



INSTITUT
FÜR
SOFTWARE

Bessere Software –
Einfach, Schneller



HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
INFORMATIK

Lohnt sich testbasierte Software-Entwicklung ?

Prof. Peter Sommerlad

IFS Institut für Software

HSR Hochschule für Technik Rapperswil

Oberseestrasse 10, CH-8640 Rapperswil

www.ifsoftware.ch

Peter Sommerlad

peter.sommerlad@hsr.ch



● Arbeitsgebiete

- Refactoring Tools (C++, Ruby, Python, Groovy, PHP, JavaScript,...) für Eclipse
- **Incremental Development (make SW 10% its size!)**
- Modernes Softwareengineering
- Patterns
 - POSA 1 and Security Patterns

● Background

- Diplom-Informatiker (Univ. Frankfurt/M)
- Siemens Corporate Research - München
- itopia corporate information technology, Zürich (Partner)
- Professor für Informatik HSR Rapperswil, Leiter IFS

Credo:

● Menschen machen Software

- Kommunikation
- Feedback
- Mut

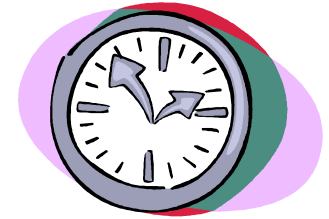
● Erfahrung durch Praxis

- Programmieren ist Handwerk
- Patterns aus der Praxis kapseln Erfahrung

● Pragmatisches Programmieren

- Test-Driven Development
- Entwicklungsautomation
- Einfachheit: Kampf gegen Komplexität

- **Was ist testbasierte Entwicklung?**
 - Pragmatische Praktiken
- **Fallbeispiel aus der Industrie**
 - C++ Framework für Web Applikationen
 - Historische Betrachtung
- **Vorteile und Risiken testbasierter Entwicklung**
 - Entwickler, Kunden, Anbieter
- **Zusammenfassung, Ausblick, Appell**



Was ist testbasierte Entwicklung?

Automatisierte Tests stellen die Software Qualität sicher

● Unit Testing

- Für jede Komponente/Klasse/Methode, die eine nicht-triviale Implementierung hat, werden eine oder mehrere Testmethoden implementiert.
- Die Tests sollten isoliert ablaufen. Benötigte Umfeld-Objekte werden mittels sog. „mock-objects“ simuliert, um Abhängigkeiten zu vermeiden.
- Oft werden Tests für Grenzfälle und das Normalverhalten erstellt.
- Überprüfen der **technischen** Qualität der Implementierung.

● Functional Testing

- Relevantes externes Verhalten wird auf Systemebene getestet.
- Testbarkeit durch Isolation von Teilsystemen mittels mock-objects verbesserbar.
- Überprüfen der **fachlichen** Qualität.

