

SE2


Zusammenfassung

FS2014 - Geschrieben von Theo Winter

Begriff	Code Snippet / Erklärung
Achieve Orthogonality	Keine Kopplung zwischen Sachen die nichts miteinander zu tun haben.
Acyclic Dependency Prinzip	<p>Der Dependency Graph von Paketen oder Komponenten soll keine Kreisläufe (Cycle) haben. Dagegen anwenden kann man:</p> <ul style="list-style-type: none"> • Dependency Inversion Principle (Highlevel module sollen nicht von lowlevel Modulen abhängig sein. Beide sollen von Abstraktionen abhängig sein. Abstraktionen sollen nicht von Details abhängig sein. Details sollen von Abstraktionen abhängig sein. • (Einfacher:) Ein neues Packet erstellen und die gemeinsame Abhängigkeit in dieses Packet verschieben <p>Siehe —> Circular Dependency für Antwort: wieso schlecht?</p>
Agil - Wann?	<p>Agil Entwickeln: Wenn die Entwicklung einer Softwarelösung fachlich und technisch komplex scheint und man schrittweise vorgehen muss.</p> <p>Traditionelles Verfahren: Bei kurzzeitigen Projekten oder solchen bei denen man die Zeit sehr gut einschätzen kann. Oder wenn es mehrere Projekte gibt die sich wiederholen oder sehr stark ähneln.</p> <p>Wenn ein Projekt klar (vorher) definierte Anforderungen erfüllen muss und Enge Zeit- oder Budgetvorgaben hat. Hier lohnt sich ein traditionelles, restriktiveres Projektmanagement-Modell mit klar definierten Phasen mehr. (gem. Wikipedia)</p> <p><i>(auch Gegenteil von Folien 28 bei Agil) —> Wenn Team weit auseinander ist sich nicht treffen kann, unterschiedliche Sprachen spricht, sehr spezifizierte Mitarbeiter hat.</i></p>
Agile Projektführung	Agile Softwareentwicklung versucht mit geringem bürokratischen Aufwand, wenigen Regeln und meist einem iterativen Vorgehen auszukommen.
Anforderungen	Sollen testbar sein, d.h. mit Zeit-Angabe und nicht Wörtern wie "instant". Auch muss man darauf achten was der Input ist. E.g. grosses Bild dauert länger zum verarbeiten als kleines Bild.
Anforderungen - Erarbeiten	<ul style="list-style-type: none"> • Zusammenarbeit mit dem Benutzer: Denken aus Benutzersicht • Kritisches Hinterfragen & Nachbearbeiten: Echte Anforderungen von ad-hoc Wünschen trennen, jeweils nach dem Grund fragen. • Genügend generell und abstrakt definieren: Details können schneller ändern, Details konfigurierbar halten • Ursprung tracken: Person, Grund, Datum <p>Requirements Review: (Folien Code Reviews S25)</p> <ul style="list-style-type: none"> • Wissenstransfer vom Kunden zum Entwickler • Ping-Pong mit Kunden um möglichst formale Requirements Specs zu erarbeiten • Kostenschätzung • QA-Testing
Anforderungen - Erfassungs-Qualität	<p>Korrekt: Keine Fehler</p> <p>Komplett: Funktionale und Nicht-Funktionale Anforderungen</p> <p>Konsistent: Ohne Widersprüche</p> <p>Concise: So knapp und formal wie möglich</p> <p>Traceable: Requirements — Design — Code — Test — Dokumentation</p> <p>Testbar: Messbare Erfolgskriterien</p> <p>Adequate: Zielpublikum, richtige Detail-Tiefe, Vokabular</p>
Anforderungen - Qualität	Qualität, d.h. Skalierbarkeit, Performance, Security, Robustness sollen auch Anforderungen sein. Dabei ist wichtig dass diese testbar sind. z.B. max Antwortzeit mit Zeit-Angabe. Diese Qualitäts-Anforderungen sollen auf echten Anforderungen basieren z.B: User Erwartung 10 Sekunden.
Arbeitspaket - Fallstricke	<ul style="list-style-type: none"> • Nicht nur generische Arbeitspakete aufführen (e.g. Use Cases schreiben, Domainmodell machen) • Auch Arbeitspakete für unproduktive Tätigkeiten erstellen (z.B. Besprechungen, Einrichten des Servers, Schreiben des Testplans) • Arbeitspakete abhakbar formulieren: tabellarisch und klein genug und gute Akzeptanzkriterien • Arbeitspakete schreiben kostet auch Zeit!
Arbeitspaket - Grösse	<ul style="list-style-type: none"> • maximal halb so gross wie dass was eine Person innert einer Iteration schafft • Paket sollte innerhalb einer Iteration fertig sein • Arbeitspakete dürfen nur "nicht angefangen", "angefangen" oder "fertig" sein. Etwas dazwischen gibt es nicht. <p>Wichtig: Entwickler schätzen den Umfang! Kunde können NUR priorisieren! (Einzigte Ausnahme: Architektur relevante Pakete können vom Syl-Architekt in bestimmte Iterationen gesetzt werden, muss aber Kunden erklärt werden, der muss absegnen)</p> <p>Falls noch nicht oder nicht genau schätzbar —> mindestens T-Shirt sizing verwenden S, M, L, XL etc.</p>
Arbeitspaket - Inhalt	<ul style="list-style-type: none"> • ID, Titel • Kurze Beschreibung, kann in der Form "As AAA I want to BBB because CCC" sein (typische User Story in Scrum) • Akzeptanz-Kriterien • Schätzung Aufwand • Priorität für Kunde • Geleistete Stunden (Zeitaufschreibung) • Status • Arbeits-Kategorie

