G64OOS (Spring 2014)

Lecture 08

Testing + Test-Driven Development

Peer-Olaf Siebers

The University of Nottingham
UNITED KINGDOM · CHINA · MALAYSIA

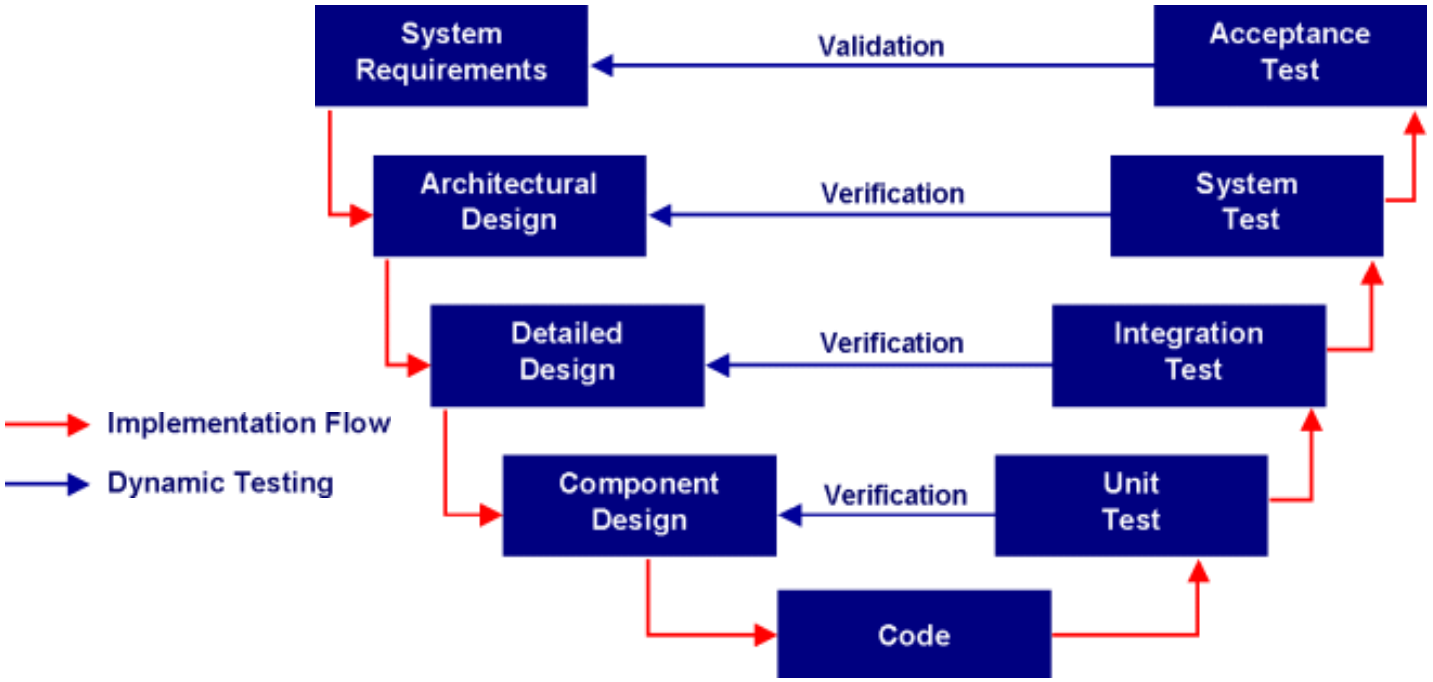
Motivation

1. Get an overview over the different types of software tests
2. Understand the relationship between Extreme Programming (XP) and Test-Driven Development (TDD)
3. Get an insight into what TDD is and how it works
4. Learn how to develop software by using a TDD approach and a Unit Testing framework

Test-Driven Development

- What is TDD

The V Software Life Cycle



For more on Software Engineering see Sommerville (2010)

