzahl Unterprogramme), Logische Strukturmetriken (McCabe, WMC), Datenstrukturmetriken (Anzahl Variablen & ihre Gültigkeit), Stilmetriken (Namenskonventionen), Bindungsmetriken [Kohäsion und Kopplung] (Anzahl Packages ausserhalb, von denen Klassen im Package abhängen → Efferent Coupling) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Tools | | **Dynamisch**: (Code ausführen) Covertura, EclEmma, Unit Tests, Integration Tests **Statisch**: Checkstyle, FindBugs, ReSharper, STAN, structure101, Metrik Anaylse (McCabe), Reviews, Formal Verification | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Tangles | | Zugriffe von unterem Layer auf obere Layer | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Zyklomatische Zahl**  (McCabe Metrik)  (auch Anzahl Testfälle für 100% Coverage) | | | | | Komplexität Logik von Graph *G* möglicher Abläufe, liefert Anzahl linear unabhängige Pfade v. Programm  *e* = Anzahl Kanten (**e**dges), *n* = Anzahl Knoten (**n**odes),  *p* = Anzahl Komponenten (com**p**onents)  Ganzes Programm: ***V(G) = e – n + 2p***  Einzelne Funktion: ***V(G) = e – n + 2***  **Annäherung**: *V(G) = nof Conditions(if,while,case,for) + 1*  *Kontrollflussgraph G:* Knoten für jede Anweisung, 1 Kante für normale Abläufe, 2 für if, while, for; switch wie if/else if/else…., letzte Anweisung in Schlaufe geht zurück zu Schleifenkopf/Condition  **Interpretation**: 1-10: Sehr Gut; 11-20: Evtl. verbesserbar, >20 Verbesserbar, >50 Schlecht | | | | | | | | | | | | | | | | | | |  | | | | | | | | | void zaehleZeichen(int &Vokalanzahl, int &GesamtZahl){  cin >> zchn;  while(zchn<=’A’) &&…{  Gesamtzahl ++;  if(zchn==’A’ || …){  Vokalanzahl ++;  }  cin >> Zchn;  }  return; | | | |
| **WMC** | | | | | Weighted Methods per Class | | | | | | | | | | | | Summe McCabe Cyclomatic Complexity für alle Methoden in Klassen | | | | | | | | | | | | | | | | | | | |
| **Lack of Cohesion of Methods**  (LCOM\*) | | | | | *m* = Anzahl Methoden; *m(A)* = Anz. Methoden, die auf ein Attribut zugreifen (Pro Attribut ein Wert)  ***LCOM\* = (m – Mittelwert(m(A)))/(m – 1)*** *🡪 Mittelwert m(A) = Anzahl Zugriffe / Anzahl Variablen*  **Interpretation**: *LCOM\* = 1* 🡪Schlechte Kohäsion *LCOM\* = 0* 🡪Maximale Kohäsion | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Afferent und Efferent Coupling**  Vektor | | | | | = Afferent Coupling:(Eingehend) Anzahl Klassen ausserhalb eines Packages, die von Klassen innerhalb abhängen.  = Efferent Coupling:(Ausgehend) Anzahl Klassen innerhalb eines Packages, die von Klassen ausserhalb abhängen.  = Abstractness: Anzahl abstrakter Klassen (innerhalb eines Packages) 🡪 1 = Viele / 0 = Keine | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Instability** I | | | | | 🡺 sollte gegen 0 tendieren. (Umsor kleiner I desto **stabiler** ist der Code 🡪 wenig Abhängikeiten) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Normalized Distance** from Main Sequence | | | | | , sollte möglichst gering sein bei gutem Package Design.  (1,1): **Zone of Uselessness**: Package mit lauter Abstrakten Klassen (Interface-Package) werden evt nicht verwendet  (0,0): **Zone of Pain:** Viele konkreten Klassen, mit vielen Abhängikeiten  Achsen: **I**nstability, **A**bstraction **Ziel**: Möglichst Nahe bei Main Sequence | | | | | | | | | | | | | | | | | | | | | | | | | | | | | Zones of Exclusion | | |
| **weitere Metriken** | | | | | NOC - Number of Classes, NSC - Number of Children (Anzahl direkter Unterklassen einer Klasse→ Klasse, die Interface implementiert, zählt als Unterklasse des Interfaces), NOI - Number of Interfaces, DIT - Depth of Inheritance Tree (Distanz der Klasse zur Klasse *Object* in Vererbungshierarchie), NORM - Number of Overridden Methods, NOM - Number of Methods, NOF - Number of Fields, TLOC - Total Lines of Code (zählt: {,}; !zählt: //bla, MLOC - Method Lines of Code (Zeilen innerhalb Methoden), Specialization Index: AVG(NORM \* DIT / NOM) A – Abstractness (Anzahl abstrakter Klassen / Anzahl Klassen eines Packages) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Software Architektur**: Architektur ist die Summe der Design-Entscheide, die von grosser Tragweite sind, und die länger leben | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Eigenschaften** | | | | Wiederverwendbar, austauschbar, testbar, einfach verständlich und damit gut dokumentierbar, stabil und langlebig, skalierbar durch versch. Deply