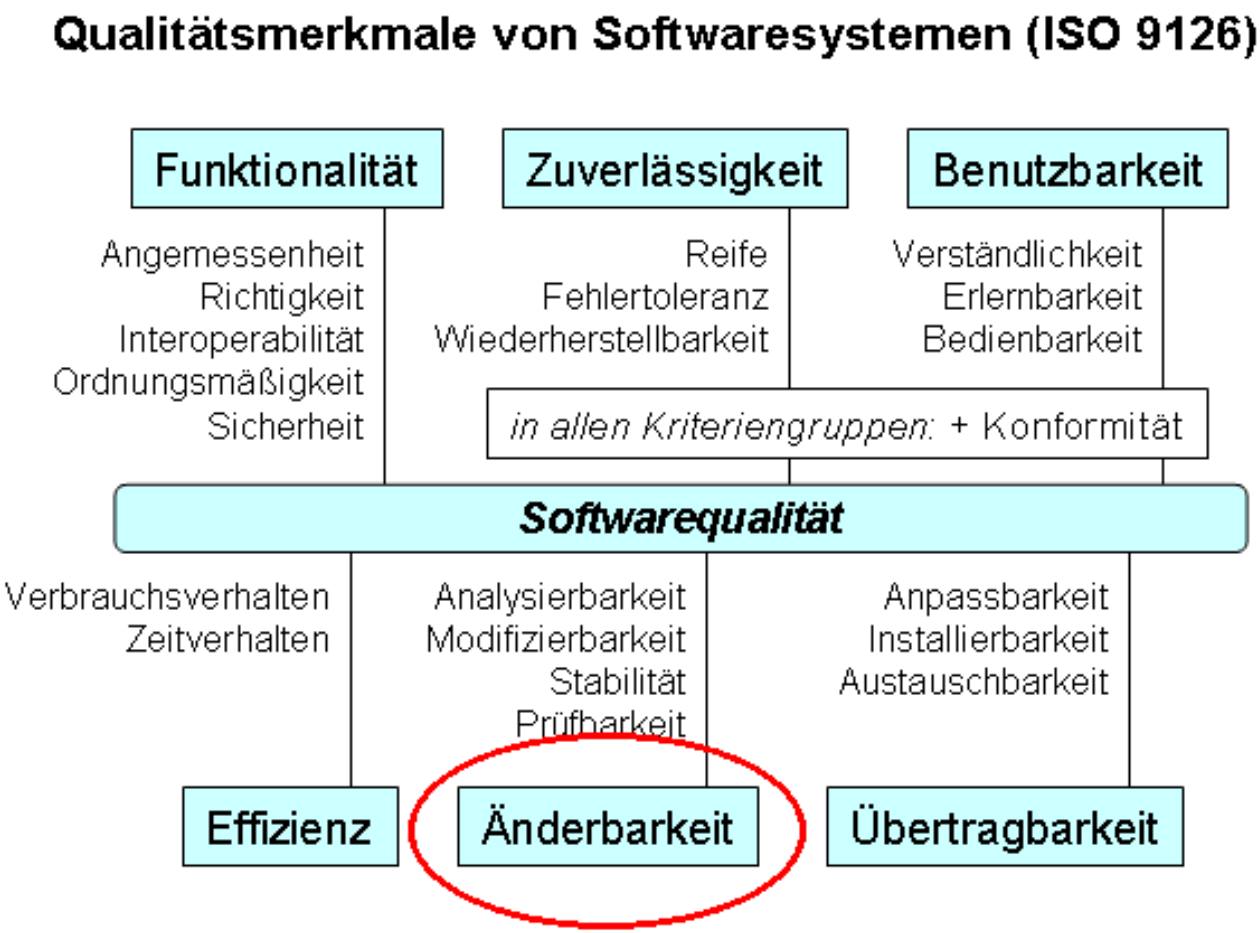
Qualität in der Softwareentwicklung

Erfahrungen aus 24x7, Gedanken und Hintergründe

Henrik Rößler

22.06.2015

Softwarequalität nach ISO 9126



Das Verhältnis von Qualität, Manuellen Tests und Entwicklern

Lemma

Alles, was nicht irgend jemand getestet hat, funktioniert auch nicht.