Unit Testing

- Unit testing is a method by which individual units of source code are tested to determine if they are fit for use
  - The purpose is to verify the internal logic code by testing every possible branch within the function, also known as test coverage
- Static Unit Testing
  - Code is examined over all possible behaviours that might arise during run time; code of each unit is validated against requirements of the unit by reviewing the code
- Dynamic Unit Testing
  - A program unit is executed and its outcomes are observed
  - Programmer observes some representative program behaviour, and reach conclusion about the quality of the system

Integration Testing

- In integration testing the separate modules will be tested together to expose faults in the interfaces and in the interaction between integrated components
  - Testing is usually black box as the code is not directly checked for errors; this is done during unit testing

- The objective of integration testing is to build a "working" version of the system
  - Putting modules together in an incremental manner
  - Ensuring that the additional modules work as expected without disturbing the functionalities of the modules already put together

System Testing

- System testing will compare the system specifications against the actual system
  - Basic tests provide an evidence that the system can be installed, configured and be brought to an operational state
  - Functionality tests provide comprehensive testing over the full range of the requirements, within the capabilities of the system
  - Robustness tests determine how well the system recovers from various input errors and other failure situations
  - Inter-operability tests determine whether the system can inter-operate with other third party products
  - Performance tests measure the performance characteristics of the system, e.g., throughput and response time, under various conditions

System Testing

- Scalability tests determine the scaling limits of the system, in terms of user scaling, geographic scaling, and resource scaling
- Stress tests put a system under stress in order to determine the limitations of a system and, when it fails, to determine the manner in which the failure occurs
- Load and Stability tests provide evidence that the system remains stable for a long period of time under full load
- Reliability tests measure the ability of the system to keep operating for a long time without developing failures
- Regression tests determine that the system remains stable as it cycles through the integration of other subsystems and maintenance tasks
- Documentation tests ensure that the system's user guides are accurate and usable

Acceptance Testing

- Acceptance testing is the phase of testing used to determine whether a system satisfies the requirements specified in the requirements analysis phase
  - The acceptance test design is derived from the requirements document
  - The acceptance test phase is the phase used by the customer to determine whether to accept the system or not

Test-Driven Development

- What is TDD

Test-Driven Development (TDD)

- TDD is a software development process that relies on the repetition of a very short development cycle.
  - The developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards
- TDD is related to the test-first programming concepts of Extreme Programming (XP) but more recently has created more general interest in its own right.
- Programmers also apply the concept of TDD to improving and debugging legacy code developed with older techniques

For more on Test-Driven Development see Beck (2003)

Extreme Programming (XP)

- What is XP

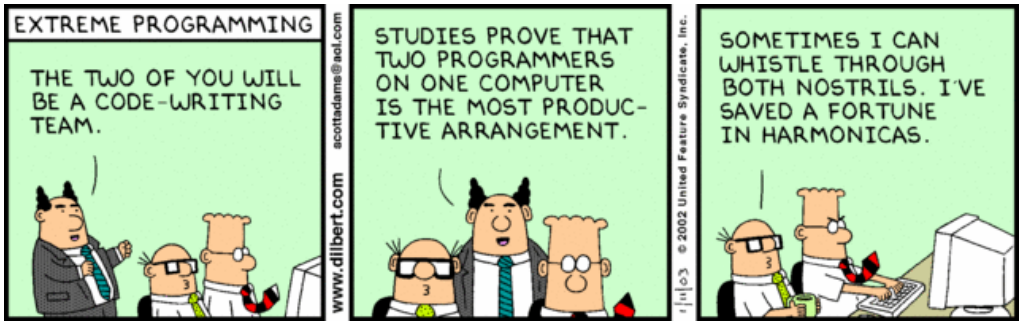
Extreme Programming (XP)

- XP is a software development methodology which is intended to improve software quality and responsiveness to changing customer requirements through frequent releases in short development cycles

- Other elements of XP include
  - Programming in pairs
  - Unit testing of all code
  - Avoiding programming of features until they are actually needed
  - Simplicity and clarity in code