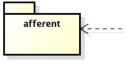

Begriff	Code Snippet / Erklärung
Arbeitspakete - Organisieren	<ul style="list-style-type: none"> müssen zentral an einem Ort gespeichert werden von allen eingesehen und editiert werden priorisierbar sein Schätzung und Ist-Zeit enthalten —> Tool wie Redmine, MS TFS oder JIRA einsetzen. NICHT EXCEL O_o
Architektur-Relevanz	<ol style="list-style-type: none"> Wahl der Implementationssprache: Ja, späterer Wechsel ist nicht möglich (ohne alles neu zu Programmieren) Wahl einer bestimmten Bibliothek: Ja, ein späterer Wechsel ist mit sehr viel Aufwand verbunden.
Architektur Review	Gründe die für ein Architektur Review sprechen: <ul style="list-style-type: none"> Wenn man jetzt nur 1 Fehler findet spart man viel Zeit Jetzt sind Fehler noch günstig, später viel viel teurer zum korrigieren Die Architektur ist wie das Fundament/Gerüst des Projekts.
Automatisierung	Siehe —> Projekt Automatisierung
Awkward (Wort)	heikel, unangenehm, schwierig, unbehaglich
Awkward Testing - Gründe	<ul style="list-style-type: none"> Code der von Zeit abhängig ist. Code der von zufälligen Zahlen abhängig ist. Code der von kaputten Collaborators abhängig ist. Awkward Inclusion Pyramid - Ein Modul ist abhängig von vielen andern, d.h. in Wirklichkeit werden Integrations-Tests durchgeführt und gar nicht micro-tests. Awkward Access - z.B. Drucker/Netzwerk/FileSystem können nicht vollständig kontrolliert werden.
Best Practices für Entwickler	<ul style="list-style-type: none"> <input checked="" type="checkbox"/> Commit code frequently (small changes, commit after each task) <input checked="" type="checkbox"/> Don't commit broken code <input checked="" type="checkbox"/> Fix broken builds immediately <input checked="" type="checkbox"/> Write automated developer tests <input checked="" type="checkbox"/> All tests and inspections must pass <input checked="" type="checkbox"/> Run private builds (Emulate integration build locally) <input checked="" type="checkbox"/> Avoid getting broken code
Beziehung 1:1	Kann man weglassen
Beziehung n:m	Braucht Zwischentabelle
Brooks'sches Gesetz / Brook's law	<p>Irgend so ein Typ (Brook) hat gesagt: "Adding manpower to a late software project makes it later".</p> <p>Wann ist das wahr: (Vorlesung 10)</p> <ul style="list-style-type: none"> Neue Mitarbeiter müssen sich zuerst in den Code einarbeiten und wissen nicht über getroffenen Entscheidungen etc. bescheid. D.h. die alten Mitarbeiter müssen den Neuen zuerst helfen bevor diese überhaupt nützlich sein können (ramp up time). Können evtl. auch Bugs einführen. Der Kommunikationsoverhead wird grösser auf Grund der Kombinatorischen Explosion (jeder spricht mit jedem). D.h. der Synchronisationsaufwand wird massiv grösser, und es müssen evtl. neue Kommunikationskanäle eingerichtet werden. Dazu kommt je mehr Leute an der gleichen Aufgabe sind desto mehr Zeit müssen sie verbringen um herauszufinden was die anderen Leute (an der Aufgabe) am machen sind. <p>Wann ist das nicht wahr / Ausnahmen:</p> <ul style="list-style-type: none"> Wenn man Spezialisten zum Projekt hinzufügt die wenig bis gar kein Training benötigen sondern dem Projekt sofort helfen können wieder auf die Richtige Spur zu kommen. (Teuer) Wenn gute Management Praktiken und Design Patterns eingesetzt wurden (bevor man neue Leute hinzufügt) dadurch wird der Effekt von Brook's law minimiert. Wenn man Leute hinzufügt bevor es zu spät ist.
Circular Dependency	<p>Das Problem bei Zirkulären Dependencies ist dass sie ungewollte Effekte in SW bringen können. z.B. tight Coupling, welches es unmöglich macht ein Modul von anderen zu separieren/ändern/ersetzen ohne gleich die anderen Module auch anzupassen. —> Domino Effekt: eine kleine Änderung in einem Modul hat ungewollte Folgen in anderen Modulen. Auch können CDs in unendlicher Rekursion oder anderen memory leaks enden.</p> <p>Siehe auch —> Acyclic Dependency Prinzip</p>
Code Qualität	Siehe —> Metriken.
Code Smell finden	<p>Code Smells die einfach zu finden sind:</p> <ol style="list-style-type: none"> Comments: Wenn //Grow array if necessary über einem offensichtlichen Code Teil steht Magic Numbers: Wenn eine bestimmte Nummer immer wieder verwendet wird. —> Sollte eher als konstante/final definiert werden Duplicate Code: Wenn ein gewisser Code-Teil immer wieder vorkommt. —> eigene Methode machen <p>TODO add more einfach zu identifizierende Code smells die an Prüfung kommen könnten.</p>
Continuous (1) Compilation	Man kann auf Änderung reagieren, ein Checkout des Codes machen. Den Code kompilieren und Testen und man wird benachrichtigt wenn etwas falsch läuft.
Continuous (2) Integration	Bei der Continuous Integration wird auch die Test-Coverage des Codes gemessen. Es wird eine durchschnittliche Code-Komplexität festgestellt. Man kann duplizierten Code erkennen. Builds werden klar nummeriert und die deployte Software wird archiviert/gelagert.

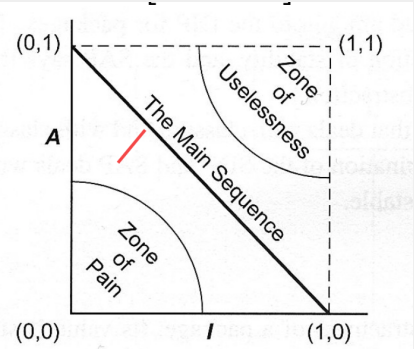
Begriff	Code Snippet / Erklärung
Continuous (3) Delivery	Bei Continuous Delivery ist die Software die ganze Zeit deployable. D.h. mit nur "einem" Tastendruck kann eine brauchbare Version für eine beliebige Umgebung erstellt werden.
Continuous (4) Deployment	Bei Continuous Deployment geht man noch einen Schritt weiter. Änderungen werden direkt in die Produktion gebracht. Der ganze Delivery Prozess ist automatisiert. z.B. Google Chrome Dev Version mit Auto-Updates (ist auch nicht 100% CD aber sehr nahe dran).
Continuous Delivery Process Overview	<pre> sequenceDiagram participant DT as Delivery team participant VC as Version control participant B as Build & unit tests participant A as Automated acceptance tests participant U as User acceptance tests participant R as Release DT->>VC: Check in VC->>B: Trigger B-->>VC: Feedback VC->>VC: Check in VC->>B: Trigger B-->>VC: Feedback VC->>A: Trigger A-->>VC: Feedback VC->>VC: Check in VC->>B: Trigger B-->>VC: Feedback VC->>A: Trigger A-->>VC: Feedback A->>U: Approval U->>R: Approval </pre>
CRISP	Complete: Der Build-Prozess beinhaltet alles was nötig ist um zu builden. D.h. build von scratch Repeatable: Konsistente Resultate basierend auf der Version im Source Baum. Informative: Wertvolle Informationen zu Erfolg oder Misserfolg des Builds angeben. Schedulable: Man kann einen Build für eine spez. Zeit planen oder durch einen Event triggern. Portable: Builden funktioniert auf jeder Machine, nicht nur auf eine speziellen.
Cyclomatic Complexity	Wird auch McCabe Metrik genannt. Wie Rechnen → Siehe McCabe
Datenbank löschen	Füllen: Unten nach Oben (Kategorie 1 zuerst) Löschen: Oben nach Unten (Log einträge N oder * zuerst, danach Kategorie)
DBC - Beispiel 2 iContract	<pre> public interface PrimitiveIntStack{ // @inv isFull() implies !isEmpty() // @inv isEmpty() implies !isFull() // @pre isFull() == false // @post top() == value // @post isEmpty() == false public void push(int value); // @pre isEmpty() == false // @pre isFull() == false // @post @return == top() @pre public int pop(); // @pre isEmpty() == false public int top(); public boolean isEmpty(); public boolean isFull(); } </pre>
DBC - Beispiel EiffelCode	<pre> require //precondition count > capacity not key.empty do .. ensure //postcondition has(x) item(key) = x count = old count + 1 end </pre>