Stossgebet eines Entwicklers

Lieber Gott, lass mich im Zusammenhang manueller Tests weder „irgend“ noch „jemand“ sein.

oder

Beobachtung

Manuelle Tests, die Entwickler gern durchführen, dienen der Entspannung.

Risiken und Umfänge manueller Tests

- Das Risiko manueller Test liegt in der natürlichsprachlichen Beschreibung der Tests und in ihrer Ausführung.
  - Wir kennen das Problem der Eindeutigkeit von den Requirements: selbst genaueste Requirements können nicht verhindern, dass Anbieter und Nachfrager von Softwareentwicklungsleistungen unter ein und demselben Satz unterschiedliche Dinge verstehen.
  - Bei der Ausführung können sich in jedem Schritt eines Tests Ungenauigkeiten einschleichen, die entweder zu nichterkannten Problemen oder zu Fehlalarmen führen.
- Für jede Änderung an der Software sind sämtliche bisherigen Tests zu wiederholen und jene neue Tests auszuführen, die die neue Funktionalität sicherstellen.

Tätigkeitsfelder

1. Die Dinge richtig tun!
2. Die richtigen Dinge tun!

Änderbarkeit von Software

Lemma

Das Qualitätsmerkmal **Änderbarkeit von Software** ist per se nicht messbar.

`http://blog.ovidiubokar.com/sites/default/files/images/wtfs_per_minute_thumb.jpg`

Ziel

Ziel

Jede/r, die/deren mit Quellcode zu tun hat, muss jedes Stück Quellcode des Projektes in angemessener Zeit verstehen.

Es geht um die Fähigkeriten

- Code zu verstehen.
- Code einfach zu schreiben.
  - **K**eep **I**t **S**mall and **S**imple, **D**ont **R**epeat **Y**ourself, **C**onvention **o**ver **C**onfiguration
  - **S**ingle **L**evel of **A**bstraction, **S**eparation **o**f **C**oncerns, **P**rinciple **O**f **L**east **S**urprise
  - **S**ingle Responsibility Principle, **O**pen Closed Principle, **L**iskov Substitution Principle, **I**nterface Segregation Principle, **D**ependency Inversion Principle

Was man in jedem Projekt sofort beginnen kann

- den Quellcode einheitlich, automatisch formatieren,
- Metriken messen,
- für jede Änderung automatisierte Unit-Tests einbauen, die Testabdeckung messen und
- sich das Leben mit einer Contineous Integration erleichtern.

Argument für Formatierung

Beobachtung

Auf den ersten Blick scheint Formatierung ein völlig untergeordnetes Thema zu sein, es kann jedoch sicherheitsrelevant sein.

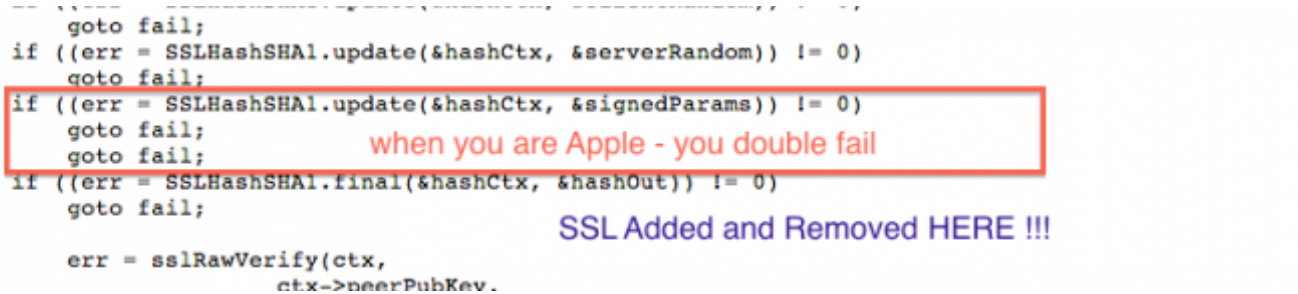
opensource.apple.com/source/Security/Security-55471/libsecurity_ssl/lib/sslKeyExchange.c?txt

```
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;

if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

    err = sslRawVerify(ctx,
                       ctx->peerPubKey,
                       dataToSign,                  /* plaintext */
                       dataToSignLen,               /* plaintext length */
                       signature,
                       signatureLen);
    if(err) {
```

when you are Apple - you double fail

SSL Added and Removed HERE !!!

