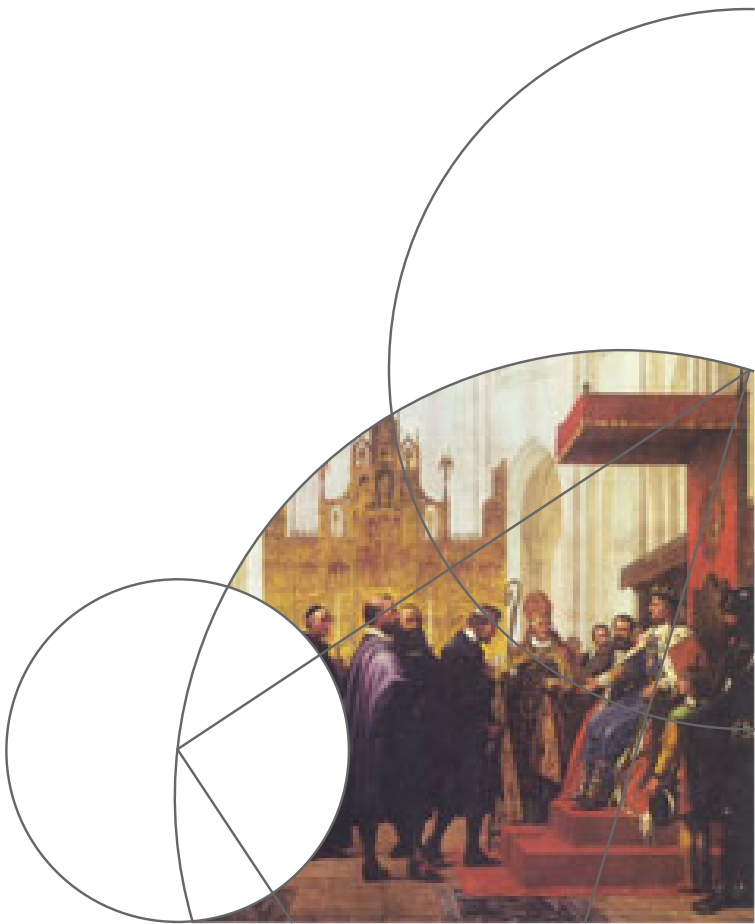
Unit Testing

Frederik M. Madsen
fmma@diku.dk



Software correctness

- Correctness w.r.t. specification.
- Formal proof.
  - Mechanized program verification.
- Hoare logic.
  - Pre- and post-conditions.
- Software testing:
  - Functional testing / integration testing.
  - *Unit testing*.
  - White-box and black-box.
  - Smoke test.

Unit testing

- Break down your program into *units*.
  - Isolate test cases $\sim$ isolate bugs.
  - Granularity:
    - Code blocks, functions, classes, packages.
      White-box $\longrightarrow$ Black-box
- Partial testing:
  - Almost never feasible to test entire input domain.
    1. Cover only "interesting" cases.
    2. Randomly generate test case and check invariants.
- Manual (by hand) or automatic (by code).
- Integrated part of build system:
  - Quickly verify implementation changes (optimizations). ☺
  - Large-scale refactoring requires change in tests. ☹

Unit testing

- Break down your program into *units*.
  - Isolate test cases $\sim$ isolate bugs.
  - Granularity:
    - Code blocks, functions, classes, packages
  - Integrated part of build system.
    - Quickly verify implementation changes (optimizations). ☺
    - Large-scale refactoring requires change in tests. ☹

Discuss:
The choice of unit granularity is important. Briefly discuss the following:

1. Pros and cons of very fine-grained units.
2. Pros and cons of very course-grained units.
3. What are the properties of a good choice of granularity?

Answers:

1. ☺ Bug isolation.
   ☺ Simple tests (easy to write).
   ☹ Many units, many tests.
   ☹ Simple tests (too trivial).
   ☹ No integration testing.
2. ☹ Little bug isolation.
   ☹ Complicated tests (hard to cover all interesting cases).
   ☺ Few units, few tests.
   ☺ Complicated tests (catches many bugs).
   ☺ Integration testing within units.
3. A good choice of granularity:
   - When a test fails, it should be relatively easy to locate bug.
   - A good tradeoff between the amount of interesting test cases in each test and the number of tests you have to write.
   - Tests should be complicated enough that they can actually find bugs, but not so complicated that bugs might slip through unnoticed.

Test-driven development

- Write unit tests *before* implementation.
- Unit tests $\sim$ (partial) formalization of specification.
- Works sometimes:
  1. Think about interface.
  2. Write tests.
  3. Implement $\rightarrow$ gain experience (interface sucks).
  4. Goto 1.

  - Pre-condition: Good, a priori interface design.

Brief detour: Reflection and annotations

- *Reflection*
  - Observe and manipulate runtime behavior at runtime.
  - `java.lang.reflect`.
    - *More in lecture 4*.
  - Example: Run all methods with name prefix `test`.
    - Automatic test harness.
    - Prone to error:
      `tetsFoo()` never runs (fails silently). ☹
      `testosterone()` runs. ☹
- *Annotations*
  - Structured meta information, verified by compiler.
  - Builtins: `@Override`, `@SupressWarnings`, `@Deprecated`
  - Make your own, define where it can be used, define fields.
  - Annotations visible through reflection.
    - Example: Run all methods with `@Test` annotation.

JUnit 4

- De facto unit testing for Java.
- Unit tests of Java programs written in Java.
- Unit granularity: Usually methods.
- Based on annotations and reflection:

```
@Test
public void testFoo() {
   Bar bar = new Bar();
   assertEquals(5, bar.foo(10,2));
}
```

- Eclipse integration:
  - Automatic test code generation.
  - Automatic build integration.
  - Reports.

Demonstration using JUnit4 in Eclipse.

Advanced stuff

- Testing private methods - 3 choices:
  1. Put test method inside tested class.
  2. Change private to protected and inherit tested class.
  3. Use the reflection API.
- *Parametrized* tests and *theories*.
  - More fun with annotations and reflection.
  - Assume we have:
    1. Data set D of data points.
    2. Collection of tests T taking a data point as input.
  - Automatically generate all combination of tests $T \times D$.
  - Seperation of test code and data points.
    - Easy to add new data points.

JUnit and concurrency

- Input-output correctness: Nothing new.
- **But**, software correctness usually includes *thread safety*.
- You have to think about possible interleavings of the executing threads:
  1. Cover only "interesting" cases.
     - All interleavings are potentially "interresting" test cases
       $\Longrightarrow$ Your test cases, almost certainly, covers only a tiny fragment of test domain. ☹
  2. Randomly generate test case and check invariants.
     - Randomly generate interleavings = combinatorial explosion
       $\Longrightarrow$ You have to generate a galactical number test cases to gain confidence in your test. ☹
- Enforcing concrete interleaving in Java requires lots of work.
  - Lots of test code.
  - Is the test code correct?

Program correctness and concurrency

- Conclusion: Do not trust multi-threaded unit tests.
  - False sense of security.
- Consideration: An operation using multiple threads, should probably not be considered a unit.
- The right approach:
  1. Minimize surface area between threads (minimize possible interleavings).
     - Minimize shared data.
     - Make shared data immutable when possible.
     - Use locking mechanisms.
  2. Use already verified patterns / data structures.
     - `java.util.concurrent`.