Begriff	Code Snippet / Erklärung												
DBC - Beispiel iContract	<pre>@inv !isEmpty() implies top() != null public interface Stack { @pre p != null @post !isEmpty() @post top() == o void push(Object o); @pre !isEmpty() @post @return == top()@pre Object pop(); @pre !isEmpty() Object top(); boolean isEmpty(); }</pre>												
DBC - Beispiel Theoretisch	<div><table><tr><th></th><th>Pflichten</th><th>Rechte</th></tr><tr><th><i>put</i></th><th>OBLIGATIONS</th><th>BENEFITS</th></tr><tr><td><i>Client</i></td><td>(Satisfy precondition:) Only call <i>put(x)</i> on a non-full stack.</td><td>(From postcondition:) Get stack updated: not empty, <i>x</i> on top (<i>item</i> yields <i>x</i>, <i>count</i> increased by 1).</td></tr><tr><td><i>Supplier</i></td><td>(Satisfy postcondition:) Update stack representation to have <i>x</i> on top (<i>item</i> yields <i>x</i>), <i>count</i> increased by 1, not empty.</td><td>(From precondition:) Simpler processing thanks to the assumption that stack is not full.</td></tr></table><p>aus [Meyer97]</p></div> <div><p>A routine contract: routine <i>put</i> for a stack class</p></div>		Pflichten	Rechte	<i>put</i>	OBLIGATIONS	BENEFITS	<i>Client</i>	(Satisfy precondition:) Only call <i>put(x)</i> on a non-full stack.	(From postcondition:) Get stack updated: not empty, <i>x</i> on top (<i>item</i> yields <i>x</i> , <i>count</i> increased by 1).	<i>Supplier</i>	(Satisfy postcondition:) Update stack representation to have <i>x</i> on top (<i>item</i> yields <i>x</i>), <i>count</i> increased by 1, not empty.	(From precondition:) Simpler processing thanks to the assumption that stack is not full.
	Pflichten	Rechte											
<i>put</i>	OBLIGATIONS	BENEFITS											
<i>Client</i>	(Satisfy precondition:) Only call <i>put(x)</i> on a non-full stack.	(From postcondition:) Get stack updated: not empty, <i>x</i> on top (<i>item</i> yields <i>x</i> , <i>count</i> increased by 1).											
<i>Supplier</i>	(Satisfy postcondition:) Update stack representation to have <i>x</i> on top (<i>item</i> yields <i>x</i>), <i>count</i> increased by 1, not empty.	(From precondition:) Simpler processing thanks to the assumption that stack is not full.											
DBC - Klasseninvarianten	<p>Garantien für Aufrufer, die immer d.h. vor und nach jedem Methodenaufruf gelten. Verantwortlich ist die Implementation der Klasse.</p> <p>Wenn eine Postcondition immer gelten soll, sollte man besser eine Invariante verwenden.</p>												
DBC - Postcondition	<p>Was gilt nach der Ausführung der Systemoperation?</p> <ul style="list-style-type: none">• Bedingungen die nach dem Aufruf gültig sind• Verantwortlich ist die Implementation der Methode <p>Wenn eine Methode eine Exception wirft unterscheidet man zwischen “Normalen Postconditions” und “Exceptional Postconditions”.</p> <p>Wenn eine Postcondition immer gelten soll, sollte man besser eine Invariante verwenden.</p>												
DBC - Precondition	<p>Was gilt vor der Ausführung der Systemoperation?</p> <ul style="list-style-type: none">• Bedingungen die vor dem Aufruf erfüllt sein müssen• Verantwortlich ist der Aufrufer <p>ACHTUNG: Dürfen nicht von Methode überprüft wären, sonst Verletzung von DRY. Eingangsparameter werden nur validiert wenn dies die explizite Aufgabe der Methode ist. Validieren heisst nicht Überprüfen von Preconditions. (Validieren soll ausserdem so nahe an den externen Quellen gemacht werden wie möglich.)</p>												
DBC - Typische Überprüfung	<ol style="list-style-type: none">1. Contains() - Post: Keine Veränderungen in Collection2. Put() Pre: Key key ist noch nicht in Collection3. Put() Post: Value & Key in Collection4. Remove() Post: Key nicht mehr in Collection												
DBC - Vererbung	<ul style="list-style-type: none">• Ein SubType muss mindestens Vertrag der Basisklasse erfüllen. D.h.• Subtype darf Precondition lockern aber nicht verschärfen.• Subtype darf Postcondition verschärfen aber nicht lockern.• Subtype darf Invariante verschärfen, aber nicht lockern.												
DBC - Vorteile / Nachteile	<p>Vorteile:</p> <ul style="list-style-type: none">• Es zwingt einem zu denken• Klare Verantwortlichkeiten• Viele Prüfungen, die bei defensiver Programmierung entstehen, erübrigen sich• Gut brauchbar als Dokumentation• Sauberes methodisches Hilfsmittel um Software Komponenten zu spezifizieren• Contract abstrahieren von der Implementation• Hilfsmittel fürs Debugging und Testen, Ergänzung zu Unit Tests (möglich wäre: automatische Analyse und automatische Generierung von Testdaten)• <i>C4J ist vielversprechend</i> <p>Nachteile:</p> <ul style="list-style-type: none">• Fehlende Sprachunterstützung• Hat sich in der Praxis nicht systematisch durchgesetzt												
Deploymentdiagramm	Man sieht wo (server) welche Komponenten deploy/verteilt werden.												