Varianten zur Vermeidung solcher Fehler



1. Einzüge automatisch formatieren,
2. Klammern der Blockanweisungen setzen,
3. Mit Exceptions arbeiten,
4. „else if (...)“ statt „if (...)“ schreiben.
5. Automatisierte Unit-Tests mit Anzeige der Testabdeckung.

Werkzeuge für C#

- Artistic Style: `http://astyle.sourceforge.net/`
- NArrange: `http://www.narrange.net/`
- Beide Tools alleine haben mir nicht gefallen - eines ist in der horizontalen Formatierung stark, eines in der Vertikalen.
- Zusätzlich ist ein Script sinnvoll, das über alle cs-Files eines Verzeichnisses iteriert und rekursiv durch alle Unterverzeichnisse geht.
- Empfehlung: als Hook ins Commit einbauen — am Besten als Server-Hook

Werkzeug für Java

- Jalopy: `http://sourceforge.net/projects/jalopy/`
- Plugins für
  - Eclipse
  - IntelliJ
  - Maven

Gegenargumente

1. die IDE kann das auch.
   - Der manuelle Anstoß wird in der IDE zu gern vergessen.
   - Es ist schwierig sicherzustellen, dass alle Projekte und Entwickler mit identischer Einstellung hantieren.
2. Formatierung ist etwas ganz Individuelles und Kreatives.
   - Cleveren Implementierungen bieten einen sehr breiten Raum für Kreativität und Individualität — los gehts!
   - Wir arbeiten miteinander — die Kommunikation mittels Code ist die Stärkste!
   - Legen wir das gewünschte Format gemeinsam fest — man kann ja Verschiedenes ausprobieren und argumentieren!
3. Bei vertikaler Formatierung finde ich die Methoden in der Klasse nicht mehr.
   - Eine solche Klasse ist eindeutig zu groß. Es ist keine Klasse, sondern eher ein Paket — SRP, SLA, POLS . . . verletzt.

Programmiersprachen und Namenskonventionen

- Jede Programmiersprache hat ihre Namenskonventionen.
- Früher oder später brauchen wir mit Entwicklern außerhalb des konkreten Projekts kommunizieren (und sei es „nur“, dass jemand auf `http://stackoverflow.com/` etwas nachschlägt).
- Respektiert man die Namenskonventionen nicht, sind Transformationen nötig — das ist ein unnötiger Aufwand, bei einer Frage könnte sich die/ der Angefragte durchaus provoziert fühlen.
- Halten wir uns an die Standards! Wie oben dargelegt: es gibt genügend Raum für Individualität.

Sprachfeatures

- Die Namenskonventionen einer Programmiersprache haben etwas mit den technologischen Möglichkeiten der Entstehungszeit zu tun.
- Die durchgehende Großschreibung von Bezeichnern zur Kennzeichnung von Makros ist veraltet.
- Wenn in C der UnderScore ‚_‘ verwendet wurde um bestimmte Kennzeichnungen vorzunehmen, ist das heute nicht mehr nötig — wir haben IDEs mit Variablen-Highlighting.
- Die Verwendung des Preprozessors in C, C# und C++ bedeutet, in einem File 2 Sprachen zu mischen. Das wird mit der Zeit unübersichtlich — die Änderbarkeit geht verloren.
- Aus gleichem Grund mag ich keine jsps: sie sind eine Mischung aus HTML und Java in einer Datei.

Metriken als Regelwerk

1. Wir können die Einhaltung der Prinzipien von Clean Code nur schwer messen. Wir brauchen Hilfsgrößen — Metriken.
2. Die Metriken drücken keine funktionalen oder Performance-Fehler aus.
3. Sie drücken aus, wie schwer wir uns unsere Arbeit machen.

