Johan Seland

Advanced Software Testing

Geilo Winter School 2013

Solution Example for the Bowling Game Kata

- Solution is in the **final** branch on Github

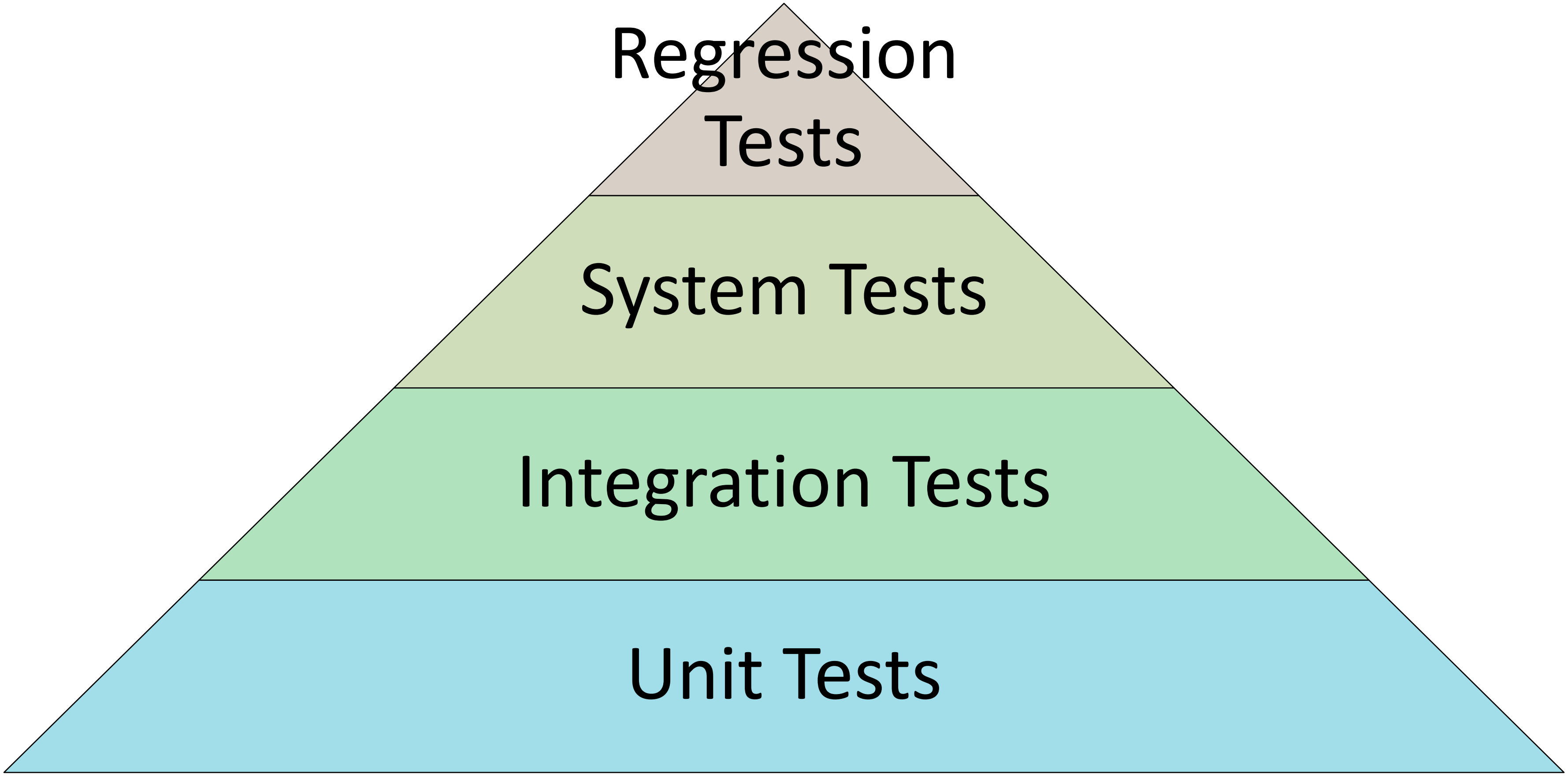
- ```
  git clone git://github.com/johanseland/BowlingGameKataPy.git

  git checkout origin/final
  ```
  ```

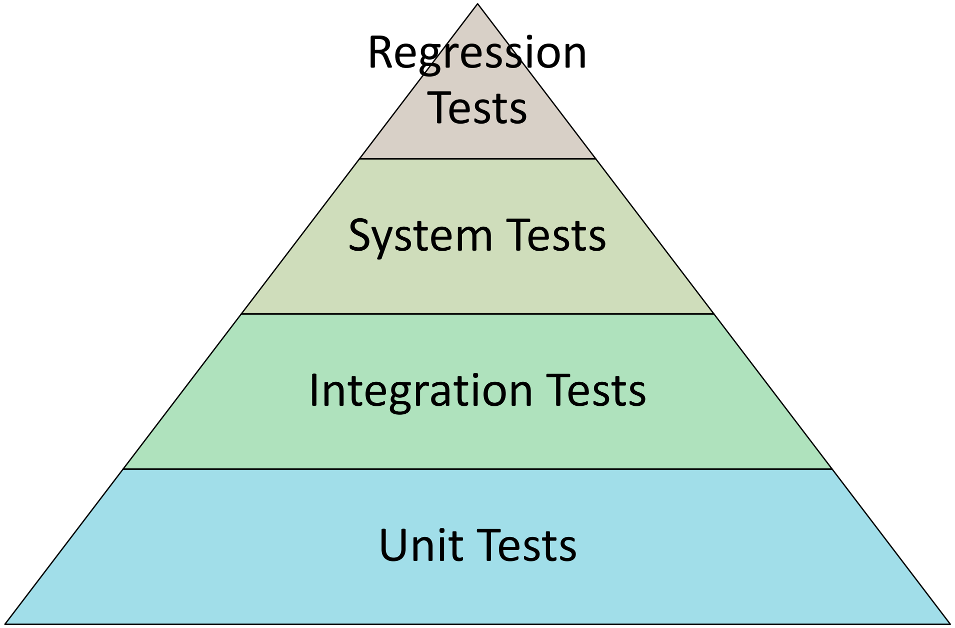
Overview of Lecture

- Testing Frameworks
- Writing good tests
- The mechanism of change
  - Refactoring
- Black-Box testing tools
- Continuous Integration

The Test Pyramid



The Test Pyramid



- **Unit Tests:**
  - Tests individual units of code
  - Function/Class level
- **Integration Tests:**
  - Tests how components communicate
  - Example: Is grid correctly initialized from init file?
- **System Tests:**
  - Are results valid?
  - Performance/Scaling/Resources/Stability
- **Regression Tests:**
  - Test for old bugs

xUnit Test Frameworks

- Testing software is made easier by test frameworks
  - At least for unit, integration and regression tests

- Typical facilities
  - Automatic test detection
  - Minimize boilerplate-code
  - Assertion functions
  - Test execution
    - Tests run in isolation
    - Command-line parsing
    - Standard output formats
  - Common terminology for talking about tests

xUnit Testing terminology

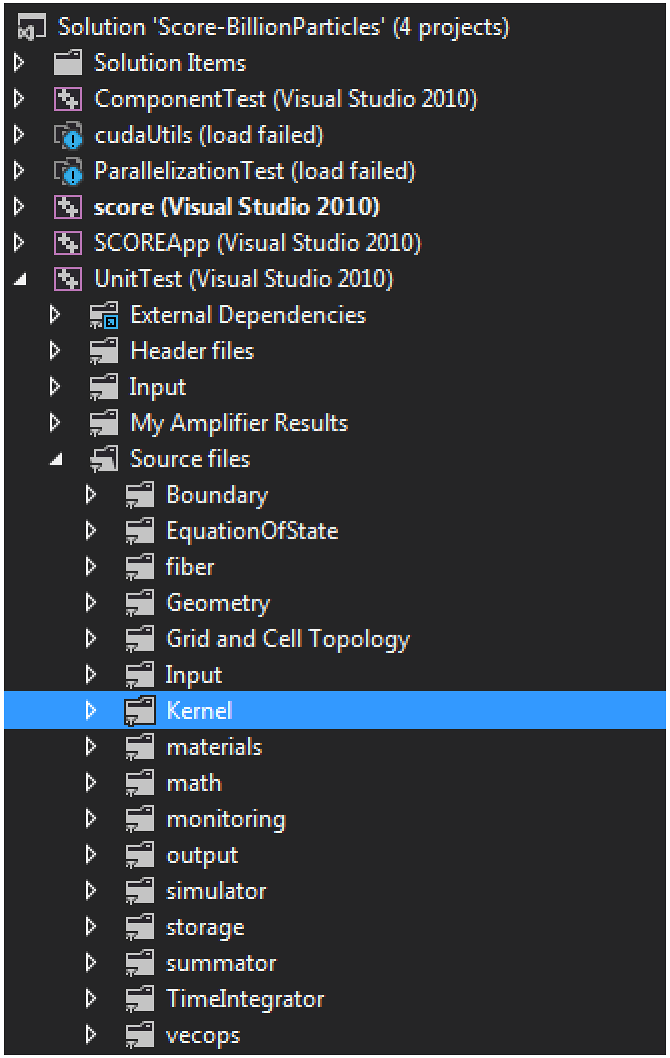
- Assertions
  - The actual tests
- Test case
  - Function invoking assertions
- Suites
  - A collection of cases
- Fixtures
  - (Data) structures set up before tests
- Mocks objects
  - Objects that mimic behavior of real objects

Every language has a xUnit-based Test Framework

- C++
  - GoogleTest, Qtest, Boost.Test