Begriff	Code Snippet / Erklärung
Design By Contract (DBC)	Man hat eine Kunde/Lieferanten-Beziehung zwischen Komponenten. Diese Beziehung ist durch Verträge geregelt.
Design for change	Man soll ein flexibles Design erstellen, so dass man z.B. später mal den DB-Provider wechseln kann ohne riesige Menge von Arbeit am eigentlichen Programm.
Divide & Conquer Strategie	Siehe —> Refactoring: Divide & Conquer
Domainmodelle vergleichen	Beide müssen gleich sortiert sein (n oben, 1 unten) Dadurch sind Unterschiede schnell ersichtlich.
DRY	Don't Repeat Yourself -
Efferent Coupling	Anzahl Packages ausserhalb von denen Klassen im Package abhängen. —> Bindungsmetriken
End of Elaboration - Checkliste	<ul style="list-style-type: none"> • Anforderungen / Requirements: Haben wir den Kunden verstanden? Ist der Funktionsumfang (Scope) durch UCs, Domain Model und nicht-funktionale Anforderungen abgedeckt? • User Interface Design: Haben wir Entwürfe gemacht, dem Kunde gezeigt; wenn möglich clickable Prototype plus Grafik-Entwürfe (Schriften, Farben)? • Software Architektur: Entwurf steht? Subsysteme und Interfaces definiert? Prototyp gemacht (durchstich durch alle Schichten)? • Entwicklungs-Werkzeuge und Methoden: Definiert und komplett aufgesetzt (IDE, version control, build server, unit testing, code analysis, inkl. dev-test-prod server, ticketing, bug tracking, UC writing proofing etc.) • Genauere Aufwandsschätzung: Liste aller Arbeitspakete?
Error Handling	<p>ist schlecht wenn “catch all ohne Behandlung”, weil:</p> <ul style="list-style-type: none"> • Programm ist sehr schwer zu debuggen da man die Fehler nicht findet • Das Programm funktioniert trotzdem nicht, der User kriegt das einfach nicht mit und meint es funktioniere • Das Programm macht etwas ganz falsches —> sehr schlimm bei Sicherheitskritischen Systemen <p>besser ist:</p> <ul style="list-style-type: none"> • wenn behandel-bar sofort behandeln (z.B. File I/O) • sonst weiter nach oben werfen • und nie ungültige Zwischenzustände —> finally block {} z.B. um Sockets zu schliessen etc. <p>bei Concurrency-Problemen:</p> <ul style="list-style-type: none"> • Stress-Tests (sehr oft ausführen) • Code-Review (Raise Conditions, Deadlocks, critical phase) • Load-Stress-Test • Kontrollierte Verzahnung
Exceptions: Global Exception Handler Example	<pre> public class ConverterApplication { public static void main(String[] args) { //Redirect default error stream, for example to file setErr(PrintStream err); //Set global Error Handler ExceptionHandler handler = new ExceptionHandler(); Thread.setDefaultUncaughtExceptionHandler(handler); SwingUtilities.invokeLater(new Runnable() { public void run() { // RUN MAIN STUFF } }); } } /* Example ExceptionHandler. All uncaught errors go here and get displayed in * a new error-frame.*/ static class ExceptionHandler implements Thread.UncaughtExceptionHandler { public void uncaughtException(Thread t, Throwable e) { System.err.println("Throwable: " + e.getMessage()); System.err.println(e.getStackTrace()); JOptionPane.showMessageDialog(new JFrame("Global Error Handler"), e.getMessage()); } } </pre>
Exceptions: Globaler Exception Handler	<p>Alle nicht lokal behandelten Exceptions kommen dort hin und Safety-Behandlungen werden durchgeführt. Protokollierung und Benutzer-Meldung.</p> <p>—> <code>Thread.setDefaultUncaughtExceptionHandler();</code></p>
Exceptions: Lokal vs Global	<p>Lokal nur für erwartete Fälle verwenden die für höhere Stellen nicht relevant sind und der Fall lokal abschliessbar ist, sonst an Anrufer delegieren. Wichtig: Keine ungültigen Zwischenzustände hinterlassen. D.h. finally-block.</p> <p>(Beispiel für lokal behandelte Fehler—> File I/O)</p>

Begriff	Code Snippet / Erklärung
Extract & Override Pattern (Fake)	<p>Den Problematischen Teil in eine separate Methode auslagern. Dann diese Methode in einer abgeleiteten Klasse überschreiben, so dass wir ihr genau den Body geben können der gebraucht wird (kann leer sein). Zum Schluss testen wir gegen diese abgeleitete Klasse.</p> <p>Wichtig: Man soll nur gerade soviel Code wegnehmen dass “awkward” wegfällt. Wenn man zuviel wegnimmt wird der Test unrealistisch!</p> <p>Was sind die Nachteile des E&O Pattern?</p> <ul style="list-style-type: none"> • Die Tests sind mit dem Code gekoppelt. • Die testbare Klasse kann nicht effektiv wiederverwendet werden. • Wenn man dies mehrmals verwendet werden Tests schwer zu lesen. <p>E&O Beispiel-Ablauf</p> <ol style="list-style-type: none"> 1. Pain-Quelle identifizieren 2. Pain-Quellen-Methode extrahieren 3. Klasse in Test umbenennen und generieren lassen 4. Extrahierte Methoden in Test-Klasse überschreiben mit z.B. statischen Rückgabewerten <p>Man kann damit 100% alle awkward MT Probleme in Java und den meisten Sprachen lösen. Dass heisst aber nicht das E&O es immer das beste Werkzeug ist</p>
Fake - Auto-Mock	Auto-Mocks sind gleich wie normale Mocks, werden jedoch “automagically” vom Computer generiert. Es gibt 2 Arten: Statische Mock Klassen, diese werden mit unserem Code erstellt und kompiliert. Dynamische Mock Klassen hingegen werden während der runtime erstellt werden. Dynamische Mocks sind heute sehr populär.
Fake - Listener	Ein Stub der unserer getesteten Klasse zuhört und sich erinnert was gesagt wurde.
Fake - Mock	Ein Fake der aktive kontrolliert dass die getestete Klasse sich korrekt verhält. Asserts verifizieren das Verhalten der Klasse und nicht den Status. D.h. Mock enthält selbst noch Asserts.
Fake - Simulator	<p>Ein Fake der schwer ist (viel Funktionalität beinhaltet) und so in komplexen Dialogen mit der getesteten Klasse engagiert werden kann. Siehe komplexer Flugsimulator mit Cockpit etc. Ein Simulator erscheint wie “the real thing”.</p> <p>Ein Simulator ist ein sehr ausführlicher Fake mit folgenden Eigenschaften:</p> <ul style="list-style-type: none"> • Lifecycle: Man kann einen Simulator instanzieren und sehr viele Tests damit durchführen • Extended Collaboration: Ein komplexes Protokoll das duzende von Simulator-Aufrufen involviert kann problemlos geschrieben werden • Once and for all: Einmal geschrieben und debugged simuliert ein Simulator komplett ein reales Objekt. D.h. man hat das Problem für alle Collaborators die den Simulator verwenden gelöst.
Fake - Stub	Einfachste Art des Fakes, sehr leicht-gewichtig. Bringt uns durch den Test aber tut sonst nichts.
Fake - Talker	Ein Stub der unserer getesteten Klasse vor-definierte Antworten gibt.
Fix Broken Windows	<p>Wenn man in einem Stadt-Quartier alle zerbrochenen Fenster repariert (& sonstige Renovationen) sinkt das Kriminalitätslevel automatisch und der Wohlstand nimmt zu.</p> <p>Dasselbe gilt beim Programmieren, wenn man Code nicht verwahrlosen lässt sondern immer schön refactored dann werden dass andere Programmierer die den Code verwenden/ändern auch tun. —> TLDR: Immer schön refactored ;-)</p>
Function Point Methode	<p>(Auch FPA - Function Point Verfahren). Dient zur Bewertung des fachlichen-funktionalen Umfang eines Informationstechnischen Systems.</p> <p>Vorteile:</p> <ul style="list-style-type: none"> • einfach • Erfahrungen können weiterverwendet werden • lässt sich als Basis für Aufwandschätzung, Benchmarking und ableiten von Kennzahlen verwenden
Funktionsumfang	engl. Scope, wird oft durch Domain Model, Use Case Diagram definiert
gerrit	Web-basiertes Tool für Code-Reviews. Quasi Firewall vor den Git-Branche. D.h. man pusht nicht mehr direkt zu den Branches, dies ist extrem wichtig für Open-Source Projekte mit vielen Leuten. z.B. Vote system ob ein commit gut ist oder nicht.
GIT - Arbeitsweise	<p>(Siehe auch —> Best Practices Entwickler)</p> <ul style="list-style-type: none"> • öfters mit Branches arbeiten (Branch = “Save As ...”) • ein Feature —> ein Branch, wenn Feature fertig —> Merge in Produktions Branch
GIT vs SVN	Git = dezentralisiert, SVN = zentralisiert. Bei GIT erfolgt das erstellen von Branches blitzschnell weil nicht die ganze Ordnerstruktur kopiert/gespeichert werden sondern nur Änderungen zum Hauptbranch.
Global Exception Handler	siehe Exceptions —> E
High Cohesion	Hohe Bindung. Code ist besser lesbar, Komplexität tiefer und der Code kann öfters re-used werden.