Extreme Programming (XP)

- Difference between Scrum and XP?
  - Scrum focuses on managing the software projects
  - XP focus on the practices



Unit Testing (UT)

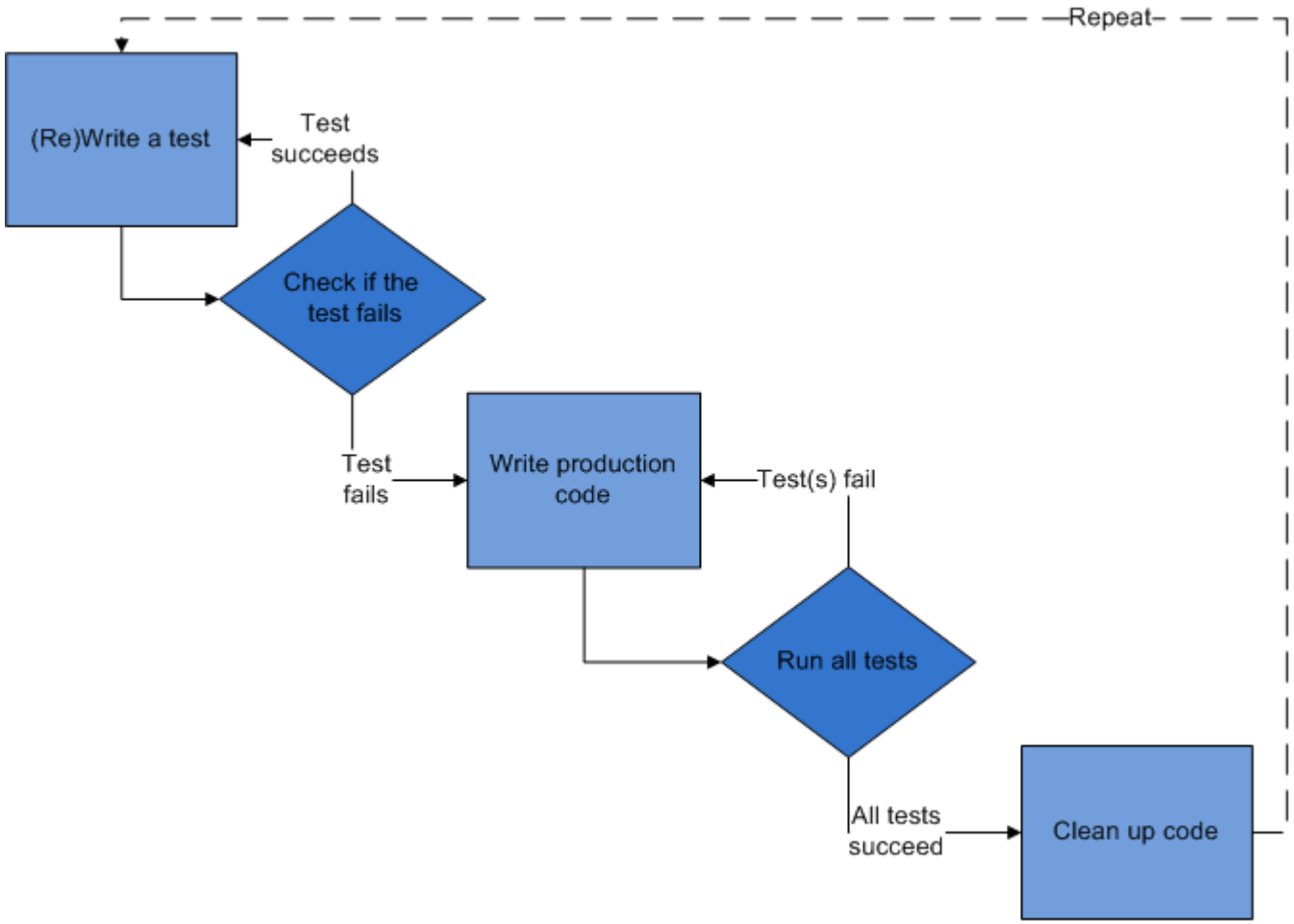
- Unit testing is a method by which individual units of source code are tested to determine if they are fit for use
  - A unit is the smallest testable part of an application