- Fortran:
  - pfUnit, fUnit
- .NET (C#, F#, etc.)
  - Visual Studio Testing Framework
- Java
  - Junit
- Python
  - UnitTest
- R
  - Runit
- Matlab
  - MUnit
- Open Source

Test code organization

- Split test and source files!
  - Consider building application as
    - Library + "App"-executable
    - Test-executables
- Separate project for different test layers
  - Unit vs Integration vs System vs Regression
- `make` should build everything
- `make check` should run as reasonable amount of tests
- Unambiguous answer if tests pass or fail



Testing best practices

- Test code is not a second-class citizen
  - Requires thought, design and care
  - It must be kept as clean as production code
- Agree on naming convention
  - TestFoo() or FooTest()?
- Test one “concept” per test
  - Often multiple tests per class/function
  - Kent Beck school: One assert per test

What to unit test?

- Be intelligent when writing unit tests!
  - Do not inflate their number

- Each test should have a meaning
  - Collapse/Cleanup tests when cleaning up the production code

- Typically test:
  - Zero-case
  - Trivial Case
  - Corner Cases (division by zero?)
  - Error handling

Clean tests – F.I.R.S.T.

- **Fast**
  - Tests should be fast.
  - You won’t run slow tests frequently
- **Independent**
  - Tests should not depend on each other
- **Repeatable**
  - They should be repeatable in any environment
  - Otherwise you have an excuse why they fail
- **Self-Validating**
  - Tests should have a boolean output
  - No manual evaluation should be needed
- **Timely**
  - Should be written *just before* the production code that make pass

Testing private methods