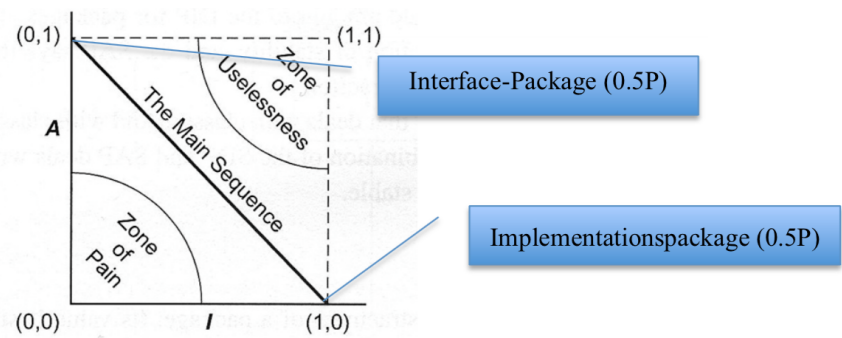
Begriff	Code Snippet / Erklärung
Ideal Engineering Days	<p>Programmierer die Aufwand schätzen, gehen davon aus dass sie unterbruchsfrei an Funktion arbeiten können. Das ist unrealistisch da —> Telefonanrufe, e-Mails, Geburtstagskuchen, Meetings, Unterbrechungen durch Tester, etwas dem Kollegen erklären.</p> <p>Man lässt aber trotzdem Programmierer in "Ideal Engineering Days" schätzen und korrigiert dann die Schätzung mit einem Faktor zwischen 1.5 und 3.</p>
Introduce Adapter	Manchmal darf oder kann man den Code des awkward collaborators nicht direkt berühren weil er eine third party library ist oder speziell alt. In solchen Fälle kann man einen "Introduce Adapter" verwenden. Dieser Adapter stecken wir zwischen unseren awkward Collaborator und den zu testenden Code.
Iteration - Planung	<ul style="list-style-type: none"> Genug Arbeitspakete, damit alle im Team beschäftigt sind. Genau soviel Arbeitspakete wie die Schätzungen zulassen, so dass sie auch innerhalb der Iteration fertig werden. Innerhalb des Teams werden die Arbeitspakete eigenverantwortlich zugeordnet und evtl. dynamisch verteilt.
KISS	Keep It Simple Stupid
Korrektheit vs. Robustheit	Korrektheit vs. Robustheit ist abhängig vom Einsatzbereich der Software. E.g. Korrektheit wichtig bei Röntgenmaschine, Robustheit bei Google-Suche.
LCOM	LCOM steht für Lack of Cohesion und ist eine Code Metrik. Klassen mit hoher LCOM sind einfacher in mehrere kleinere Klassen zu refactoren als solche die ein tiefer LCOM Wert haben.
McCabe Komplexität bei If-Statements	<p>Misst Komplexität der logischen Struktur eines Programms. Die Basis ist der Kontrollflussgraph G eines Programms. Die Zyklomatische Zahl ist die Anzahl an linear unabhängigen Pfaden durch ein Programm).</p> <pre> Switch +1 For +1 If +1 Case +1 Am Schluss noch +1 </pre> <p>Else: Ein reines Else zählt nicht als Verzweigung. D.h. wenn ein Else hinzugefügt wird ändert sich die Komplexitätszahl nicht.</p> <p>Bewertung der Anzahl: (Faustregel: bei >10 genauer ansehen ob vereinfachbar) 1-10: einfaches Programm ohne grosses Risiko 11-20: komplexeres Programm, moderates Risiko 21-50: komplexes Programm, hohes Risiko 50+: untestbares Programm sehr hohes Risiko</p>
McCabe Metrik - Vorteile/ Nachteile	<ul style="list-style-type: none"> + einfach zu berechnen + ergibt minimale Anzahl Testfälle für 100% Zweigabdeckung - vereinfacht zu stark - berücksichtigt nicht Verschachtelung von Bedingungen - berücksichtigt nicht zusammengesetzte Bedingungen - einfache switch-Anweisungen führen zu hohem Wert
Mediator (gegen hohe Kopplung)	Wenn man eine hohe Kopplung zwischen Objekten hat (n:n) kann man dies mit einem Mediator verhindern. Der Vermittler (Mediator) trägt die Verantwortung zwischen den Komponenten. Er ist eine Hilfs-Instanz. z.B. wenn ein Font ändert wird dies nur dem Vermittler gesagt und dieser teilt sein Wissen dann den anderen Komponenten mit. Der Mediator bündelt gleichwertige Komponenten. <i>Unterschied zu Facade: Facade, ist über mehrere Subsysteme gestellt (die evtl. legacy sind). Eine Facade würde man bei einem neuen Projekt nicht benutzen. Subsysteme bei einer Facade sind nicht gleichwertig.</i>
Memento (ermöglicht z.B. undo Mechanismus)	Wird häufig beim Design von grafischen Anwendungen eingesetzt um ein "UNDO"-Vorgang anzubieten. Man will einen Zustand möglichst exakt zurücksetzen. D.h. man will quasi eine Moment-Aufnahme eines Objekts machen, daher der Name "Memento". Der Inhalt des Memento wird wegabstrahiert und vom Objekt selbst gehandelt.
Metriken	<p>Typen von Metriken:</p> <ul style="list-style-type: none"> Produktmetriken <ul style="list-style-type: none"> Zyklische Komplexität Projektmetriken <ul style="list-style-type: none"> Beispiel in Folien (Bug Reports) ist nicht gut investierte Arbeitszeit alles was der Projektmanager wissen will
Metriken - Afferent Coupling	 <p>Anzahl Klassen ausserhalb eines Packages, die von Klassen innerhalb des Packages abhängen.</p>
Metriken - Efferent Coupling	 <p>Anzahl Klassen innerhalb eines Packages, die von Klassen ausserhalb des Package abhängen.</p>
Metriken - Instabilität	<p>Instabile Pakete hängen von stabilen Paketen ab. D.h. Pakete mit wenig/keinen Abhängigkeiten sind instabil.</p> <p>I = (AfferentCoupling / EfferentCoupling+AfferentCoupling)</p>

