# Testing Tools



## Testing in General

Motivation and Alternatives

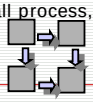


## Correctness of Software

- Sad but true: (hand-written) code is bug-ridden
- Typical statistics:
  - error rate

| software | errors / 1000 LoC |
|---|---|
| "normal" software | 25 |
| "important" software | 2 to 3 |
| medical software | 0.2 |
| Space Shuttle | less than 0.1 |

  - Space Shuttle software:
    - 3 millions LoC with 300 errors
    - 3,000 millions cost amounts to $1000 per LoC
    - 15,000 man years
- Cost to fix errors may be higher than original development cost
  - see rationale on slide 2-19, waterfall process, Boehm's spiral model





## Software Testing

- Goal: find "many" errors before shipping software to customers
  - cost of fixing errors after deployment
  - acceptance / confidence of users
- Approach: try out software on in typical usage scenarios
  - scenarios can be derived from use cases
  - problem: supply typical input data



## Limitations

- Purpose of testing:
  - By testing one can find errors in code.
  - **The passing of tests does not guarantee the absence of errors**
    - the erroneous code was not covered by a test
    - the input data may have been "fortunate"
    - the error lies in missing fault tolerance
- Scope of testing:
  - only functional requirements are checked
  - non-functional requirements are not covered by test cases; here, profiling is needed



## Alternative: Software by Construction

- Ideal: software correct by construction
  - formally specify software
  - create software by a "constructive proof" of specification
  - Model Driven Architecture
- Approaches:
  - Efforts in the specification of program semantics [Floyd], [Hoare], [Dijkstra], …
  - Program specification languages: VDM, Z ([2])
- Obstacles:
  - So far, "programming by specification" does not seem to be a feasible approach in real-world software development scenarios.
  - Only functional requirements can be formulated.

## Continuous Testing (1)

- Code is changed constantly, e.g. for:
  - fixing errors
  - adding new functionality
- Unified Process: iterative and incremental software construction
- Agile methods (e.g., Extreme Programming [3]): code is changed constantly as part of the methodology



## Continuous Testing (2)

- Today's software development practice especially relies on continuous testing:
  - team development
  - complex design
    - changing a class can easily break code in other classes; e.g., redefining a method affects the caller
    - relationships between classes are not always obvious; e.g., dynamic dispatch in frameworks
  - component integration
    - inclusion of components not developed in-house
    - complex, at times subtle interaction of components



## Automated Testing

- Testing:
  - test cases defined by
    1. code to be checked
    2. code to run tests (simulating operation)
    3. input data to be used
    4. output data expected
- (Semi-) Automatic testing:
  - have this procedure executed automatically at certain points during development
  - generate reports on test results



## Test-driven Development

- Automated testing during development: test-driven development (TDD)
- Testing becomes part of development process
- Test case formulation becomes part of the programming task
- Positive side effects:
  - partial specification of program semantics
  - test cases contribute to the documentation of software artifacts



## Unit Testing

Making sure the system does what it is supposed to do.



## Unit Testing

- Small test cases are run against pieces of the software (hence the name unit test)
- OO again turns out to be a well-chosen paradigm:
  - plenty of units
- Large number of tests can be run automatically
- Goals of unit testing:
  - model requirements in unit tests
  - use these to ensure that:
    - new features are implemented correctly
    - old features continue to work as expected

Tests individual modules as well as classes of the application.

## Integration Testing

- Larger test cases are run on higher level
- OO-structures: packages
- Goals are the same as for unit tests
- Similar to “Acceptance Testing” in EXtreme Programming

Tests whether the modules of the application work together.