Varianten, Gut aufgeteilt und somit parallel entwickelbar, kein Overengineering | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Entscheide** | | | | Statefull/stateless; singlethreaded/multithreaded; asynchron/synchron; FIFO/Priority Queue; P2P/Multi P. Broadcast | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Architektur Analyse** | | | | Umliegende Systeme, Aktoren, Schnittstellen, Randbedingungen (IT Landschaft), Nutzer, Rollen, Rechte, Security, Performance, Usability, Portability, UX Vorgaben | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Diagramme** | | | | Context Diagram (Grobe Übersicht der Systeme/Aktoren), Interface Diagramm, UC Diagramm, Aktivitätsdiagramm, Deployment Diagram, Package Diagram, Zustandsdiagramme, Prozesse/Threads, Message Queues | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Typische Architektur-Muster** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Single User Desktop Program  z.B. Word, Photoshop | | | | | | | | | | + UX, Performance, Sicherheit, Verfügbarkeit, geringe System-Komplexität  - Teamfähigkeit, Deployment/Cloud/Work Everywhere, Portierbarkeit | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Multi-Tier Architectures | | | | | | | | | | + Skalierbarkeit & Performance, Verteiltes & koordiniertes Arbeiten  - Performane (asynchron, nicht-deterministische Latenz), Verfügbarkeit, Sicherheit | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Fat Client & Server | | | | | | | | | | + User Experience (UX), Performance, Sicherheit - Deployment, Portabilität, Verfügbarkeit | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Mobile = «Fat Client» & Server | | | | | | | | | | Native Apps: +Besseres UX, +mehr Sicherheit; -Deplyoment, -Teuer da unportabel | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Thin Client: Browser-basiert & Server z.B. Load balanced Web | | | | | | | | | | + Deployment/Cloud/Work Everywhere, Last-Skalierbarkeit, Portabilität  - Performance, UX, Browser-Kompatibilität, Sicherheit, Verfügbarkeit, Testbarkeit | | | | | | | | | | | | | | | | | | | | | | | | | | |
| DB-zentrierte Architektur | | | | | | | | | | + Einfacher Aufbau, Erweiterbarkeit im Kleinen, Last-Skalierbarkeit  - Erweiterbarkeit (Wildwuchs), Hammer-Nagel-Syndrom, Wartbarkeit, DB-Vendor ausgeliefert | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Message-Queue basierte Systeme  Producer 🡪 Consumer | | | | | | | | | | + Skalierbarkeit, Performance, Separation of Concerns top umsetzbar  - Schwieriger Entwurf, schwierigere Fehlersuche (Logging hilft!) | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **NFR: Nicht Funktionale Anforderungen** Anforderungen an die Qualität (WIE) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Usability/Accessability, Availability, Performance, Interoperability, Secuirty, Speicherplatz, Anz. User, Erweiterbarkeit | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Vorteile | | | | Anforderungen an Architektur, Hardware, Usability/Accessability ist klar. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Design By Contract** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Precondition: | | | Bedingungen, die vor call erfüllt sein müssen (Aufrufer verantwortlich). Prüfe in Methode nie die Precondition | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Postcondition: | | | Bedingungen, die nach dem Aufruf erfüllt sein müssen | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Class Invariant | | | Garantien für Aufrufer, die immer(nach&vor Aufruf) gelten. Verantwortlich dafür ist Implementation der Klasse | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Beispiel Stack mit Cofoja 🡪🡪 Contracts als Spezifikationsmittel gehören zum **Interface** und sollte daher auch für Interface-Methoden definiert werden. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | @Invariant({})  public interface PrimitiveIntStack {  @Requires({"!isFull()"})  @Ensures({ "top() == value", “isEmpty()" })  void push(int value);  @Requires({"!isEmpty()"})  @Ensures({“!isFull()", "old (top()) == result" })  int pop();  @Requires({"!isEmpty()"}) @Ensures({})  int top(); } | | | | | | |