Werkzeuge

- Jenkins mit einer Menge Plugins – für Java: Checkstyle, Findbugs, PMD.
- Diese Plugins gibt es auch für Eclipse und IntelliJ – mit einheitlicher Konfiguration.
- SonarQube mit einer Menge Plugins für einzelne Sprachen: C#, Java, JavaScript, PHP, C++ – einige davon sind kommerziell.
- Für mich macht es keinen Sinn, über die Bewertung zu diskutieren bzw. die Bewertung der Verstöße abzuschwächen.
- Niemand wird eine Abschwächung rückgängig machen.
- Der Code bekommt keine bessere Qualität, wenn man die Limits herabsetzt.

Zyklomatische Komplexität - Was ist das? Wie wird sie berechnet?

- Wird als Wege durch den Code **einer Methode** gemessen:
  - Start mit 1.
  - ein **if**
    - multipliziert mit der Zahl der möglichen Ausgänge
    - beide Zweige werden einzeln weiter verfolgt
    - am Ende wird mit der höheren Zahl weiter fortgesetzt
  - eine Schleife (**for**, **foreach**, **while** und **do**) erhöht um 1
  - ein **case** wird analog **if** behandelt, nur dass mehr als zwei Zweige weiter verfolgt werden müssen

Zyklomatische Komplexität - Was bedeutet ein Reissen des Limits?

- Die Grenze liegt im Allgemeinen knapp über 10.
- Ein Reißen dieser Grenze kann Hinweis auf Verletzung folgender Prinzipien sein
  - **K**eep **I**t **S**mall and **S**imple,
  - **S**ingle **L**evel of **A**bstraction,
  - **S**ingle **R**esponsibility **P**rinciple und ggf. auch
  - **D**ont **R**epeat **Y**ourself
  - Eventuell werden auch Fehler nicht per Exception, sondern per Rückgabewert behandelt.

Zyklomatische Komplexität - Wie heile ich einen Verstoss?

- Das Maß fordert die Entwickler auf, einen Teil der Komplexität hinter weiteren Methodenaufrufen zu verbergen:
  - Komplizierte boolesche Konstruktionen lassen sich in Methoden auslagern, die ein **bool** zurückgeben.
  - Wiederholungen boolesche Konstruktionen lassen sich zusammenfassen.
  - **switch-case**-Konstrukte könnten möglicherweise durch Polymorphismus ersetzt werden (ggf. mit Reflection)
  - lange **if**- oder **if-else**-Ketten könnten auch eine Aufforderung zum Polymorphismus bedeuten.

Coupling einer Klasse - Was ist das? Wie wird das Coupling berechnet?

- Von wieviel anderen Packages ist **eine Klasse** abhängig?
- Zählen der **using**-Direktiven/ **import**/ **include**
- ohne die **using-alias**-Direktiven (C#)
- Hinzu kommen die Zahl der Klassennamen, die im Quelltext mit Namespace verwendet werden

Coupling einer Klasse - Was bedeutet ein Reissen des Limits?

- Das Prinzip **S**ingle **L**evel of **A**bstraction könnte verletzt sein.
- Mein erster Verdacht würde darauf fallen, dass sich die Klasse
  - sowohl um die Instrumentierung zur Lösung einer Aufgabe,
  - um die Instrumente selbst,
  - als auch um die Details der Instrumente kümmert.

Coupling einer Klasse - Wie heile ich einen Verstoss?

- Das Maß fordert die Entwickler auf, die Aufgabe der konkreten Klasse zu überdenken.
- Mögliche Schritte:
  - Teilaufgabe(n) bilden,
  - Namen für diese Teilaufgabe(n) finden,
  - für jede identifizierte Teilaufgabe eine Hilfsklasse minimal notwendiger Sichtbarkeit bilden,
  - Code verschieben und
  - **using**-Direktiven minimieren (**import**/ **include**) ...

Duplicated Code - Was ist das? Wie wird er ermittelt?