## How to Use Test-driven Development

- Implement tests as executable code
- To implement a class, you...
  1. figure out what functionality it will have
  2. map this functionality into attributes and methods
  3. write a test case that tests at least the public methods
  4. delvelop the class
  5. test the class, if a test fails, go back to 4.
- Tests serve a double purpose:
  - automatic verification
  - documentation of how to use the class



## Creating Tests

- Your tests need to cover all the “important” cases, while minimizing the total number of tests
- Test should:
  - Be sure to include *normal* as well as
  - *boundary cases* in the tests
- A test can work on an individual class (unit tests)
  - this can be achieved most of the time
  - sometimes difficult to test all functionality this way
- Tests can also work on whole parts of the system (integration tests)
  - these tests often use customer supplied test data as input and output



## Unit Testing Details

- Tests are useful for quality-control, test can...
  - *work* correctly (the class behaves as desired)
  - *fail* (result not as expected)
  - cause an *error* (test did not complete properly)

Note that there is a difference between failure and error!



## JUnit

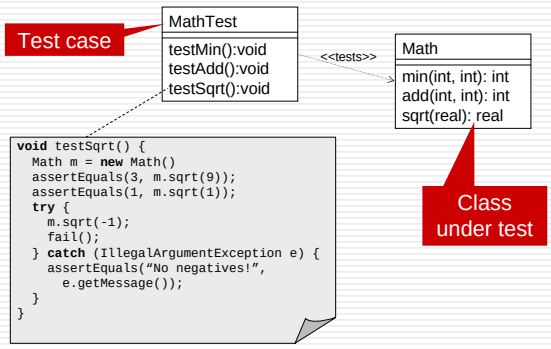
- JUnit is a unit testing framework for the Java platform.
  - most of what is said here applies to unit testing in general
  - many testing frameworks for Java are based on JUnit
- Test are organized hierarchically, usually along the package structure of the application
  - test cases can be aggregated into test suites
  - one suite per package of the application
  - one case per class of the application
  - one method in the test case per method in the class



## JUnit (2)

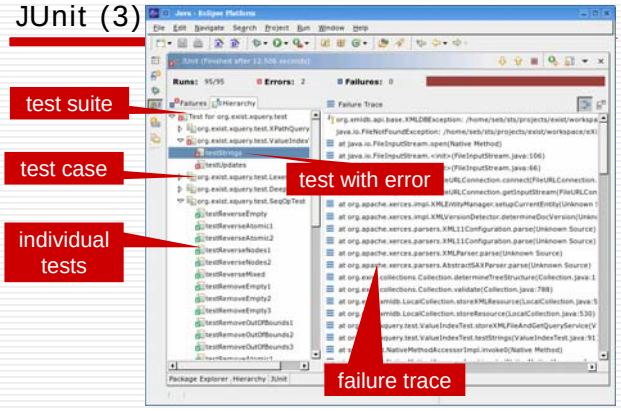
- Whole family of unit testing frameworks: xUnit
- About 30 ports to various languages
  - started with Python
  - JUnit is the Java port

## Example: Math Class






## JUnit (3)






## Qulality Control

- Automation allows for easy, yet complete testing
- In team development, you normally run tests before submitting a change to the repository
- Tests to run before checkin are called a *regression suite*

New developer workflow:

1. Get an initial version from the repository.
2. Write code (create changes).
3. ***Run regression suite***, if fails: fix; else:
4. Upload the changes into the repository.
5. Get the changes from the repository.



## Implementing Unit Tests

- Each test will test a number of *assertions*
  - implemented in **`assertXXX`** methods in JUnit
  - "Is the actual value that is returned by the method under test the one that was expected?"

test that should work and return the value expected

if execution ever comes here, the test will fail

make sure we get an error and the message is right.

```
void testSqrt() {
  Math m = new Math()
  assertEquals(3, m.sqrt(9));
  assertEquals(1, m.sqrt(1));
  try {
    m.sqrt(-1);
    fail();
  } catch (IllegalArgumentException e) {
    assertEquals("No negatives!",
      e.getMessage());
  }
}
```



## Implementing Unit Tests (2)

- Each test case extends junit.framework.TestCase
- TestCases can be bundled into suits
- JUnit provides methods for initialization and clean-up:
  - setUp() is called before each testing method
  - tearDown() is called after each method