● Idealerweise: Test-First

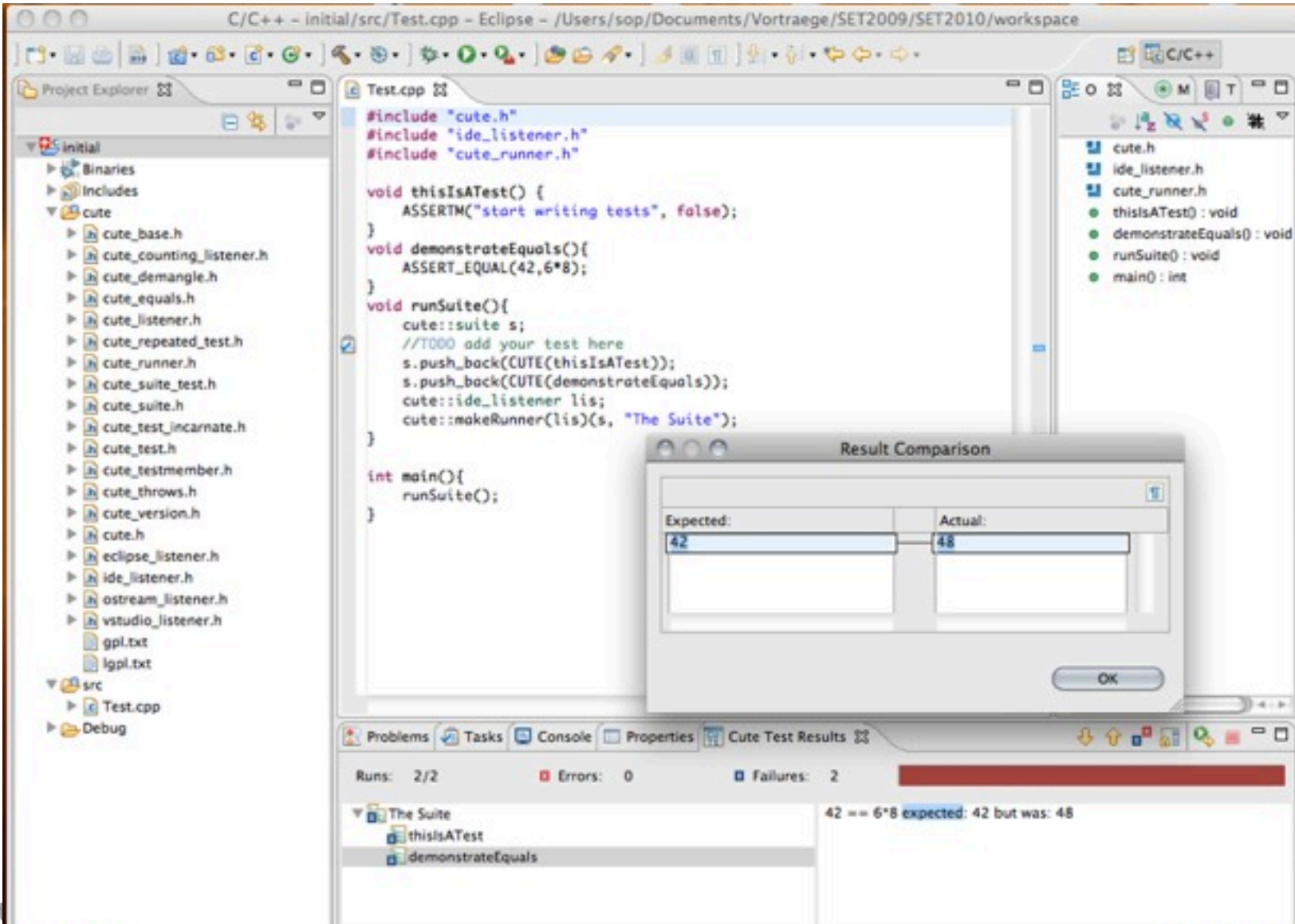
- Zuerst Test und Schnittstelle realisieren, dann implementieren.

- **Test eines Moduls (Übersetzungseinheit, = Quell-Datei)**
 - oder kleinere Einheit...
- **Test durch Programmierer selbst**
- **automatisch, wiederholbar**
- **vorausgesetzte Ergebnisse - Assertions**

Unit Testing Frameworks:

- **JUnit (Java), NUnit (C#), CUTE(C++), etc.**
 - www.junit.org
 - <http://ifs.hsr.ch/cute/updatesite> -> Eclipse CDT

CUTE Beispiel



The screenshot shows the Eclipse IDE with the following components:

- Project Explorer:** Shows a project named 'initial' with a sub-project 'cute' containing various header files like `cute_base.h`, `cute_counting_listener.h`, etc.
- Test.cpp:** Contains the following code:

```
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"

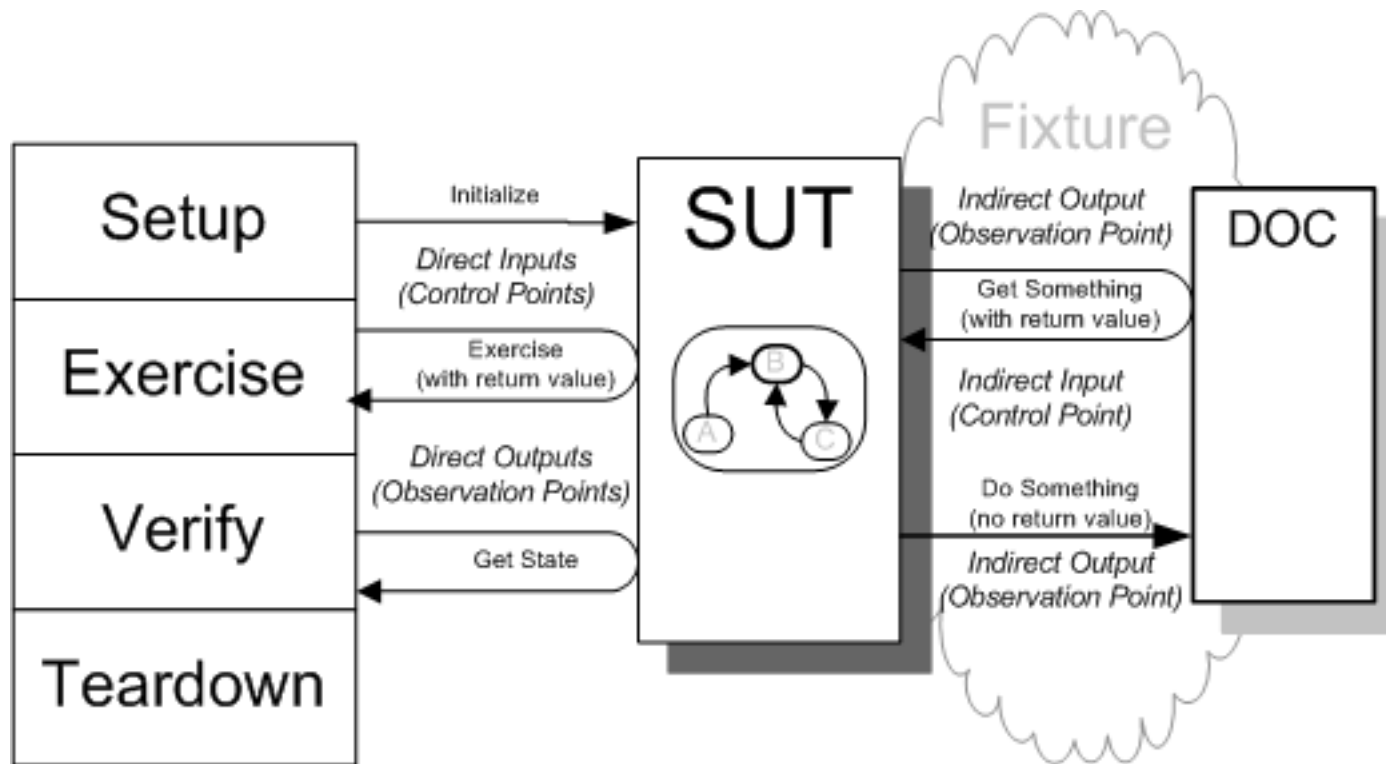
void thisIsATest() {
    ASSERTM("start writing tests", false);
}

void demonstrateEquals(){
    ASSERT_EQUAL(42, 6*8);
}

void runSuite(){
    cute::suite s;
    //TODO add your test here
    s.push_back(CUTE(thisIsATest));
    s.push_back(CUTE(demonstrateEquals));
    cute::ide_listener lis;
    cute::makeRunner(lis)(s, "The Suite");
}

int main(){
    runSuite();
}
```
- Result Comparison Dialog:** A modal dialog with two input fields. The 'Expected:' field contains '42' and the 'Actual:' field contains '48'. An 'OK' button is at the bottom.
- Bottom Status Bar:** Shows 'Runs: 2/2', 'Errors: 0', and 'Failures: 2'. A red progress bar is partially filled. Below it, the 'Cute Test Results' view shows a tree structure with 'The Suite' expanded to show 'thisIsATest' and 'demonstrateEquals'. The error message '42 == 6*8 expected: 42 but was: 48' is displayed.