Es gibt verschiedene Ansätze, SonarQube nutzt einen ziemlich fortschrittlichen:

- Der Code eines Projekts wird in einen abstrakten Syntax-Baum überführt.
- Alle Teilbäume ab einer minimalen Größe werden versucht mit anderen Teilbäumen in Deckung zu bringen.
- Das ist auch für Unit-Tests kritisch, da diese häufig ein ganz ähnliches Aussehen haben.
- Auch das Ändern einzelner String-Literale oder Zeilenumbrüche helfen nicht zum Fix.

Duplicated Code - Was bedeutet ein Reissen des Limits?

- Zunächst zeigt diese Metrik einen Verstoß gegen das Prinzip **D**ont **R**epeat **Y**ourself.
- Ich würde vermuten **S**eparation **o**f **C**oncerns liegt auch im Argen.
- Mehrere Klassen und/ oder Methoden, Packages etc. nutzen eine gemeinsame Struktur und damit ein gemeinsames Konzept.
- Wiederverwendung durch **Copy and Paste** ist gefährlich, da bei nötigen Änderungen wahrscheinlich nicht alle Kopien gefunden werden.

Duplicated Code - Wie heile ich einen Verstoss?

- Das Maß fordert die Entwickler auf, ein gemeinsames Konzept zu extrahieren:
  - mehrere Codeschnipsel innerhalb einer Methode oder Klasse → private Methode extrahieren,
  - mehrere Codeschnipsel innerhalb mehrerer Klassen in einer Hierarchie → protected Methode in eine Basisklasse herausziehen - ggf. Hilfsklasse nutzen,
  - mehrere Codeschnipsel innerhalb mehrerer Klassen in mehreren Hierarchien → Methode in Hilfsklasse auslagern,
  - identische Operationen mit unterschiedlichen Typen → Generics einführen, ggf. diese Typen in gemeinsame Klassenhierarchie schieben (sofern noch nicht passiert),
  - identische Hilfsklassen → vereinheitlichen
  - etc. . . .

Abgrenzung zu Abnahme-Tests

Unit-Tests	Abnahme-Tests
einzene Klassen/ kleine Gruppen von Klassen	Gesamtsystem
zu testende Unit	Zusammenspiel der Module
an einzelne Module gebunden	ans Gesamtsystem gebunden
ausschließlich funktionale Tests	auch Lasttests
hohe Breite der Abdeckung	geringe Breite der Abdeckung
geringe Tiefe der Abdeckung	hohe Tiefe der Abdeckung
Verantwortung beim Entwickler	Verantwortung bei QA/ Projekt
geringer Analyseaufwand	hoher Analyseaufwand
schnelle Fixes	Fixes dauern länger

Java: Kurzporträt junit

- „Urvater“ der automatisierten Unit-Tests
- Download `https://github.com/junit-team/junit/wiki/Download-and-Install`
- Tutorial `http://junit.org/`
- sehr gute Integration in Jenkins, Cobertura (Instrumentierung des Codes), Eclipse, IntelliJ
- OpenSource d.h. keine relevanten Nutzungsbeschränkungen

Java-Klassen für Unit-Tests

- Tests werden in Klassen organisiert
- gehören in eine eigene jar
- sind auf dem Produktionssystem nicht mit zu deployen
- alles, was Objektorientierung bietet, kann verwendet werden

Java-Methoden in Unit-Tests

- jeder Test ist eine **parameterlose public void Instanz-Methode** mit der Annotation **Test**
- Vorbereitungsmethoden pro Test: wie Tests, jedoch mit der Annotation **Before**
- Nachbereitungsmethoden pro Test: wie Tests, jedoch mit der Annotation **After**
- fliegt aus der Methode eine Exception, wird der Test als fehlgeschlagen bewertet
- daher wird für die Prüfung der Ergebnisse **Assert** verwendet

C#: Kurzporträt NUnit

- Download `http://nunit.org/index.php?p=download`
- Tutorial `http://nunit.org/index.php?p=quickStart&r=3.0`
- passable Integration in Jenkins, OpenCover (Patch der VM) und SonarQube
- eigenes Tool zur Ausführung der Tests
- OpenSource d.h. keine relevanten Nutzungsbeschränkungen