```
class Solver {
private:
    int iterationCount;
    double intermediate_result;
    void iterate();
public:
    double solve();
}
```

- **Short answer:**
  You should only test the public interface of a class
- **Reality:**
  The crucial computation happens in private methods
- **Possible fixes:**
  - Split into impl-namespace
  - Mark as protected instead and let test-class be subclass
  - Let test be a friend class (C++)
  - Mark as package-private (Java)

Testing Floating Point Computations

- **HERE BE DRAGONS!**
- Floating point in general is not associative
  - a op b != b op a
  - Beware of parallel computations
- Floating point is sensitive to compiler settings!
  - Fused operations
  - Compiler optimizations
  - Flush to memory
- Know your precision

Testing Floating Point cont'd

- For numerical algorithms, an estimate of the tolerance is needed
  - You can not simply test for equality
  - Absolute vs Relative error
  - Too low tolerance: Test might fail when implementation is correct
  - Too high tolerance: Test will not detect real errors

- Do you have (or can derive?) an error estimate/bound
  - Might be publishable itself!
  - Might just be asymptotic with unknown coefficient

Unit Tests for Iterative Methods

- Unit tests is not the place to test for stability of iterative methods
  - A black-box or system test
  - Will often require manual inspection of results
  - Goal of automated tests should be to decide if the implementation is correct

- **Unit** tests for iterative methods should test the **implemenation**
  - Constant input fields
  - Convergence criteria
  - Detecting invalid input

- Can you split it out so each substep has an analytical solution?

Strategies for Testing Floating Point

- Use analytic cases when available!

- Write tests to compare results between previous run of simulator
  - Typically require manual inspection
  - Often require dedicated post-processing tools
    - Bitwise reproducibility is not attainable

Mocks and fakes

- Often you can not rely on real objects for tests
  - Databases
  - Sockets
  - Huge datasets
  - Displays
  - Amazon Instances

- **FAKE** objects have working implementation with shortcuts
  - In-memory filesystem, constant grids

- **MOCK** objects are pre-programmed with expectations
  - Mock-libraries make it easier

Using Fakes: Non-testable code

- Use of InputFileReader is hardcoded in class (tight coupling)