| **Behavioral Subtyping** (Subtyp muss mindestens Vertrag der Basisklasse erfüllen)  **-** Subtyp darf **Preconditions** lockern, aber nicht verschärfen ( ||-Verknüpfung )  - Subtyp darf **Postconditions** verschärfen aber nicht lockern( &&-Verknüfung)  - Subtyp darf **Invariante** verschärfen aber nicht locker ( &&-Verknüpf.) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **DbC Zyklus:**  **1. Declare:** Ergänze Interface um Deklaration einer neuen Methode  **2. Refactor:** Ergänze alle bestehenden Postconditions um Ausdruck basierend auf dieser neuen Abfrage. Auch Invariant  **3. Contract:** Formuliere Pre-/Postcondition für neue Method | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Pure Method** | | | must not update any preexisting state, but is allowed to modify objects that have been created after entry into the pure method. e.g. map.put(), map.remove() is not pure // map.contains(), stack.top() sind pure | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| @ThrowEnsures("ExeptionName", "ensures") | | | | | | | | | | | | | | | method() == old(method()) | | | | | | | | | | @Ensures({“contains(k) ? result != null : resul == null”}) | | | | | | | | | | | |
| **Error Handling** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Prävention | | | Architektur und Code Reviews, Unit und Integration Tests, Statische Code Analyse, Concurrency Testing, Logging | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Behandlung** | | | Lokale Behandlung wenn möglich oder Fehler nicht weiter relevant, ansonstne an Aufrufer delegieren | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Konservativ**: Error Handling Prozedur aufrufen, Fehlermeldung anzeigen, Shutdown | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Optimisitsch**: Neutrales Resultat, Bestmögliches plausibles Resultat, Warnung loggen | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Korrektheit** | | | Niemals ein ungenaues Resultat liefern (oft bei kritischen Systemen) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Robustheit** | | | Versuche Software am Laufen zu halten (oft bei unkritischen Systemen) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Defensive Programming | | | Systematische Fehlerprüfung von allen Inputs, sowie eine systematische Fehlerbehandlung. Defensive Programming macht nur in bestimmten Anwedungsbereichen Sinn, da es mit einem grösseren Aufwand verbunden ist.  **Nachteile**: u.U nicht testbar, kostet Laufzeit und Programmierzeit, duplizierter code | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Fehlerbarrikade | | | Hinter der Barrikaden sind die Daten gültig. Beim Grenzübertritt überprüfen. (Facade) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Exception Policy | | | Lokales oder globale Behandlung, Checked oder Unchecked Exceptions? | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Exceptions** | | | Für möglichst produktive Quellen, für sicherheitsrelvante Fehler, | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Assertions** | | | Können ab-, eingeschalten werden, für States die nie eintreten sollen, Pre und Post Conditions *Beispiel***:** if (a && b) { ... } else if (!a) { ... } else { assert !b; ... } 🡪 Kein ausführbarer Code in Assertions! | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Logging** | | | Für Diagnose, um Fehler zu identifizieren, System Irregularitäten erkennen (Log4J benutzen) 🡪 Postmortum Analyse | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Testing** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Eine hohe Testabdeckung ist eine notwendige aber nicht hinreichende Bedingung. Unit Test beweise nicht die Absenz von Fehler! | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Dependency Injection**: 1.Interface erstellen,2.Objekte über Konstruktor injection, 3.Fake und Entity implementieren Interface. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Fake** | Vereinfachte schnelle Implementierung (z.B In-Memory DB) | | | | | | | | | | | | | | | | | | | | | **Stub** | | | | | Auf Testfall zugeschnittene Antworten | | | | | | | | | |
| **Mock** | Auf Testfall zugeschnitten; prüft Call Reihenfolge und Inhalt | | | | | | | | | | | | | | | | | | | | | **Dummy** | | | | | Objekte nur herumgereicht, aber nie inspiziert | | | | | | | | | |