Struktur eines Testfalls: Four Phase Test

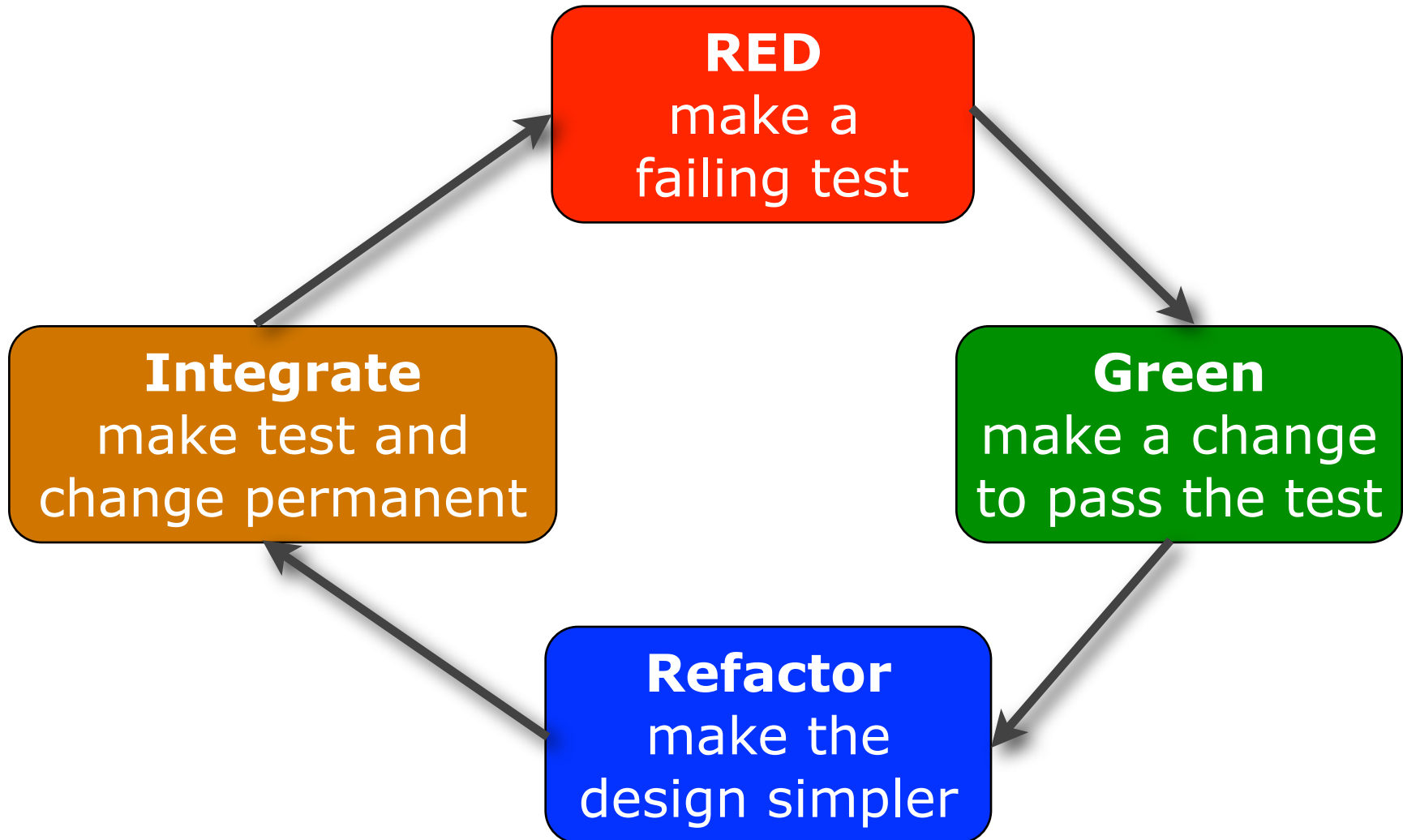


● **Quelle: xunitpatterns.com**

Pragmatische Prinzipien beim Unit Testing

- **Alles Testen was fehlschlagen kann**
 - keine Tests für "funktionslosen" Code
- **Alles Testen was einmal fehlgeschlagen ist**
 - bei Fehler zuerst einen Test schreiben, der den Fehler reproduziert, dann korrigieren
- **Neuer Code ist schuldig, bis Tests die Unschuld beweisen**
- **size(test code) >= size(production code)**
- **Tests sind integraler Bestandteil des Builds**
 - nicht weitermachen bevor Tests 100% laufen
- **Alle Tests laufen bevor man ins Repository eincheckt**

Noch besser TDD: Test-Driven Design



Fallbeispiel aus der Praxis



Grosses hochperformantes Framework für Web-Applikationen mit mehreren Kunden und Produktivsystemen im Einsatz seit ca. 1997 und bis heute weiterentwickelt und gepflegt.

Geschichte in den 1990ern

● **Web Application Framework in C++**

- Internet Banking Schweizer Grossbank
- Online Game mit ca. 10000 users
- Finanzinformationsportal mit 5 – 10000 gleichzeitig aktiven professionellen users
 - noch heute im Einsatz und aktiv weiterentwickelt

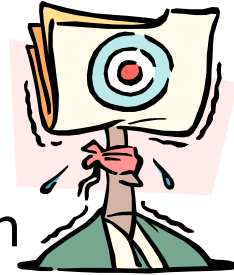


● **1997 – keine automatischen Unit Tests**

- jeder Entwickler/Applikation hat eigene Version, Divergenzgefahr
- Korrektur von Kernabstraktionen und Infrastruktur notwendig, aber zu riskant
- Refactoring-Stau

- **Furcht vor Änderungen am Framework**

- Risiko: Destabilisierung existierenden Codes