```
class Simulator {
    Simulator() {
        ...
        grid = inputFileReader( filename );
    }

    double computeGradient();
};
```

- Problems:
  - I/O might take a long time
  - Input files must be distributed with tests

Using Fakes: Introduce Parameter Object

- We instead use an abstract class defining our behaviour

```
class AbstractInputFileReader {
    // Pure virtual method
    virtual grid readGrid( string filename ) = 0;
};

class ConcreteInputFileReader : public AbstractInputFileReader {
    grid readGrid( string filename ) {
        // fstream, fopen etc.
    }
};
```

- Parameter object is passed to simulator

```
class Simulator {
    Simulator( AbstractInputFileReader* inputFileReader ) {
        ...
        grid = inputFileReader->readGrid( filename );
    }

    double computeGradient();
}
```

Using Fakes: Introduce fake object

- In the test code, we add simplified version

```
class FakeInputFileReader : public AbstractInputFileReader {
    grid readGrid( string filename ) {
        // Create grid with constant value etc.
    }
}
```

- And pass this to simulator

```
TEST( Simulator, test_computeGradient ) {
    AbstractInputFileReader* reader = new FakeInputFileReader();
    Simulator sim( reader );
    ASSERT_EQUAL( 0.0, sim.computeGradient() );
}
```

Fake objects conclusion

Testable code will often have:

- More classes
  - Should not be a problem in modern IDEs
  - Use ECB in Emacs
- Smaller classes
  - That follow the *Single Responsibility Principle (SRP)*
- Looser coupling
  - One of the benefits of object-oriented languages

Changing software

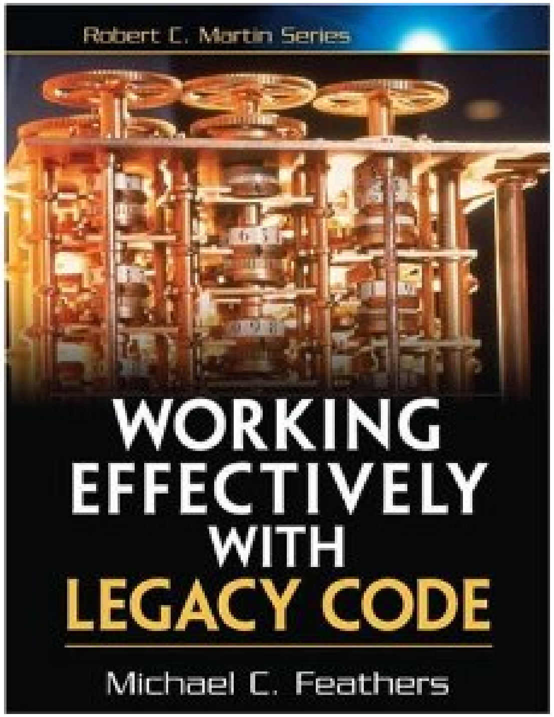
- Why do you want change?

1. To add a feature
2. To fix a bug
3. To improve the design (refactoring)
4. To optimize

The mechanisms of change cont'd

	Add a Feature	Fix a bug	Refactoring	Optimizing