## Common JUnit Methods

| | |
|---|---|
| assertEquals(expected, actual) | compares expected value to actual value. Test will fail if they differ |
| assertFalse(expression), assertTrue(expression) | evaluate a boolean expression. |
| assertNull(var), assertNotNull(var) | test whether var is null |
| fail() | causes test to fail. Commonly used with exception testing, see example |

see [1] for details

## Coverage: Path-Completeness

- How do you make sure, the code is properly tested?
- Path-completeness
  - every branch of code is covered by a test
  - beware that path-completeness does not guarantee complete tests!

```
int median(int x, int y, int z) {
  return z;
}
```

```
void testMedian() {
  ...
  assertEquals(2, o.median(4, 1, 2));
}
```



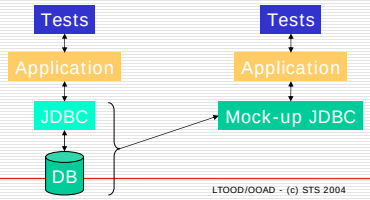
## Coverage: Data Partitioning

- It is unfeasible to test all possible combinations of input data,
- therefore you just test the characteristic cases.
- There is not general recipe how to do this, it takes experience and a close look at the method under test
- Generally, you will want to include:
  - some “normal” cases
  - some fence-post ones - i.e. the bordering cases that are at the ends of the domain of allowed values
  - some cases that are outside to test for proper handling of errors



## Mock-Ups

- Simulate parts of a system if you don’t
  - have that part (yet)
  - want to be dependend on a specific implementation
  - want bugs in the sub-part to cause your test to fail
- Mock-ups are usually not functional
  - just provide enough functionality to run the test
  - “advanced” functionality (e.g., multi-user) not implemented






## Debugging

Finding the cause of a failure



## Need for Debugging

- Once an error has been discovered: what then?
- Tests usually only unveil the presence of an error
  - This is the *symptom*.
- Example:
  A precondition is violated because of an unexpected null value in the database.
  - Where has that value been created?
  - (i.e., where is the *cause* of this error?)
- Approach: “debugging”
  - inspection of the states a software at runtime
  - finding the statement by which an erroneous state is entered

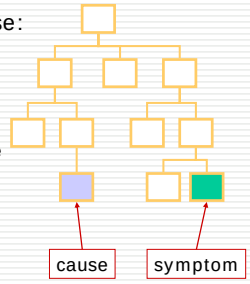


## Debugging Tools

- Debugging tools are common development tools
- The current form of debuggers used to be called “symbolic debuggers”, since debugging is done one the source code level
  - inspect variables, method invocations, etc
  - instead of memory dumps, and call stacks
  - … but you still have to know what a variable means
- Debuggers are usually integrated with IDEs.

## Problem: Symptom and Cause

- Debugging is difficult because:
  - Symptom and cause are often seperated
  - Symptom may disappear with fixture of different bug
  - Symptom might be outside the scope of your system (e.g., library bug)






## Debugging Concepts

- Typical debugging concepts are
  - breakpoints
  - step execution
  - watches
- Moreover, modern debugging environments have features like
  - changing bindings of variables
  - hot code replacement: changing code during debugging process



## Breakpoints

- A breakpoint is a point in the (source code of a) program where execution should stop
- It allows inspection of the program's state
- The developer can continue execution
- Many debuggers give you the option of "conditional breakpoints"
  - developer can specify a condition
  - execution stops only if condition is true
- Exception-based languages also offer trapping of exception



## Step Execution

- Execute one line at a time.
- Comes in different flavours:
  - You can step into method calls:
    - "Step-into" vs.
    - "Step-over"
  - You can run to the end of the current method
    - "Step-out"