Nachteile:

- gewöhnungsbedürftige Integration in VisualStudio
- externes Tool zur lokalen Ausführung
- Anzeige der Testabdeckung über SonarQube möglich

C#-Klassen für Unit-Tests

- Tests werden in Klassen organisiert
- gehören in eine eigene Assembly, also eigene csproj als Bibliothek
- sind auf dem Produktionssystem normalerweise nicht mit zu deployen
- die Klasse mit den Tests bekommen das Attribut **[TestFixture]**, um für die Testausführung sichtbar zu sein
- alles, was Objektorientierung bietet, kann verwendet werden
- das Direktory-Layout ist so flexibel, wie **.csproj** und **.sln** das ermöglichen und auf Jenkins nachgebildet werden kann.

C#-Methoden in Unit-Tests

- jeder Test ist eine **parameterlose public void Instanzmethode** mit dem Attribute **[Test]**
- Vorbereitungsmethoden pro Test: wie Tests, jedoch mit dem Attribute **[SetUp]**
- Nachbereitungsmethoden pro Test: wie Tests, jedoch mit dem Attribute **[TearDown]**
- fliegt aus der Methode eine Exception, wird der Test als fehlgeschlagen bewertet
- daher wird für die Prüfung der Ergebnisse **Assert** verwendet

Das Problem:

- viele Methoden rufen Methoden fremder Verantwortungsbereiche auf
- sind das externe Bibliotheken oder das Laufzeitsystem ist das kein Problem, denn das Verhalten dieser Aufrufe ändert sich nur, wenn neue Versionen verwendet werden.
  - $\rightarrow$ bei Versionswechsel dieser externen Bibliotheken sind möglicherweise die Tests anzupassen
  - $\rightarrow$ anzupassende Tests sind ein Hinweis darauf, dass sich das Zusammenspiel mit dieser Bibliothek ändert
- werden interne Komponenten aufgerufen, kann sich täglich vieles ändern
  - das Verhalten
  - transitive Abhängigkeit auf weitere Komponenten

Umfassende Tests - Variation der Rückgabewerte von tieferen Aufrufen

- Unit-Tests spielen ihre Vorteile erst dann richtig aus, wenn sowohl über die Aufrufparameter als auch von tieferen Schichten zurückgelieferten Werte sinnvoll variiert wird.
- Stichworte sind
  - Methode der Grenzwert-Tests
  - Exceptions aus tieferen Schichten
- $\rightarrow$ wir brauchen das Verhalten der tieferen Schichten je Test unterschiedlich einstellen.

Mögliche Zugänge

Man könnte

1. unterschiedliche Implementierungen bereitstellen und die Tests jeweils gegen die passende Implementierung laufen lassen oder
2. eine mit unterschiedlichem Verhalten aufladbare Implementierung bereitstellen,
   - vor jedem Test das jeweils gewünschte Verhalten einstellen und
   - die Tests dagegen laufen lassen.
   - Gegebenenfalls könnte man die Aufrufe auch überwachen.

- Das letztere Konzept nennt sich Mocking.
- Für die Variation des Verhaltens wird im Allgemeinen Polymorphismus benutzt.

Java: easymock

- Download: `https://bintray.com/artifact/download/easymock/distributions/easymock-3.3.1.zip`
- Tutorial: `http://easymock.org/getting-started.html`
- OpenSource d.h. keine relevanten Nutzungsbeschränkungen
- Unterstützt **strict** und **nice** Verhalten.
- Syntaktisch werden Generics benutzt.