Begriff	Code Snippet / Erklärung
Metriken - LCOM	<p>LCOM = Lack of Cohesion of Methods*.</p> <p>$(\text{AnzahlMethoden} - \text{Mittelwert}(\text{AnzahlMethodenDieAufAttributZugreifen})) / (\text{AnzahlMethoden} - 1)$</p> <p>Typische Beispiele:</p> <ul style="list-style-type: none"> D.h. wenn jede Methode nur auf ein Attribut zugreift → LCOM* = 1 → schlechte Kohäsion jede Methode greift auf jedes Attribut zu → LCOM* = 0 → maximale Kohäsion Getter/Setter verschlechtern LCOM*
Metriken - MENSCH	<p>Statische Code Analysis Tools können nicht überprüfen:</p> <ol style="list-style-type: none"> 1. sinnvolle Methoden/Klassennamen ← Aussagekraft 2. Verständlichkeit (Warnsignal, wenn nicht alle den Code verstehen) 3. Übereinstimmung mit Architektur-Ideen & Diagrammen 4. Code Smells (die üblichen verdächtigen, nicht alles) <p>1. evtl. auch sinnvolle Kommentare ← erklärt Algorithmus z.B.</p>
Metriken - Normalized Distance from Main Sequence	<p>$D = \text{Abstractness} + \text{Instabilität} - 1$</p>  <p>Figure 20-13 Zones of Exclusion</p> <p>D sollte möglichst gering sein bei gutem Package Design.</p>
Metriken - Tools	<ul style="list-style-type: none"> STAN: Abhängigkeiten finden zwischen Paketen SonarQube: Automatisches analysieren von Quellcode nachdem er Committed wurde. EclipseMetrics: Verschiedene Code Metriken
Metriken - Traditionell	<ul style="list-style-type: none"> Umfangsmetriken: Anzahl Codezeilen, Anzahl Unterprogramme etc. Logische Strukturmetriken: Zyklomatische Zahl von McCabe Datenstrukturmetriken: Anzahl Variablen, ihre Gültigkeit & Dauer Stilmetriken: Namenkonventionen Bindungsmetriken: Kohäsion & Kopplung, z.B. Anzahl Packages ausserhalb von denen Klassen im Package abhängen → Efferent Coupling
Metriken (Liste)	<ul style="list-style-type: none"> Number Of Classes (NOC) - wie der Namen sagt Number Of Children (NSC) - Anzahl direkter Unterklassen einer Klasse, Klasse die Interface implementiert zählt als Unterklasse dieses Interfaces. Number Of Interfaces (NOI) - wie der Namen sagt Depth Of Inheritance Tree (DIT) - Distanz der Klasse zur Klasse Object in der Vererbungshierarchie. Number of Overriden Methods (NORM) - Totale Anzahl der überschriebenen Methoden Number of Methods (NOM) - wie der Namen sagt Number Of Fields (NOF) - wie der Namen sagt Total Lines of Code (TLOC) - Anzahl Codezeilen, zählt nur Zeilen die nicht leer sind. Zeilen mit "/*" werden gezählt. Kommentarzeilen werden nicht gezählt. Auch KLOC genannt ($TLOC = KLOC * 1000$) Method Lines of Code (MLOC) - Zählt Codezeilen innerhalb Methoden. Zählt Zeilen mit nur "/*". Specialization Index = Mittelwert von $NORM * DIT / NOM$ Cyclomatic Complexity (McCabe Metrik, CC) - Zählt Anzahl Wege durch Methoden, siehe McCabe Metrik Weighted Methods per Class (WMC) - Summe von McCabe für alles Klassen Lack of Cohesion of Methods (LCOM*) - siehe → Metriken - LCOM Afferent Coupling - siehe Metriken Efferent Coupling - siehe Metriken Instability - siehe Metriken Abstractness - Anzahl abstrakter Klassen / Anzahl Klassen eines Packages Normalized Distance from Main Sequence - siehe Metriken
Micro Tests Probleme	<p>Setup: Man muss viele, irrelevante Objekte konfigurieren bevor man dass Objekt testen kann welches einem interessiert.</p> <p>Variable Results: Wir erhalten nicht immer das selbe Resultat weil ein Downstream Collaborator nicht immer gleich zu funktionieren "scheint".</p> <p>Execution Speed: Das Ausführen von Tests dauert zu lange. Wir haben Timeout-Probleme.</p> <p>Spreading Activation: Es gibt Probleme die eigentlich bei anderen Objekten liegen, welche wir momentan gar nicht testen wollen aber von unserem Objekt verwendet oder berührt werden.</p> <p>Access to Data: Der Test hat wenig Einfluss über Inputs und Outputs.</p>

Begriff	Code Snippet / Erklärung
Nicht funktionale Anforderungen	<p>Nicht funktionale Anforderungen ergänzen Use Cases und Domain Modell.</p> <p>Typische Beispiel für nicht funktionale Anforderungen:</p> <ul style="list-style-type: none"> • Performance: "Antwortszeit für eine Produkt-Suche bei 100'000.." • Mengengerüst: "200'000 Artikel; 20000 gleichzeitige Besucher" • Sicherheit: "Intrusion Detection, Logging, Plausibility Checking" • Erweiterbarkeit: "Später automatischer Import von Lieferanten Daten" • Benutzerfreundlichkeit: "Produkt Manager Einführung innert 2 Tagen"
Observer Pattern	<p>Pattern zum beobachten von Daten —> Änderungen in GUI (blah.)</p> <p>Zyklische Beziehung zur Laufzeit (Cyclic Dependency) wieso? Das User-Frontend "hört" auf Änderungen im Daten Modell. Dieses ruft "alle Listener" auf ohne sie zu kennen, darunter auch User-Frontend. —> D.h. zyklische Beziehung, diese ist allerdings OK da die Beziehung "kontrolliert" ist und man so immer noch recht problem Änderungen am GUI machen kann ohne zu fürchten dass dies das darunter liegende Daten-Modell tangiert (oder umgekehrt) so lange die Schnittstellen immer noch gleich sind.</p>
Package Dependency Graph	<p>Interpretiert Abhängigkeiten zwischen Packages. Damit kann man eine Aussage zur Kopplung machen, man kann Zyklen erkennen und evtl. auch Code Smells wie z.B. Feature Envy. (Einzelne Klassen wären evtl. besser in einer anderen Package aufgehoben.)</p> <p>Der Dependency Graph sollte ein azyklischer, gerichteter Graph sein (DAG).</p>
Pareto-Prinzip	<p>10% der Klassen stellen 80% der genutzten Funktionalität zur Verfügung. Er reicht aus diese Kernklassen mit Softwareverträgen zu schützen.</p>
Projekt Automatisierung	<p>Bei der Projekt Automatisierung werden moderne SE-Tools wie Build Server, Analysis Server, Automatische Tests etc. eingesetzt um das Leben des Programmier einfacher zu machen und die Code Qualität zu verbessern.</p> <p>Typische Vorteile der Projekt-Automatisierung:</p> <ul style="list-style-type: none"> • Man findet Fehler früher durch den Einsatz von automatischen Tests (Probleme treten sofort auf beim Einchecken) • Software läuft nicht nur an einem Ort sondern auf allen PC's durch den Einsatz von einem CI-Server (z.B. Jenkins) —> Releases/Buils werden zentral erstellt nicht von jedem Dev. • Die Code-Qualität wird kontinuierlich verbessert und hat einen hohen Standard durch den Einsatz von Coding-Standards z.B. in Eclipse sowie Code-Analyse auf dem Build-Server durch Tools wie Sonar. • Man hat eine gute Projekt-Übersicht durch automatische Code-Dokumentations Tools (javadocs etc.) • Jeder Entwickler programmiert mit den gleichen Code-Standards/Styles —> Code Checkstyle in IDE
Projekt Automatisierung Typische Risiken	<ul style="list-style-type: none"> • Lack of Deployable Software • Late discovery of defects • Lack of project visibility (people not notified about changes) • Low quality software
Qualitätsmodell - ISO9126	<div> <div> <p>Qualitätsmodell nach ISO 9126 (ISO25000)</p> </div> </div>
Redmine	<p>Tool zur Verwaltung von Aktivitäten, Arbeitspaketen, Arbeitszeiten.</p> <ul style="list-style-type: none"> • Man kann verschiedene Tickets gruppieren und einem anderen Ticket zugewiesen werden, damit kann man Use Cases 1:1 abbilden. • .. <p>todo: evtl. redmine mal noch ansehen, KA ob wichtig für Prüfung.</p>
Refactoring	<p>Vorgehen bei grösseren Refactoring Aufgaben: Schritt für Schritt und nach jedem Schritt immer automatische Tests durchlaufen lassen. (gibt 1 von 2 Punkten, sollte noch erweitert werden TODO)</p>