| **TDD** | 1. Specify it (Essens-, Test-, Assert First), 2. Frame it, 3. Evolve it (Simplest first, Refactor mercilessly) 🡪 Always baby steps | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1. RED: Test fails, Code compiles, No implementation 2. GREEN: Make it passing (Do the simplest thing that could possibly work) 3. Refactor until consistent and good design 4. Integrate in codebase (optional) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Typen** | Unit Test, Integration Test, System Tests (Cucumber, Selenium), Performance Test, Usability Test, Blackbox Test, Whitebox Test | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Coverage** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Statement** | | Meistverwendete Variante, zählt Zeilen, welche durch Tests abgedeckt sind. (Ausgeführte Anw. / Gesamtzahl Anw.) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Branch** | | Jeder mögliche Zweig (if/else, switch,..) wird gezählt. (Abgedeckte Zweige / Gesamtzahl Zweige) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Decision** | | Das Verhältnis von ausgewerteten atomaren Werten (Term, Bedingung, ...) innerhalb von Ausdrücken zu allen vorhandenen atomaren Werten in einem Modul. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Path** | | Das Verhältnis der getesteten Pfade (Wege im Kontrollflussgraph), zu allen möglichen Pfaden in einem Modul. (unreal.) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Integration Testing** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| +Endecken mehr Fehler, +sind langlebiger, +können E2E Test ersetzen, - können hohe Laufzeiten haben | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Unit Testing** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| - Irreführende Testabdeckung, -oft nichtssagend (2+2=4), -überleben Refactorings nichts, - viel Mocking in oberen Layer | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Performance Testing** 1. Get it to run 2. Get it right 3. Make it fast (nicht zu früh optimieren) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Profiling (Whitebox) | | | | | | | | | | | Antwort auf: “Wo wird die Zeit verbraten?» (Tool: TimeTracker, Für Java: NetBeans). Jedem Methoden Aufruf wird ein Timer Feld angehängt und neu kompiliert. | | | | | | | | | | | | | | | | | | | | | | | | | |
| Last-Messungen (Blackbox) | | | | | | | | | | | Antwort auf: «Wann geht der Server in die Knie?» (Tools: gatling.io, JMeter) | | | | | | | | | | | | | | | | | | | | | | | | | |
| Wiederholte Performance Messung | | | | | | | | | | | Antwort auf: «Sind die typischen Antwortszeiten immer noch etwa gleich?» (Tool: Selenium) | | | | | | | | | | | | | | | | | | | | | | | | | |
| Application Telemetry | | | | | | | | | | | Viele Sensoren / Log Outputs programmieren 🡪 zur Lokalisierung von Fehlern | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Usability Testing** etwa zur Mitte des Projektes mit 3 geeigneten Testpersonen (normale User) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Software, unvoreingenommene User in Usability Lab ("helfen Sie uns", "denken Sie laut mit", "Sie können nichts falsch machen", „Die Software wird getestet, nicht Sie“), Moderator der Szenario-Aufträge gibt und beobachtet (und fragt, aber nicht hilft). | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Varianten** mit Paper Prototyping, Clickable Wireframes und Mockups, A/B Testing, UX-Benutzerführung/(Navigation Map) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Code Smells**  (If it stinks, change it) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Conditonal Complexity** | | | | | | | | Sehr viele unübersichtliche Fallunterscheidungen | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Duplicated Code** | | | | | | | | Entsteht oft durch Copy-Paste und Abändern von bestehendem Code | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Large Class** | | | | | | | | Klasse hat viele Responsibilities.Tipp: viele Instanzvariablen | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Long Method** | | | | | | | | Wenn eine Methode zu lang ist, um sie zu verdauen. ( > 10 Zeilen) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Switch/Case** | | | | | | | | Switch Statements besser mit Polymorphismus ersetzen (oder Hashmap) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| interface Command { public void execute(); } | | | | | | | | class RequestHandler {  public void handleRequest(int action) {  switch(action) {  case LOGIN: doLogin(); break;  case LOGOUT: doLogout(); break;  case QUERY: doQuery(); break; } } } | | | | | | | | | | | | | | | | | | class LoginCmd implements Command {  public void execute() { doLogin() } }  class RequestHandler {   private Map<Integer, Command> cM;  public void handleRequest(int action) {  cM.get(action).execute();} } | | | | | | | | | | |