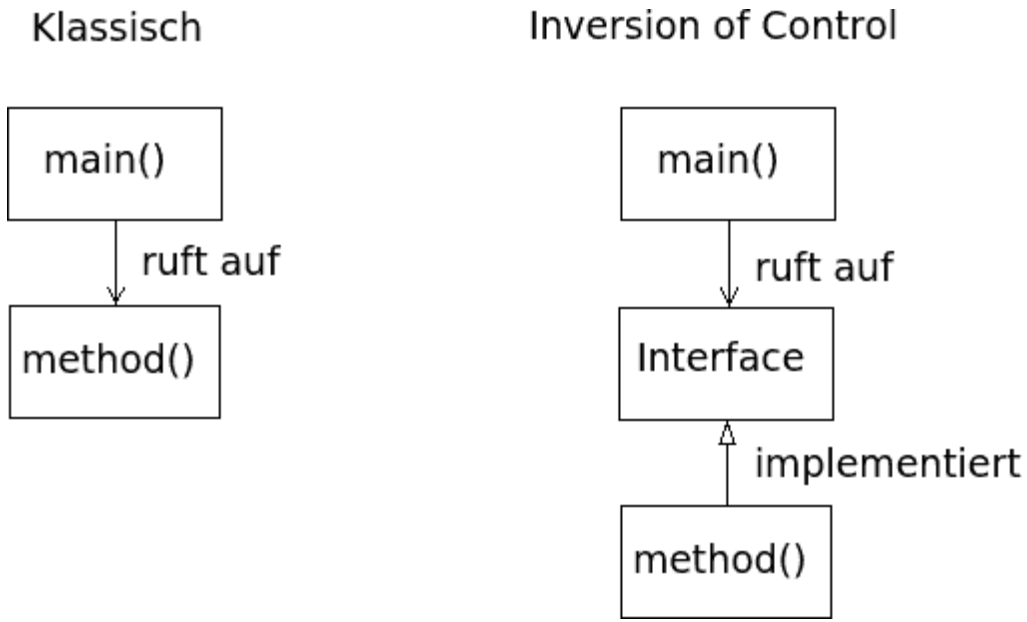
C#: Kurzporträt Moq

- Download:
  `https://code.google.com/p/moq/downloads/list`
- Tutorial:
  `https://github.com/Moq/moq4/wiki/Quickstart`
- OpenSource d.h. keine relevanten Nutzungsbeschränkungen
- Unterstützt **strict** und **nice** Verhalten.
- Wird mittels Lamda-Ausdrücken aufgeladen.

Polymorphismus

- Die Forderung nach Polymorphismus ist ein Hammer, denn in Java und C# können nur Instanzmethoden polymorph sein.
- In C# müssen diese Methoden auch noch als virtuell gekennzeichnet werden, in Java sind alle Instanz-Methoden automatisch virtuell.
- Klassenmethoden können nicht polymorph sein.
- $\rightarrow$ für einen Aufruf brauchen wir immer eine Instanz.
- $\rightarrow$ die aufrufende Klasse braucht immer eine Referenz auf diese konkrete Instanz.
- Interfaces und abstrakte Klassen sind Ausgangspunkte dafür.

Inversion of Control



- „ruft auf“ stellt auch die Richtung der Abhängigkeit dar.
- Durch das Einfügen des Interfaces (der abstrakten Klasse) wird diese Abhängigkeit in zwei Pfeile aufgebrochen.
- Der untere Pfeil dreht sich herum: **Inversion**.
- **main()** ist nicht mehr von **method()** abhängig.

Instanziierung

- beim Start die Referenz(en) auf die konkrete(n) Instanz(en) für ihre Abhängigkeit(en) mitgeben: **Dependency Injection**.
- Mock-Frameworks können die Instanz für die Tests zur Verfügung stellen.
- Für unsere Produktionssysteme brauchen wir uns etwas einfallen lassen.
- Manuell oder durch sogenannte IoC-Container.
- Manuell wird schnell unübersichtlich und man bekommt so etwas wie ein Singleton-Problem auf den Tisch.
- IoC-Container regeln auch Deployment- und Shutdown-Reihenfolgen.

Abnahmetests

- Hier bin ich schwach aufgestellt.
- Ich würde empfehlen, sich mit der Webseite von Fitnesse auseinander zu setzen: `http://fitnesse.org/` insbesondere mit der Seite: `http://fitnesse.org/FitNesse.UserGuide.ProjectDeathByRequirements`

Schlussworte

Danke für die Aufmerksamkeit

Bitte um Fragen