TDD vs. UT

- What is the difference between TDD and UT?
  - TDD refers to when you test; UT refers to what you test
  - TDD is a philosophical approach to writing code: Write the tests first. The tests you write are unit tests.
  - UT is about testing a code in small isolated units while TDD scales up to include integration and acceptance tests
  - In traditional UT you write the test after you wrote the code
  - Unit Testing is about testing a code in small, isolated units

Source: http://programmers.stackexchange.com/questions/59928/difference-between-unit-testing-and-test-driven-development

TDD Cycle


Repeat
(Re)Write a test
Test succeeds
Check if the test fails
Test fails
Write production code
Test(s) fail
Run all tests
All tests succeed
Clean up code

TDD Cycle

- Re(write) a test
  - Each new feature begins with writing a test; this test must inevitably fail as the feature being tested has not been implemented
    - If it does not fail: New feature already exists or the test is defective
  - To write a test, the developer must clearly understand the feature's specification and requirements
    - The developer can accomplish this through Use Cases and User Stories that cover the requirements and exception conditions
    - It makes the developer focus on the requirements before writing the code (differentiating feature of TDD versus writing UTs after the code is written)

TDD Cycle

- Check if test fails
  - Self test: It rules out the possibility that the new test will always pass
- Write production code
  - Write some code that will cause the test to pass
  - Code written at this stage might not be perfect
  - Code written is only designed to pass the test
    - Do not add additional functionality

TDD Cycle

- Run all tests
  - If all test cases now pass, the programmer can be confident that the code meets all the tested requirements.

- Clean up code
  - Now the code can be cleaned up as necessary
  - By re-running the test cases, the developer can be confident that code refactoring is not damaging any existing functionality

Benefits of TDD

- By focusing on writing only the code necessary to pass tests, designs can be cleaner and clearer than is often achieved by other methods
- When writing feature-first code, there is a tendency by developers and the development organisations to push the developer on to the next feature, neglecting testing entirely
- By focusing on the test cases first, one must imagine how the functionality will be used by clients; therefore programmer is concerned with the interface before the implementation
- Can lead to more modularised, flexible, and extensible code

Disadvantages of TDD

- TDD is difficult to use in situations where full functional tests are required to determine success or failure (e.g. user interfaces, programs working with databases, programs depending on network configurations)
- If developed by the same person the tests may share the same blind spots as the code
- High number of passing unit tests may bring a false sense of security (resulting in fewer additional software tests)
- The tests themselves become part of the maintenance overhead of a project

Break

- See you back in 10 minutes

C++ Unit Test Frameworks

- GoogleTest (http://code.google.com/p/googletest/)
- CppUTest (http://www.cpputest.org/)
- CPUnit (http://cpunit.sourceforge.net)
- Boost Test Library (http://www.boost.org/doc/libs/release/libs/test/doc/html/index.html)
- UnitTest++ (http://unittest-cpp.sourceforge.net/)
- ...

- Developing a C++ Unit Testing Framework (http://accu.org/index.php/journals/368)

Hands-On TDD Example

- Task: (http://wiki.codeblocks.org/index.php?title=UnitTesting)
  - Create a function that checks if a given year is a leap year or not

- Logic:
  - A year is a leap year when it can be divided by 4, unless it can be divided by 100. But in case it can be divided by 400, then again it is a leap year.

Hands-On TDD Example

Create an empty project (add UnitTest++ paths) and add files:

- LeapYear.h
- LeapYear.cpp
- LeapYearTest.cpp

{LeapYearTest.cpp}

```
#include "UnitTest++.h"

int main() {
    return UnitTest::RunAllTests();
}
```

Run all tests

```
Success: 0 tests passed.
Test time: 0.00 seconds.
```

Hands-On TDD Example

- How do we get started?
  - We need to think of a test that will help us in the design
  - We want a global leap year checking function which can be accessed from any class
  - How would it look like?
    - bool IsLeapYear(int Year);
  - Now we need a test that calls this function ...