- Risiko: Change Propagation in „fertige“ Applikationen
- Risiko: Mehraufwand für wiederholtes Testen
- Risiken sind schwer einzuschätzen



- **Qualitätsmängel im Framework**

- FIXME Kommentare: quick hacks, trial and error
- Verwendung von C-legacy considered harmful: sscanf, vprintf, char *
- Teilweise unsauberes Verhalten im Multi-threaded Fall
- Teilweise unsauberes Memory Management

- **Änderungen erfordern zu viel Mut**

- und Angstschweiss, dass etwas kaputt geht

1998: Demo von Test-first Programming von Kent Beck und Erich Gamma

„Das müssten wir auch machen!“



- Wenn das Framework mit automatischen Tests abgestützt wäre, dann sollten Änderungen am Framework ohne Probleme machbar sein.
- Technische Varianten könnten gegeneinander abgewägt werden.
- Technische Verbesserungen könnten idealerweise ohne Implikation auf Applikationen erfolgen.
- Impakt-Abschätzung von (Schnittstellen-) Änderungen wäre möglich.
- Daily Build mit automatischen Tests verbinden.

Was hat es gebracht?

(1) 2000

- **Stabilität**

- Grenzfälle besser abgedeckt
 - z.B. 0, 1, viele, ganz viele, negativ ?



- **Refactoring im Framework wurde möglich**

- z.B. Reduktion von unnötigem Locking, optimiertes Memory Management

- **Portabilität auf andere Plattformen**

- AIX, Windows NT, Teilbereiche auf exotisches IBM Host OS (TPF)

- **Nachträgliche Implementierung von Tests teilweise aufwändig**

- erste Tests teilweise zu umfangreich
- Lernkurve!!! Huhn-Ei Problem bzgl. Refactoring

Was hat es gebracht?