## Watches

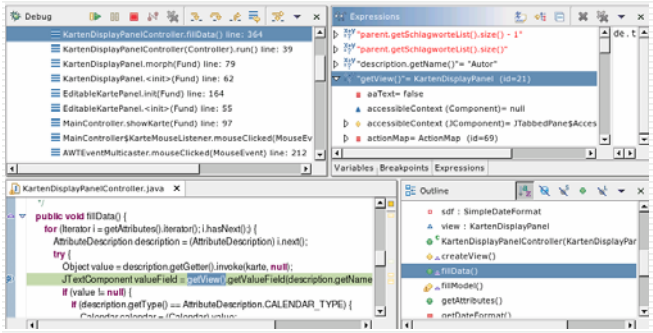
- Most debuggers automatically display all varibales in the current context
- Developers can configure an additional list of expression to show, these are called *watches*



## Hot Code Replacement

- Changes to the application's code are injected into the running system.
- Useful to:
  - test fixes
  - work on systems that take very long to start
- Support is often limited to method bodies:
  - Currently you cannot change signatures in Java
  - If you could, what would this mean in concurrent systems?

## Screenshot






## Logging

Understanding complex inner workings



## Motivation

- What is logging?
  - Writing messages during runtime (to console, a file, an e-mail address, ...)
- Isn't this the poor-man's approach to debugging?
  - No.
  - Complex systems are hard to debug.
  - Well-placed log messages are usually much easier to understand.



## Naive Approach

- Just dump messages to the console:

```
x = algorithm.calculateResult();
System.out.println("x = " + String.valueOf(x));
```

- Disadvantages:
  - production system will still dump messages
  - too many messages if used a lot
  - cannot be turned off
  - cannot be directed to anything but the console
  - might interfere with proper messages
- Not a good idea in general



## Logging Libraries

- Supply sophisticated means for logging
- Log4J is one for Java. Logging work like this:

```
class MyClass {
  private Logger LOG = Logger.getLogger(MyClass.class);
  public void method1() {
    ...
    x = algorithm.calculateResult();
    LOG.debug("x = " + String.valueOf(x));
  }
}
```

level of message: debug



## Log4J

- Can be configured without changing the code
  - to log to different places
  - to log only messages of a certain level (debug, info, warn, error, ...)
  - to have a different debug level for certain classes
  - almost any format of log message
  - many "appenders" (e.g. rolling file, e-mail, visual, text file, XML file, ...)

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <appender name="console" class="org.apache.log4j.ConsoleAppender">
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern" value="%d{DATE} [%t] %-5p (%F [%M]:%L) - %m\n"/>
    </layout>
  </appender>
...
```

## Example File