| **Comments** | | | | | | | | Zu viele unnötige Kommentare | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Magic Numbers | | | | | | | | Konstante ausser "", 0, 1. Auch literale Strings sind eine Art von Magic Numbers | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Primitve Obsession | | | | | | | | Klassen zu wenig abstrahiert. Low level instructions verwendet (if-else,...) 🡪 ADMIN\_ROLE = 1 | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| final String phoneNum = "5551230004";  System.out.println(  "Our support line in North America is (" +  phoneNum.substring(0,3) + ") " +  phoneNum.substring(3,6) + "-" +  phoneNum.substring(6,10)); } | | | | | | | | | | | | | | | | | | public class Phone {  private final String unformattedNumber;  private String getAreaCode() {  return unformattedNumber.substring(0,3); }  public String toFormattedString() {  return "(" + getAreaCode() + ") " + getPrefix() + "-" } | | | | | | | | | | |
| Feature Envy | | | | | | | | Klasse manipuliert Daten anderer class. Oft Duplicated Code 🡪Move Method, Extract Method | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| class Phone { /\*fields and methods \*/ }  class Customer… { private Phone mP;  public String getNumber() {  return "(" + mP.getAreaCode() + ") " +  mP.getPrefix() + "-" + mP.getNumber();  }} | | | | | | | | | | | | | | | | | | class Phone { public String toFormattedString() {  return "(" + getAreaCode() + ") " +  getPrefix() + "-" + getNumber(); } }  class Customer… { private Phone phone;  public String getNumber() {  return phone.toFormattedString(); } } | | | | | | | | | | |
| Inappropriate Intimacy | | | | | | | | Eine Klasse verwendet interne Variablen und Methoden einer anderen Klasse | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| public class Student {  public Set<Workshop> getWorkshops() { … }  public void setWorkshops(Set<Workshop> ws) { … }  } | | | | | | | | | | | | | | | | | | Set<Workshop> workshops = student.getWorkshops();  workshops.add(Workshop.DPW300);  student.setWorkshops(workshops); | | | | | | | | | | |
| Lazy Class | | | | | | | | Klasse tut zu wenig um ihre Existenz zu rechtfertigen | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Middle Man | | | | | | | | Klasse delegiert fast alles | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Long Parameterlist | | | | | | | | Nur so viel an Methode übergeben, dass sie sich alles beschaffen kann, was sie braucht. | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Oddball Solution | | | | | | | | Gleiches Problem auf zwei Varianten gelöst | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Refused Bequest | | | | | | | | Client verwendet direkt Subklasse (Verletzt *Liskov Substitutionsprinzip*) 🡪Methoden nicht richtig implem.  Beispiele: inherited Method throws *UnsupportedOperationException*, inherited Method has empty Body | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Shotgun Surgery  Solution Sprawl | | | | | | | | Änderungen verursacht Anpassungen in vielen Klassen | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Wenn Code für eine Aktion auf diverse Klassen verteilt ist 🡪 Oft resultiert Duplicate Code | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| SpeculativeGenerelatity | | | | | | | | Wenn für eine mögliche Verwendung in der Zukunft programmiert wird.(e.g Interface für nur 1 Klasse) | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Refactoring** (Improving the design of existing code without changing the logic. Always **baby steps**) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Vorgehen**: **1**.verify that all test green **2**.Decide what code to change **3**. implement refactoring (baby steps) **4**. Run tests (no altered behaviour!) **5**. Repeat until the refactoring is complete or revert to an earlier state | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Architektur Refactoring:** If you think good architecture is expensive, try bad architecture  **1**.Layeringeinführung **2**.Zyklische Abhängikeiten entfernen **3**.Business Layer entwirren, **4**.Querverbindungen entfernen. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Refactorings** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Rename Meth, Field, Param** | | | | | | | | | Umbenennen, dass der Zweck besser wiederspiegelt wird | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Change Method Signature** | | | | | | | | | Change name, Return type, visiblity scope | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Change Value to Reference | | | | | | | | | If you have many identical instances of a single class that you need to replace with a single object | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Encapsulate Field | | | | | | | | | Public field 🡪 Make private & provide accessors (Getter/Setter) | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Collapse Hierarchy | | | | | | | | | Vereinige Basisklasse und Unterklassen | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Decompose Conditional | | | | | | | | | Extract Method für Bedingung, then-Block und else-Block. | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Introduce Null Object | | | | | | | | | Ersetze Null mit Null Object | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Substistute Algorithm | | | | | | | | | Ersetze Algorithmus mit einfacher verständlichem | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Extract Interface | | | | | | | | | Subset in eigenes Interface | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Extract Method/Inline Method** | | | | | | | | | Ein Codefragment in Methode zusammenfassen / Methode wieder inline einfügen | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Extract Subclass/Superclass | | | | | | | | | Erzeuge zweite Klasse und verschiebe relevante Attr./Meth. | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Extract Variable/Inline Temp | | | | | | | | | Speichere Ausdruck in temp Var, Name erklärt Zweck. / Ersetzte Referenzen auf die Var mit Ausdruck. | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Move Method, Field** | | | | | | | | | Verschiebe die Methode in andere Klasse. Erzeuge ein neues Feld und passe Clients an. | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Push down Field, Meth, Constr. | | | | | | | | | Objekt in Kinder verschieben | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Pull up Field, Method, Constr. | | | | | | | | | move Obj to Parent | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Replace Constructor with *Factory Method* | | | | | | | | | Ersetze Konstruktor mit Factory Methode | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Replace Exception with  *Test* | | | | | | | | | Eine Methode wirft eine CheckedException auf Grund einer Bedingung, die der Aufrufer überprüfen könnte → Der Aufrufer soll die Bedingung prüfen und die Exception verhindern. | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Replace Inheritance with *Delegation*** | | | | | | | | | Erzeuge in der Subklasse eine Referenz zur Basisklasse und ändere Methoden so, dass sie über die Referenz an Basisklasse delegieren. Löse die Vererbung auf. | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Replace Magic Number with *Constant*** | | | | | | | | | Ersetze Literal mit einer sprechender Konstante | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Replace Nested Cond. with *Guard Class* | | | | | | | | | Verwende für Spezialfälle Guards und entsprechende Meth. | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Replace Temp with  *Query* | | | | | | | | | Extrahiere Temporäre Variable in Methode, Ersetze Ref auf Variable mit Methodenaufruf. | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| double basePrice = quantity \* itemPrice;  if (basePrice > 1000) {  return basePrice \* 0.95;  } else { return basePrice \* 0.98; } | | | | | | | | | | | | | | | | | | | | if (basePrice() >1000) {  return basePrice()\*0.95;  } else { return basePrice() \* 0.98; }  double basePrice() { return quantity\*itemPrice; } | | | | | | | |