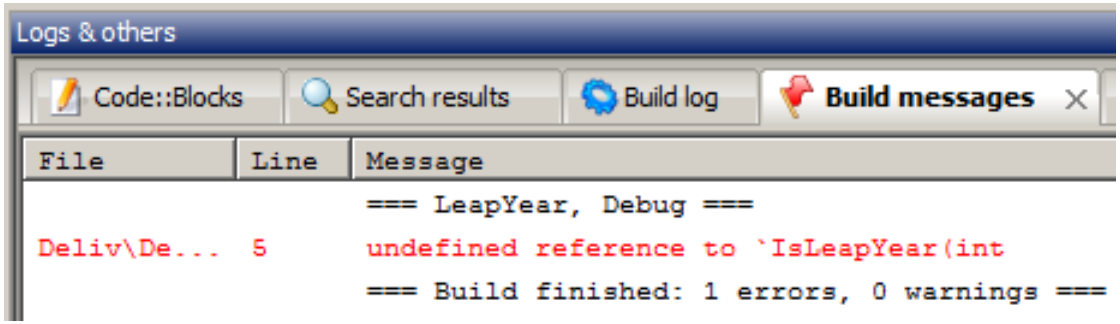
Hands-On TDD Example

{LeapYearTest.cpp}

```
#include "UnitTest++.h"
#include "LeapYear.h"

TEST(OurFirstTest){
    const bool Result=IsLeapYear(1972);
    CHECK_EQUAL(true, Result);
}
```

Disaster :(



Hands-On TDD Example

{LeapYear.h}

```
#ifndef LEAPYEAR_H_INCLUDED
#define LEAPYEAR_H_INCLUDED

bool IsLeapYear(int Year);

#endif // LEAPYEAR_H_INCLUDED
```

{LeapYear.cpp}

```
#include "LeapYear.h"

bool IsLeapYear(int Year){
    return true;
}
```

Success :) What next?

```
Success: 1 tests passed.
Test time: 0.00 seconds.
```

Hands-On TDD Example

{LeapYearTest.cpp}

```
#include "UnitTest++.h"
#include "LeapYear.h"

TEST(OurFirstTest){
    const bool Result=IsLeapYear(1972);
    CHECK_EQUAL(true, Result);
}

TEST(OurSecondTest){
    const bool Result=IsLeapYear(1973);
    CHECK_EQUAL(false, Result);
}
```

Disaster :(

```
L:\Workspace\TestPit\LeapYear\LeapYearTest.cpp:11: error: Failure in OurSecondTe
st: Expected 0 but was 1
FAILURE: 1 out of 2 tests failed (1 failures).
Test time: 0.01 seconds.
```

Hands-On TDD Example

{LeapYear.cpp}

```
#include "LeapYear.h"

bool IsDivisableBy4(int Year){
    return(Year%4)==0;
}

bool IsLeapYear(int Year){
    return IsDivisableBy4(Year);
}
```

Success :) What next?

```
Success: 2 tests passed.
Test time: 0.00 seconds.
```

Hands-On TDD Example

{LeapYearTest.cpp}

```
#include "UnitTest++.h"
#include "LeapYear.h"

TEST(OurFirstTest){
    const bool Result=IsLeapYear(1972);
    CHECK_EQUAL(true, Result);
}

TEST(OurSecondTest){
    const bool Result=IsLeapYear(1973);
    CHECK_EQUAL(false, Result);
}

TEST(OurThirdTest){
    const bool Result=IsLeapYear(1900);
    CHECK_EQUAL(false, Result);
}
```

Disaster :(

Hands-On TDD Example

{LeapYear.cpp}

```
#include "LeapYear.h"

bool IsDivisableBy100(int Year){
    return(Year%100)==0;
}

bool IsDivisableBy4(int Year){
    return(Year%4)==0;
}

bool IsLeapYear(int Year){
    return IsDivisableBy4(Year)&&!IsDivisableBy100(Year);
}
```

Success :) What next?