```
...
  <category name="de.tuhh.gkns.client" additivity="false">
    <priority value="debug"/>
    <appender-ref ref="console"/>
  </category>
  <category name="org.exist.xmldb" additivity="false">
    <priority value="warn"/>
    <appender-ref ref="console"/>
  </category>
  <root>
    <priority value="error"/>
    <appender-ref ref="console"/>
  </root>
</log4j:configuration>
```



## Logging

- Logging code is usually left in the production system
  - can turn on logging for certain components (single classes) at customer
  - minimal intrusion
  - much easier to reproduce bugs with suitable log
- Sometimes also important for legal reasons
  - e.g. web site logs



## Profiling

Where are we wasting all that time?



## Profilers

**Profilers** help finding codesections of your program that consume an inappropriate amount of processing time. These codesections represent *bottlenecks* in your program.

- Processing time consumption can be due to
  - The use of an inappropriate algorithm (bubble sort)
  - The unnecessary creation of large numbers of objects
  - The unnecessary synchronization of threads or
  - The repeated invocation of an operation
- Bottleneck are typically not where you expect it:
  - your special algorithms are usually highly optimized ...
  - ... while somewhere else in your program String.toLowercase() is invoked unnecessarily a few 100.000 times



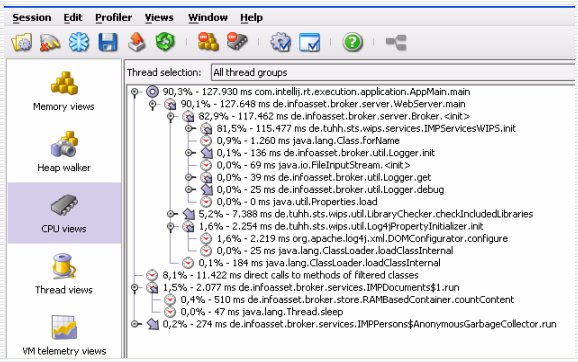
## Profilers (2)

- A profiler can determine relative amount of processing time used by methods.
  - It is your job to judge whether this is inappropriate.
- Optimize the most consuming sections
  - this will achieve high over-all effect

Profiling of Java programs is supported by the Java Virtual Machine, that defines interfaces by which the profilers can read profiling information of a program.



## Screenshot of a Profiler

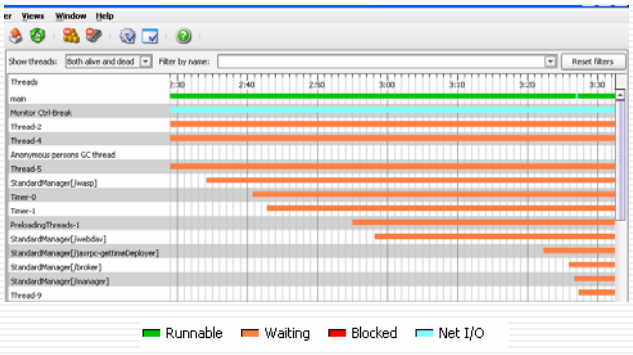


## Extensions of Profilers

- Monitor thread activity
  - This simplifies the task of finding threads blocking one another.
- Provide means for “heap walking”
  - to trace memory leaks
  - walk graphs of objects which are not released and thus cannot be garbage collected



## Thread Monitoring






## Load Testing

Simulating real use



## Load Testing Tools

Load Testing Tools simulate user load to a system. This helps track how the system works under heavy load.

- Load Test Preparation
  1. Usage patterns of users are defined as scripts (called *agendas*). Recording tools help simplify this task.
  2. The agendas of different user roles can be combined to represent typical system usages
     - e.g. 98% browsing users with 2% of users buying products



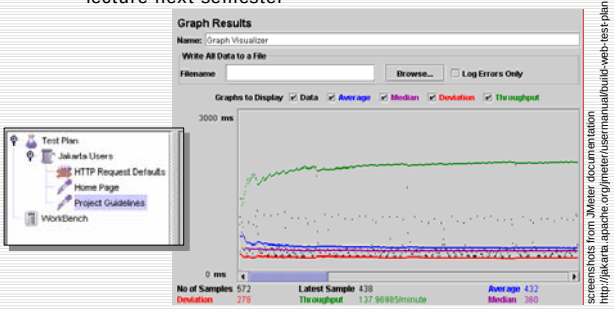
## Load Test

- Load Test Execution
  - load testing tool plays agendas against the system
    - the number of simulated users is increased over time, e.g.:
      - from 50 to 2000 concurrent users or
      - from 10 until the tested system breaks down
  - load testing tool records performance data:
    - average response time per request,
    - percentage of dropped requests
    - percentage of error pages sent back
- The measured data can be use to ...
  - determine the *maximum number of concurrent users* a system configuration can handle or
  - test whether the system is able to *handle a given load*



## JMeter

- Free Java Tool: JMeter
  - you will learn more about it in the Web-Engineering lecture next semester




screenshots from JMeter documentation: http://jakarta.apache.org/jmeter/usermanual/build-web-test-plan.html

## References


- Books
  - [1] Eric M. Burke & Brian M. Coyner. **Java Extreme Programming Cookbook.** Chapter 4 (JUnit) available at http://www.oreilly.com/catalog/jextprockbk/chapter/ch04.pdf
- Articles
  - [2] Thomas McGibbon. **Analysis of Two Formal Models: VDM and Z.** available at http://www.dacs.dtic.mil/techs/2fmethods/vdm-z.pdf
  - [3] Extreme Programming FAQ. http://www.jera.com/techinfo/xpfaq.html