Structure	Changes	Changes	Changes	Changes?
New Functionality	Changes	-	-	-
Functionality	-	Changes	-	-
Resource Usage	-	-	-	Change

What about legacy code?



- *The main thing that distinguishes legacy code from non-legacy code is tests, or rather a lack of tests."*
  Michael Feathers

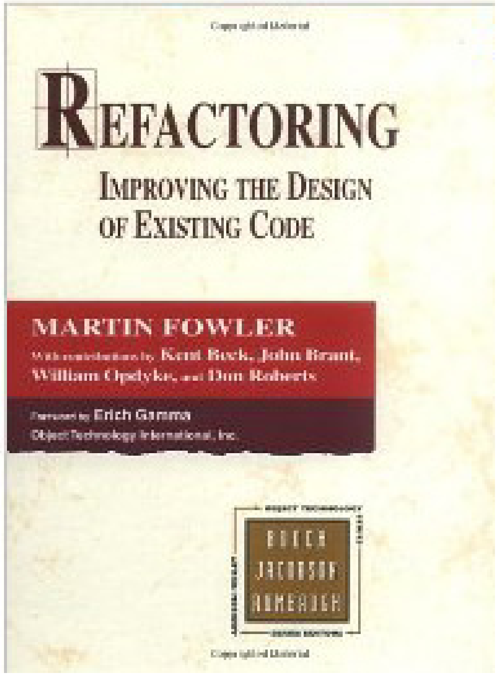
- Alternative definition:
  Code you are afraid to change, cause you can not see the consequences

BRING IT UNDER TEST!

Refactoring

Code refactoring is the process of changing a computer program's source code without modifying its external functional behavior in order to improve some of the nonfunctional attributes of the software

- Make it work
- Make it right
  - Maintainability
  - Extensibility



Some refactoring techniques

- Rename field
  - Change a name into a new one that better reveals its purpose
- Extract method
  - Turn part of a larger method into a new method
- Move field
  - Move to a more appropriate class or source file
- Extract class
  - Move part of the code from an existing class into a new class
- Generalize Type
  - Create more general types to allow for more code sharing
- Many more at refactoring.com

Refactoring in practice

- IDEs have some support for automatic refactoring
  - Guarantees that behavior does not break
  - Java IDE have
- C++ is probably the most difficult language ☹
  - Templates are specially difficult
  - But we are getting there as well (QTCreator in particular)
- Lean on the compiler
  1. Alter declarations to cause compile errors
  2. Navigate to errors and make changes
  3. Rerun tests!

- Pair programming!

Other testing tools

- A plethora of tools to analyse your program and tell you something about is status
  - At the source-code level or program level
  - Commercial or open-source

- White-box testing is biased
  - You generally write them yourself