Begriff	Code Snippet / Erklärung
Refactoring - Divide & Conquer Strategie	Man soll in möglichst kleinen oder übersichtlichen/logischen Schritten refactoren. Dazu kann man eine Method z.B. in mehrere kleinere Methoden aufteilen. Dadurch wird es einfach nur einen übersichtlichen Teil aufs Mal zu refactoren. Kurz gesagt man soll jeweils möglichst kurz im "nicht-kompilierbaren" Status sein.
Refactoring - Typischer Ablauf	<ol style="list-style-type: none"> 1. Verifizieren dass alle automatisierten Tests durchlaufen 2. Entscheiden welchen Code man ändern möchte 3. Das Refactoring an diesem Code implementieren 4. Die automatisierten Tests (Microtests) durchlaufen lassen zum verifizieren dass man nichts kaputt gemacht hat. 5. So lange weiter machen bis das Refactoring komplett ist oder wenn etwas kaputt ist zu einem früheren Status zurückkehren
Refactoring - Ziele	<ul style="list-style-type: none"> • Duplizierten oder toten Code entfernen • Komplexen Code vereinfachen • Unklaren Code erklären/kommentieren/klarer machen <p>und dabei:</p> <ul style="list-style-type: none"> • keine Funktionalität kaputt machen (break) • nichts verschlechtern • kein neues Verhalten und keine neue Funktionalität einführen
Requirements	Siehe —> Anforderungen
Requirements Review	Siehe —> Anforderungen Erarbeiten
RUP Dokumente pro Phase	<p>Inception:</p> <ul style="list-style-type: none"> • Vision • Projekt-Eckwerte • IT-Landschaft • Annahmen und Einschränkungen • <i>(optional: Business Rules & Processes)</i> <p>Elaboration:</p> <ul style="list-style-type: none"> • Use Cases • Use Cases Diagramm • Architektur Entscheidungen • GUI Entwürfe / Wireframes • Domain Modell • Architektur Schichten Diagramm • Nicht funktionale Anforderungen • Glossar • Meilensteine • Test/QA Plan • Zeitaufschreibung • Risiko Liste • Liste der Stakeholder • <i>(optional: Zustandsdiagramme)</i> • <i>(optional: System Sequenz Diagramme)</i> • <i>(optional: User Groups Personas)</i> • <i>(optional: Aktivitäts Diagramme)</i> • <i>(optional: Gantt Chart)</i> • <i>(optional: Design by Contracts)</i> <p>Construction:</p> <ul style="list-style-type: none"> • Deployment Diagramm • Test Protokolle • Daten Modell • Metriken Resultate • (AUCH: Code & Dokumentation) • <i>(optional: Design Klassen Diagramme)</i> • <i>(optional: Migrations Plan)</i> • <i>(optional: Sequenz Diagramme)</i> • <i>(optional: Timing Diagramme)</i> <p><i>*Optional = Abhängig von Projekt, manchmal nötig.. gem. Grafik von D. Keller</i></p>
RUP Hauptdokumente	<p>Requirements: Das will der Kunde / Das wird der Kunde kriegen / Das wollte der Kunde (A)</p> <p>Projektplanung: So wollen wir vorgehen / So gehen wir vor / So hatten wir es geplant (A)</p> <p>Software Architektur Dokumentation: So bauen wir es / So haben wir es gebaut</p> <p>Projekt Nachverfolgung: Hier stehen wir im Moment / So ist es</p>