| Replace Type Code with *Class/Subclass* | | | | | | | | | Type code occurs when, instead of a separate data type, you have a set of nubmers or strings that form a list of allowable values from some entity. Often these specific numbers and strings are given understandaable names via constants, which is the reason for why such type code is encountered so much 🡪 Create Class for each coded type | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Replace Type Code with  *State or Strategy* | | | | | | | | |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | **Code Smell – Refactoring – Matrix** | Comment | Conditional Complexity | Duplicated Code | Feature Envy | Inapporpriate Intimacy | Large Class | Lazy Class | Long Method | Long ParameterList | Magic Numbers | Middle Man | Oddball Solution | Primitive Obsession | Refused Bequest | Shotgun Surgery | Solution Sprawl | Speculative Generality | Switch/Case | | Change Method Signature | x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | Change Value to Reference |  |  |  |  |  | x |  |  |  |  |  |  | x |  |  |  |  |  | | Collapse Hierarchy |  |  |  |  |  |  | x |  |  |  |  |  |  |  |  |  | x |  | | Decompose Conditional |  |  |  |  |  |  |  | x |  |  |  |  |  |  |  |  |  |  | | Extract Method | x |  | x | x |  |  |  | x | x |  |  |  |  |  |  |  |  |  | | Extract Class |  |  | x |  | x |  |  |  |  |  |  |  | x |  |  |  |  |  | | Extract Subclass |  |  |  |  |  | x |  |  |  |  |  |  |  |  |  |  |  |  | | Inline Method |  |  |  |  |  |  |  |  |  |  | x |  |  |  |  |  |  |  | | Introduce Null Object |  | x | x |  |  |  |  |  |  |  |  |  |  |  |  |  |  | x | | Move Field |  |  |  | x | x |  |  |  |  |  |  |  |  |  | x |  |  |  | | Move Method | x |  |  | x | x |  |  |  |  |  |  |  |  |  | x |  |  |  | | Pull up Field |  |  | x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | Pull up Method |  |  | x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | Push Down Field |  |  |  |  |  |  |  |  |  |  |  |  |  | x |  |  |  |  | | Push Down Method |  |  |  |  |  |  |  |  |  |  |  |  |  | x |  |  |  |  | | Rename Method | x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | x |  | | Rpl. Inheritance w. Delegation |  |  |  |  | x |  |  |  |  |  |  |  |  | x |  |  |  |  | | Rpl. Magic Number w. Const. |  |  |  |  |  |  |  |  |  | x |  |  |  |  |  |  |  |  | | Rpl. Temp w. Query |  |  |  |  |  |  |  | x |  |  |  |  |  |  |  |  |  |  | | Rpl. Type Code w. Class/Subclass |  |  |  |  |  |  |  |  |  |  |  |  | x |  |  |  |  | x | | Rpl. Type Code w. State/-Strategy |  |  |  |  |  |  |  |  |  |  |  |  | x |  |  |  |  | x | | Substitute Algorithm |  |  | x |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | Unify Interfaces with Adapter |  |  | x |  |  |  |  |  |  |  |  | x |  |  |  |  |  |  | | *Factory einführen* |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | x |  |  | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Refactoring to Pattern** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Conditional Complexity** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Replace Condtional Logic with Strategy** (verständlicher, kann zur Laufzeit ausgetauscht werden, Overkill) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Conditional logic in a method controls which of several variants of a calculation are executed. Create a Strategy for each variant and make the method delegate the calculation to a Strategy instance. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| public class Loan {  public double capital() {  if (expiry == null && maturity != null) {  return commitment \* duration() \* riskFactor(); }  if (expiry != null && maturity == null) {  if (getUnusedPercentage() != 1.0) { return commitment \*  getUnusedPercentage()\*duration()\*riskFactor();  } else { return …. | | | | | | | | | | | | | | | | | | | | | public class Loan {  private CapitalStrategy capitalStrategy;  public double capital() { return capitalStrategy.capital(this); }  public abstract class CapitalStrategy {  public abstract double capital(Loan loan); }  public class CapitalStrategyTermLoan extends CapitalStrategy {  public double capital(Loan l) {  return l.getCommitment() \* duration(l) \* riskFactorFor(l) }  } // weitere CapitalStrategy-Subklassen | | | | | | | | | | | | | | | |
| **Replace State-Altering Conditionals with State** (Zustandsübergänge eines Objekts kontrollieren) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| The conditional expressions that control an object's state transitions are complex. Replace the conditionals with State classes that handle specific states and transitions between them. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | | |  | | | | | | | | | | | | | | | |
| **Move Embellishment to Decorator** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Place each embellishment in its own class and let that class wrap the type of object it needs to embellish so that clients may wrap the embellishment around objects at runtime, when special-case behavior is needed | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | | |  | | | | | | | | | | | | | | | |