(2) 2000

- **Austauschbarkeit von (ungünstigen) Implementierungen**
 - Elimination von C-Legacy und FIXMEs
- **Besseres Interface Design von neuen Dingen**
 - Test-first und Testability sind gute Ratgeber fürs Schnittstellendesign
- **Einfachere Flexibilisierung**
- **Vertrauen in „fremden“ Code bei Entwicklern gestiegen**
 - Konsolidierung von Code
 - reduziertes Risiko, bzw. Risiko einschätzbar, indem man Testcase schreibt, bevor man Änderung macht



- **Framework unter dem Namen COAST heute am IFS gepflegt, wird als Open Source verfügbar gemacht**
- **Applikationen an der HSR (Web SSO) und bei früherem Kunden in Produktion und Weiterentwicklung**
- **Intensives Refactoring auf modernes C++ im Rahmen der COAST Wartung**
 - Ziel: weitere Code-Vereinfachung und Qualitätsverbesserung
 - Nebeneffekte: Scons Plug-in für Eclipse CDT, weitere CDT plug-ins

Vorteile und Risiken testbasierter Entwicklung

anhand eigener Erfahrungen



Vorteile testbasierter Entwicklung (Entwickler)

- **Qualität des Codes gesteigert**
 - bei guten Testfällen, Grenzfälle abdecken
- **sicheres Refactoring wird ermöglicht**
 - kein Domino-Effekt bei Änderungen
- **gut testbarer Code hat besseres, einfacheres Design**
 - hohe Kohäsion, niedrigere Kopplung
- **Testcode macht Entwickler zum "Opfer" seines Schnittstellendesigns**
 - bessere und einfacher nutzbare Interfaces
- **Testcode dokumentiert Anforderungen**
 - Test-First macht das explizit: zuerst die Anforderung als Test kodieren
- **Debugger wird verzichtbar**
 - Bug wird im Testcase manifestiert, tritt nie wieder auf



Schwierigkeiten für Entwickler

- **Zeit, Coaching und Erfahrung nötig für gute Tests**
 - Support durch Kollegen und Führung!
- **Tests sind auch Code**
 - brauchen Refactoring
- **unnötige Tests behindern**
 - Plattformtests, "false positives" behindern
 - zu viele "Normalfälle" getestet
- **Tests müssen rasch ablaufen**
 - so oft wie möglich ausführen
- **Tests müssen isoliert sein**
 - keine Abhängigkeiten von anderen Tests und Infrastruktur
 - Mock-Objekte einsetzen
- **Tests für existierenden Code schreiben ist schwerer**
 - oft zu viele Abhängigkeiten im Code -> schlechtes Design
 - aber essentiell für Refactoring von "Altcode"
- **schlechte Tests sind Hinweis auf schlechtes Design**
 - **Heute gibt es gute Literatur zum Thema.**

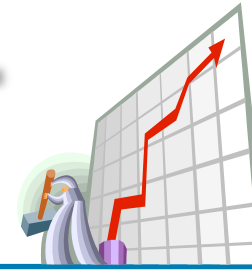




- **Qualität der Lösung ist besser**
 - oft schnellere, kostengünstigere Ergebnisse
 - keine lange Test- und Integrationsphase mit aufwendigen Korrekturen notwendig
- **Änderungswünsche können ohne Qualitätsrisiko rasch umgesetzt werden**
 - inkrementelle Entwicklung ist einfacher
- **Wartung und Weiterentwicklung günstiger**
 - Fehler treten nur einmal auf
 - Fehler werden durch Tests erkannt und nicht erst durch Anwender
- **Lebensdauer der Software ist länger**