Begriff	Code Snippet / Erklärung
RUP Phasen (Rational Unified Proess)	RUP-Phasen: <ul style="list-style-type: none"> • Inception: Dient dem Ausformulieren einer Vision. • Elaboration Es wird ein Architektur-Prototyp erstellt. • Construction Man konzentriert sich auf das Entwickeln des Produkts. Nach der Konstruktions-Phase sollte man 4 Produkte vorweisen können: Code/Software, Dokumentation (inkl. User-Manuel, Wartungshandbuch), Test Suite und Support Dokumente. • Transition Man übergibt das Produkt dem Kunden.
Schichtenarchitektur - Asymmetrisch	Wieso: Nur von oben nach unten —> deshalb asymmetrisch. Dadurch wird die Kopplung verringert.
Schichtendiagramm	Man sieht die Packages, Übersicht über die Code-Klassen
Scope	siehe —> Funktionsumfang, wird oft durch Domain Model, Use Case Diagram definiert
Scrum Artefakte	<ul style="list-style-type: none"> • Product Backlog (geordnet nach Priorität, ähnl. todo-liste, liste von Anforderungen, dient als Input für Iteration.) Enthält Aufwandsschätzung vom Team. Dingen im Product Backlog müssen in Sprint platz haben.. Wenn grösser als Sprint —> “Epic” dann muss aufgeteilt werden. • Sprint Backlog (was soll alles in diesem Sprint erledigt werden, wird im Planning Meeting entschieden). Man überlegt sich wie man Sprint Backlog in einzelne Tasks (erledigbar innerhalb eines Tages) aufteilen kann. —> kommt auf Scrum-Board. • Inkrement (Stück Software) ist potentiell auslieferungs-fähig. == Potential Shippable Product • Impediment Backlog (Hindernis) - Liste von Dinge die Entwicklungsteam bei Arbeit behindern
Scrum Ereignisse	<ul style="list-style-type: none"> • Sprint (dauert höchstens 30 Tage, wenn fertig —> nächster Sprint). • Planungs-Meeting (findet am Anfang des Sprints statt), im Meeting 1 befindet sich Product Owner + Team + ScrumMaster. Team (Programmierer) entscheidet wie viel gemacht wird (nicht Product Owner und nicht ScrumMaster). Im Meeting 2 braucht es keinen Product Owner. • Daily Scrum Meeting findet vor dem Task-Board statt, es wird diskutiert was erledigt wurde, wer wo dran ist etc. Dauert ca. 15min. • Sprint Review am Ende des Sprints gibt es diese Review. Der ScrumMaster und das Team sowie weiter Stakeholders sind dabei. Man zeigt was man in diesem Sprint erreicht hat. D.h. man zeigt am Kunden was man jetzt hat. (Inkrement) • Sprint Retroperspective am Ende eines Sprints (ScrumMaster + Team, ohne Product Owner). Es geht darum wie das Team seine Arbeit verbessern kann. Daraus kann man einen Impediment Backlog bilden. Wer versucht was zu verbessern. Was gibt es zu verbessern.
Scrum Team	<ul style="list-style-type: none"> • Product Owner (verantwortlich für Produkt-Backlog), alle Anforderungen laufen über den Product-Owner. (wer was will muss sich an ihn wenden) • Scrum Master (Team Chef?!) • Development Team. Das Team ist Selbst-Organisierend, d.h. es gibt keinen Architekten.
Separation of Concerns (SoC)	Klare Aufteilung der Zuständigkeiten: Immer nur ein Thema pro Klasse/Methode, das ergibt dann auch die erwünschte hohe Kohäsion. <p>Präzisierung: immer nur ein Thema pro Methode, alles andere ist Unsinn, keine Ausnahmen manchmal mehr als ein Thema pro Klasse (weil man öfter mal eine Klasse mit unterschiedlichen Fähigkeiten ausstatten will), aber aufpassen, Klasse nicht überladen.</p>
Singleton (Faking & Mocking)	Möglichkeiten mit Singletons umzugehen <ul style="list-style-type: none"> • Extract & Override: referenziert zum awkward collaborator • Alternate Construction: des Problem-Objekts als fake object. • Set Fake Singleton: anstelle des awkward singleton • Redirection: der Aufrufe ins real oder fake object
Slipping	“Slipping” nennt man es wenn seinem Code gefälschte Objekte “unterjubelt” (übergibt).
Slipping Patterns	<ul style="list-style-type: none"> • Inversion of Control: Anstatt dass Klasse A Klasse B kontrolliert, kehren wir die Beziehung um, Klasse B kontrolliert jetzt Klasse A. • Dependency Injection: Anstatt das wir provider = new CustomProvider(); machen, schreiben wir neu: provider = Provider.getProvider();. Dadurch ist der Provider in der Kontrolle der Provider Implementation. • Slipping a Fake: Gleiche Technik wie Dependency Injection aber unterschiedliches Motiv. alt: provider = new CustomProvider(); neu: provider = getProvider(); Wir verwenden hier eine statische Methode so dass wir diese extrahieren und überschreiben können.
Slipping Techniken	<ul style="list-style-type: none"> • Argument Slip: Die getestete Method akzeptiert unser Fake Object als Argument. • Parameterize Method: Wenn die getestete Methode den Collaborator instanziiert oder eine globale Variable verwendet, extrahieren wir sie in eine parametrisierte Methode. • Constructor Arguement Slip: Wir übergeben dem Konstruktor der Klasse ein fake collaborator anstelle eines echten, awkward collaborators. • Alternate Constructor: Wenn der Konstruktor für die Zusammenarbeit zuständig ist bauen wir einen alternativen Konstruktor der ein fake übergibt. Dies nennt man auch “Overloading”. • Passing a factory: Wen die Klasse mehrere awkward collaborators hat sollten wir der Klasse eine fake factory übergeben die dann alle awkward collaborator fakes erstellt.

Begriff	Code Snippet / Erklärung
Software prüfen	<p>Wie kann man Software prüfen?</p> <ul style="list-style-type: none"> • Testen • Reviews machen • Formale Verifikation (mit Mathematik dahinter, Korrektheitsbeweis) • Metriken (d.h. Kennzahlen, z.B. lines of code) • Dynamische Analysen (Test-Abdeckung von Code)
Technical Dept	<p>Wenn man Refactoring aufschiebt und nicht regelmässig durchführt entsteht technical dept. D.h. Code wird unhandlich, kompliziert und/oder unsauber.</p> <p>Entstehung: Technical Dept entsteht normalerweise weil z.B. ein Kunde eine Änderung dringend braucht und man keine Zeit hat diese schön zu refactoren. (Sollte man jedoch unbedingt nachher tun, langfristig lohnt sich eine grosse Menge an Technical Dept nicht. —> teuer)</p> <p>Folgen: Auf Technical Dept müssen Zinsen gezahlt werden: 1. Schlecht wartbar, 2. Mehr Aufwand bei Änderungen 3. Es wird schwer Zeiten einzuschätzen für neue Features da man sich immer wieder mit der Technical Dept rumschlagen muss.</p> <p>Big Picture Folgen von Technical Debt: 1. High Employee Turnover, 2. Poor Adaptability to market changes, 3. Low employee motivation & productivity, 4. Poor team interaction & cohesiveness</p>
Test Coverage - Aussagekraft?	<p>Test Coverage ist die statistische Abdeckung von Tests zu Code, da die Variablen jedoch einen Weite-Raum haben sind unterschiedliche Ausführungen immer noch möglich. Dazu kommt dass Concurrency bei der Test-Coverage nicht beachtet wird. Und Test Coverage sagt auch noch nicht über die Qualität der geschriebenen Tests aus.</p> <p>D.h. tldr: Test Coverage gibt einem einen Anhaltspunkt/Richtwert und hilft einem Lücken in der Test Abdeckung zu finden.</p>
Test Driven Development	Siehe AUSDRUCK - TDD TESTAT
Tier	<p>Physikalisch getrennt z.B. Webserver und Datenbankserver.</p> <p>Nachteil von Tiers? Weil langsamer / Performance / teuer / fehleranfälliger / Verfügbarkeit / Sicherheit</p> <p>Vorteil von Tiers: Skalierbarkeit / Performance / Weniger Komplexität / Verteiltes & Koordiniertes Arbeiten</p>
Use Case - Beschreibung	<p>User Cases sind Beschreibungen, klar aus Benutzersicht, also gut mit Kunden kommunizierbar. Sie beschreiben Funktionalität im Kontext (z.B. warum, was in welcher Reihenfolge, wie oft?). Use Cases können gut zum Abstecken des Funktionsumfang (Scope) verwendet werden.</p> <p>Use Cases gibt es in verschiedene Detaillierungs-Geraden:</p> <ul style="list-style-type: none"> • brief: drei Zeilen • casual: eine halbe Seite • fully dressed: ein bis zwei Seiten mit vorgegeben Abschnitten
Zones of Exclusion	 <p>Figure 20-13 Zones of Exclusion</p>
Zyklische Import Abhängigkeiten	Siehe —> Acyclic Dependency Prinzip
Zyklomatische Komplexität	Siehe —> McCabe Metrik (ist das gleiche).
ISO9126	Siehe Qualitätsmodell - ISO9126