- Testing tools does not lie

Static Code Analysers

- Programs that look at your source code
- Identifies common errors
  - Bounds checking for arrays
  - Memory leaks
  - Resource leaks
  - Stylistic errors
  - Code duplication

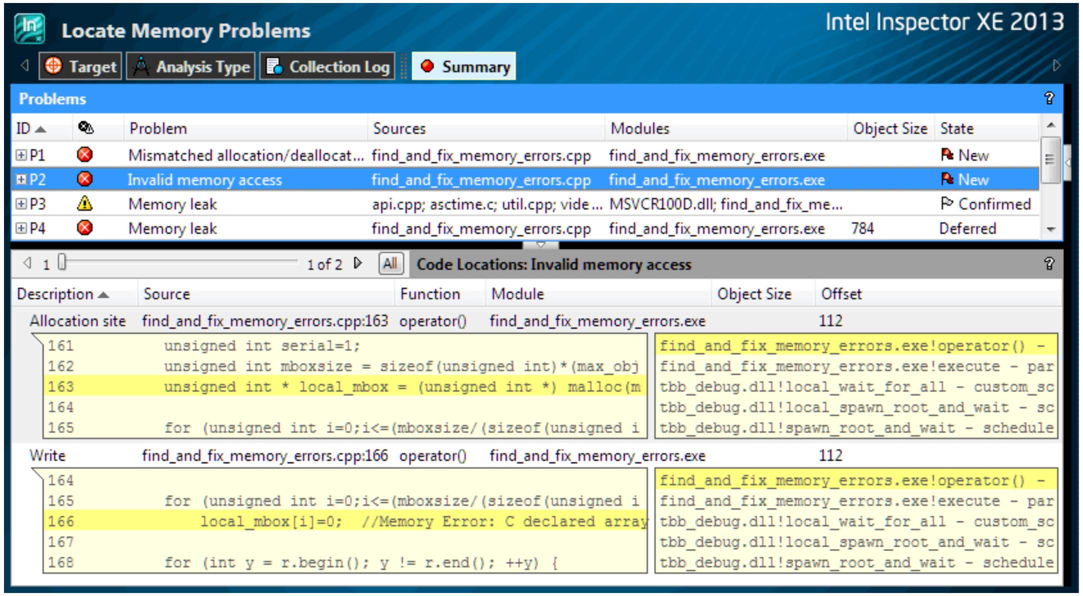
- Compute code metrics

- Subject to false positives and negatives

- Common tools: Cppcheck, Cpplint

Profilers

- Tools that monitor the execution of a program
- Allows you to understand their behaviour when running on real code



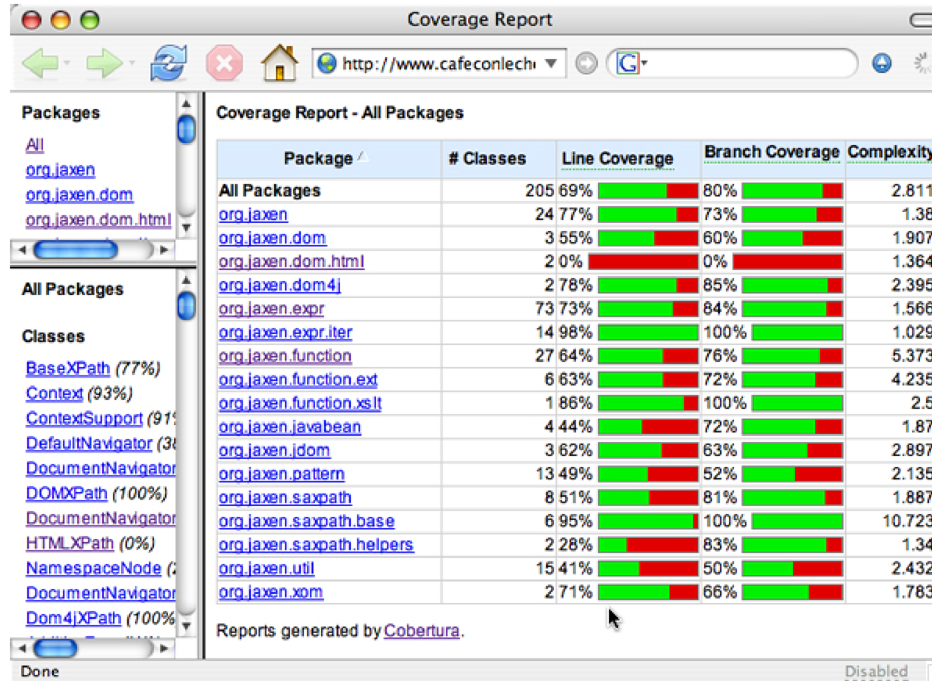
- Used to detect:
  - Performance metrics
  - Memory leaks
  - Parallelization errors

- Program must be *instrumented*
  - Compiler switch
  - Run in a virtual environment
  - Often program execution is slowed down dramatically
    - Yet another benefit of many separate test programss

- Example Tools: Valgrind, gcov, Intel Parallel Studio

Code coverage

- Measure which code is executed at all



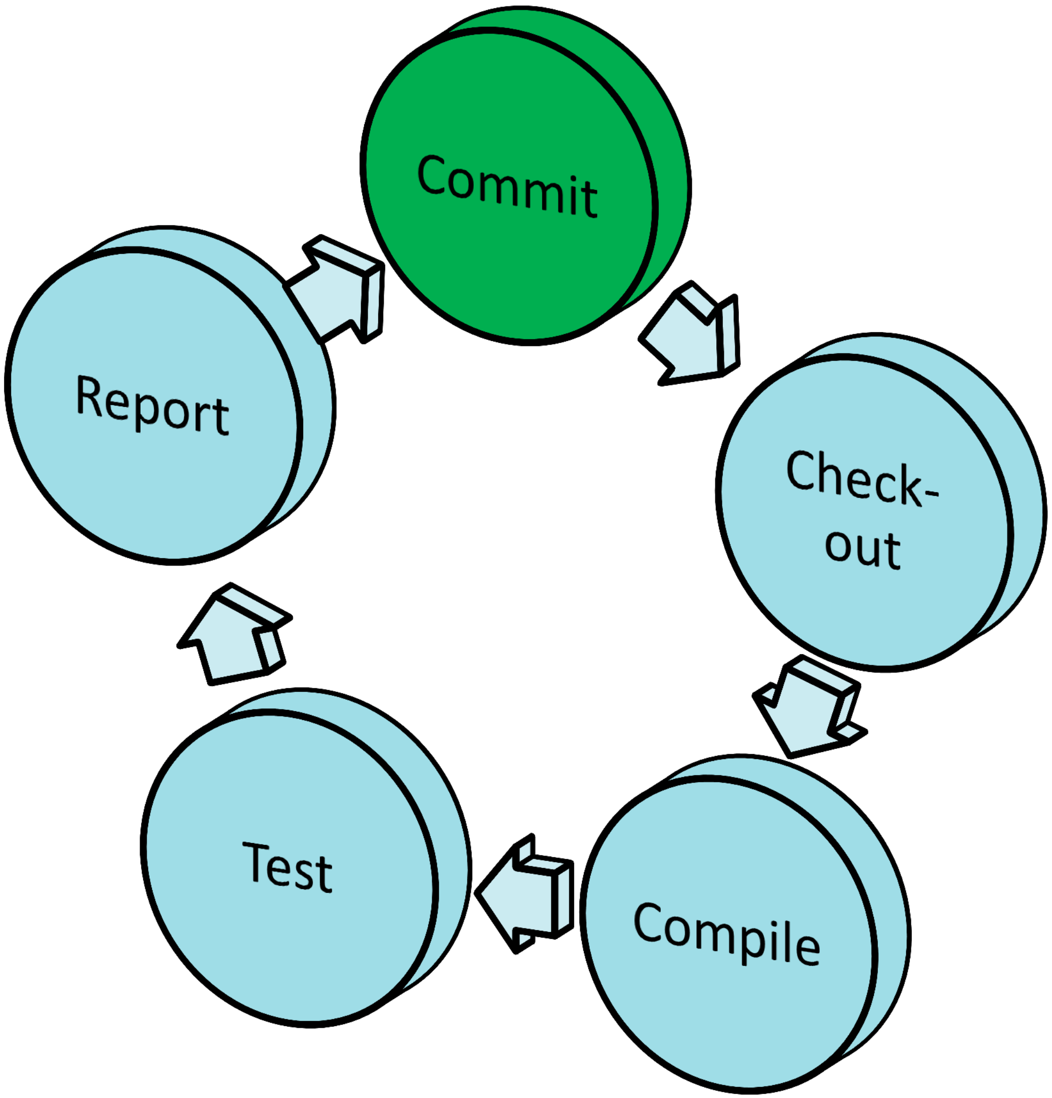
- Used to detect if you have code that is not covered your tests
  - Are you testing each direction of an if-statement?
  - Is it code you are not executing

- Example programs: gcov (GCC stack), Cobertura (Java), coverage (Python)

CONTINOUS INTEGRATION

Motivation

- If daily builds are good
    - Continuous builds are better
- If daily testing is good
    - Continuous testing is better
- Detect issues as early as possible
- Jenkins is a web-service that happily builds code and executes test all-day
    - Jenkins will build on all your platforms
    - Execute long-running tests
    - Syndicate results across builds


Commit
Check-out
Compile
Test
Report

Benefits

- Instant feedback
  - Everyone can see status
- Latest executable available
- Build on all platforms all the time

Downside:

- Should really be run on dedicated server