Hands-On TDD Example

{LeapYearTest.cpp}

```
#include "UnitTest++.h"
#include "LeapYear.h"

TEST(OurFirstTest){
    const bool Result=IsLeapYear(1972);
    CHECK_EQUAL(true, Result);
}

TEST(OurSecondTest){
    const bool Result=IsLeapYear(1973);
    CHECK_EQUAL(false, Result);
}

TEST(OurThirdTest){
    const bool Result=IsLeapYear(1900);
    CHECK_EQUAL(false, Result);
}

TEST(OurFourthTest){
    const bool Result=IsLeapYear(2000);
    CHECK_EQUAL(true, Result);
}
```

Disaster :(

Hands-On TDD Example

{LeapYear.cpp}

```
#include "LeapYear.h"

bool IsDivisableBy400(int Year){
    return(Year%400)==0;
}

bool IsDivisableBy100(int Year){
    return(Year%100)==0;
}

bool IsDivisableBy4(int Year){
    return(Year%4)==0;
}

bool IsLeapYear(int Year){
    return IsDivisableBy400(Year)||(IsDivisableBy4(Year)&&!IsDivisableBy100(Year));
}
```

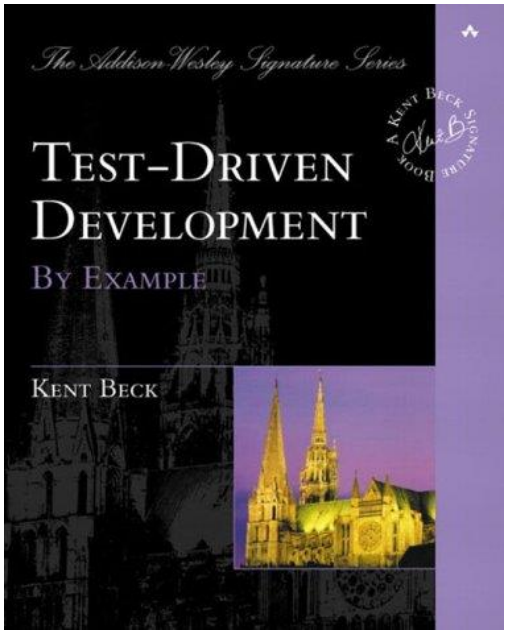
Success :) and ready for shipping

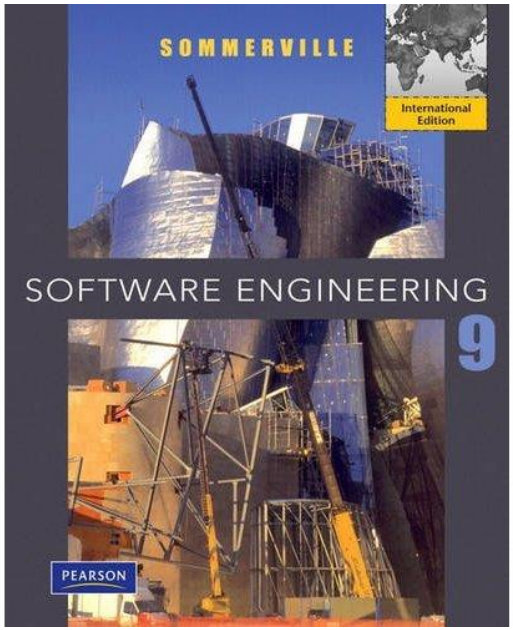
Real World Experience from an XP Expert

- Grazziela Figueredo

Resources

- Beck (2003) Test-Driven Development: By Example
- Sommerville (2010) Software Engineering (9th Ed.)





Summary

- What have you learned?

Questions / Comments