- **Auf den ersten Blick sieht testbasierte Entwicklung teurer aus**
 - mehr Code zu schreiben
 - Refactoring bedeutet funktionierenden Code zu ändern
- **Qualität kann "zu gut" sein**
 - interne QA Abteilung "überflüssig"
 - "Reifezeit" in QA/Systemintegration unnötig
 - rasches häufiges Deployment nicht üblich
- **Einfachheit von Änderungen kann zu Oszillation in den Anforderungen führen**
 - innere Konflikte kommen schneller heraus





- **Hohe Qualität führt zu anerkannten Lösungen**
 - Kundenzufriedenheit
- **Zufriedene Entwickler**
 - weniger Stress durch Sicherheitsnetz aus Tests
- **Leichtere Einarbeitung neuen Personals**
 - Tests verhindern "kaputtmachen"
- **innere Qualität des Codes kann verbessert werden**
 - Zukunftssicherung durch Refactoring und proaktive Wartung
 - Gute Testbarkeit -> gutes Design!
- **Portierbarkeit auf andere Plattformen**

Risiken für Anbieter von Software-DL



- **Qualität geschätzt aber Preis nicht gezahlt**
 - Kunde erkennt Agilität und nutzt Anbieter aus
- **zufriedene Kunden kündigen Support & Wartungsverträge**
 - zu gute Qualität "rächt" sich
 - speziell bei Quellcodelieferung inkl. Testcases
- **Schlechtere Softwarequalität führt zu mehr Kundenkontakten**
 - bessere Kundenbindung, Projektpotential
- **Neue Kunden sind schwer von Vorteilen zu überzeugen**
 - Qualitätsniveau ist (noch) nicht marktüblich



- **Ich bin "test-infiziert"**

- bereue jede eigene Entwicklung ohne Tests
- fordere konsequent von Studenten und Mitarbeitern automatisierte Tests
- kenne Chancen aber auch mögliche Risiken und Gegenmassnahmen

- **Testbasierter Entwicklung gehört die Zukunft**

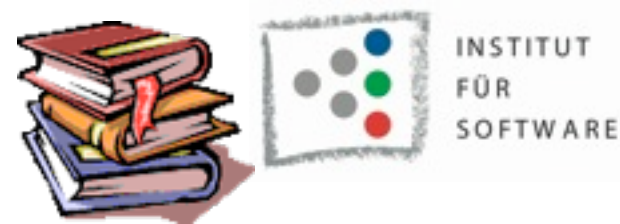
- Java-IDEs zeigen wie Integration aussehen soll
 - CUTE bietet das auch für C/C++ in Eclipse CDT
- Kernbestandteil unserer Ausbildung an der HSR
- gute autom. Applikationstest tw. noch schwierig

- **Lernen Sie, selbst testbasiert zu entwickeln**
 - unterstützen Sie Ihre Entwickler dabei
 - Praxis ist notwendig
 - Literatur unterstützt
- **Erlauben Sie ihren Entwicklern testbasierte Entwicklung zu erlernen**
 - beachten Sie die Lernkurve zum Erfahrungsaufbau, es braucht Zeit
 - Ausbildung, Übung, Literatur (HSR fragen!)
 - einfach Anfangen, nicht Abwarten
- **Achten Sie auf testbasierte Entwicklung bei Ihren Lieferanten!**



Fragen ?





- **Andy Hunt, Dave Thomas, Mike Clark: Pragmatic Starter Kit: <http://pragmaticprogrammer.com>**
 - Pragmatic Unit Testing with JUnit/NUnit
- **J.B. Rainsberger: JUnit Recipes**
- **Kent Beck: Test-driven Development by Example**
- **Dave Astels: Test-driven Development**
- **Lisa Crispin, Tip House: Testing Extreme Programming**
- **Johannes Link: Unit Tests mit Java**

- **CUTE - C++ Unit Testing Easier (mit Eclipse Plug-in)**
 - <http://r2.ifs.hsr.ch/cute> - Info und Bug Tracking
 - <http://ifs.hsr.ch/cute/updatesite> - Eclipse CDT plug-in