Jenkins best practices

- Email only when build breaks/tests start to fail
  - Per project participant list
  - If people start filtering emails you have lost

- Everyone can look at build configurations
  - Avoids mysterious cron jobs on private workstations
  - Jenkins is not a high-security system

- Do not build on your own workstation
  - Highlights new dependencies

- Use clean builds

Jenkins slaves

You need something not on Jenkins server

- Matlab
- Windows
- GPUs
- Fluent

- Hudson can start jobs on slaves
  - Extremely easy to set up

Jenkins tips

- Performance monitoring
  - Runtime for test is in XML-report
  - Small Python script to extract it
- Correctness monitoring
  - Compare output to prev. output

- Get cron jobs in there as well

- Back up Hudson!

More possibilities

- Store profiling information
- Validate single-thread vs parallel implementations
- Validate against gold standard
- Analyze compiler warnings
- Static code analyzer
- Check for memory leaks (Valgrind)
- Let managers/supervisor know about available metrics?

Jenkins at SINTEF Applied Mathematics

- Server was set up in summer 2010
- Specialized servers added later
  - GPU build server
  - Windows build server
- Informal tutorial session
- Quickly adopted for many projects
  - People continue to use it!

Thing not covered

- Social Processes
  - Code-Reviews
  - Pair-Programming
- Bug/Issue tracking
- Acceptance Testing
  - Fitnesse Framework
- Various code metrics

Concluding remarks

- You need a testing strategy
- Your testing strategy should consist of a battery of
  - White-box testing at the source level
  - Black-box testing at the program level
- Automate as much as possible
  - Minimize the amount of human parsing necessary
- Execute tests as often as possible
  - Continuous Integration is an enabling technology for this

Reading list