| **Introduce Null Object** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Some methods return `null` instead of real objects, you have many checks for `null` in your code. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| if (customer == null) {  plan = BillingPlan.basic();  } else {  plan = customer.getPlan();  } | | | | | | | | | | | | | | | | | | class NullCustomer extends Customer { boolean isNull() { return true; }  Plan getPlan() { return new NullPlan(); }  } customer = (order.customer != null) ? order.customer : new NullCustomer();  plan = customer.getPlan(); | | | | | | | | | | | | | | | | | | |
| **Indecent Exposure** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Encapsulate Classes with Factory** (oder Dependency Injection) 🡪 gray: classes not visible outside of packages | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Clients directly instantiate classes that reside in one package and implement a common interface. Make the class constructors non-public and let clients create instances of them using a Factory. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | | |  | | | | | | | | | | | | | | | |
| **Oddball Solution** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Unify Interfaces with Adapter** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Clients interact with two classes, one of which has a preferred interface. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | | |  | | | | | | | | | | | | | | | |
| **Substitute Algorithm** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Replace existing algorithm with a newer, cleaner one | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Solution Sprawl** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Move Creation Knowledge to Factory** (Enkopple Client vor komplexer Erzeugungslogik, Aufwändig) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| Code used to instantiate a class is sprawled across numerous classes.Move the creation knowledge into a single Factory class. | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Patterns** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Strategy**  The Strategy Pattern defines a family of algorithms, encapsulates each one and makes them interchangable. Stategy lets the algorithm vary independently from clients that use it. | | | | | | | | | | | | | | | | | | | | **State**  The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class. | | | | | | | | | | | | | | | | |
| **Adapter** | | | | | | | | | | | | | | | | | | | | **Factory** | | | | | | | | | | | | | | | | |
| The Adapter Pattern converts the interface of a class into another interface the client expects. Adapter lets classes work together that couldn’t otherwise because of incompatible interfaces. | | | | | | | | | | | | | | | | | | | | The Factory Method Pattern de nes an interface for creating an object, but lets subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. | | | | | | | | | | | | | | | | |
|  | | | | | | | | | | | | | | | | | | | |
| **Proving** | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Loop Measure** | | | | | Typischerweise eine natürliche Zahl, die Beweist, dass eine Loop terminiert (x-i) | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| **Hoare Triples** | | | | | **Beispiel:**  🡪 Precondition  Programm  🡪 Postcondition | | | | | | | | | | | | | | | | | | | | | | | abs funct | | | | | | | mult y mit x | swap ohne Speicher |
| **Proving Correctness**  (Pre- und Post-Condition gegeben) | | | | | Aufgabenstellung: | | | | | | | | | | | Lösung: | | | | | | | Vorgehen:  1. Zuunterst beginnen  2. x in der untersten Reihe mit (x + 2) ersetzen und in der freien Zeile einsetzen (Schritt wiederholen bis zuoberst)  3. x in der drittuntersten Reihe mit (x + 1) ersetzen und in der freien Zeile darüber einsetzen  4. Ist man zuoberst angekommen, muss man die Korrektheit des Ausdrucks mathematisch beweisen:  🡪 Verification Condition | | | | | | | | | | | | | |
| **If:**    **{ -x >= 0 } }**  **{ x >= 0 }** | | | | | | | | | | | | | **While: (Inv ist gegeben)** | | | | | | | | | | 1. Schritt: Nach Schema ausfüllen  Loop invariant ∧ ¬ while condition  Loop invariant ∧ while condition  Loop invariant  Loop invariant  2. Schritt: Verification Conditions ermitteln (durch einsetzen) | | | | | | | | | | | | | |