LTAT.05.006: Software Testing

Lecture 01: Introduction to Software Testing

Dietmar Pfahl
email: dietmar.pfahl@ut.ee

Spring 2019
Structure of Lecture 1

• Introduction and Motivation
• Course Information
• Definitions and Principles
• Lab 1
2012: Knight Capital loses 440M USD

- August 12th: New Trading Software installed
- Administrator forgets to deploy on one out of eight server nodes
- New code repurposed a flag previously used for testing scenarios
- On that one server node, old trading algorithm interprets flag differently and starts buying and selling 100 different stocks randomly without human verification
- NYSE has to suspend trade of several stocks
- Knight Capital loses 440 Mio USD in only 30 minutes, until system is suspended
- Investors have to raise 440 Mio USD in order to rescue the company
List of SW Problems from Guru99 /1

• In April 2015, Bloomberg terminal in London crashed due to software glitch affected more than 300,000 traders on financial markets. It forced the government to postpone a 3bn pound debt sale.

• Nissan cars have to recall over 1 million cars from the market due to software failure in the airbag sensory detectors. There has been reported two accident due to this software failure.

• Starbucks was forced to close about 60 percent of stores in the U.S and Canada due to software failure in its POS system. At one point store served coffee for free as they were unable to process the transaction.
List of SW Problems from Guru99 /2

- Some of the Amazon’s third party retailers saw their product price is reduced to 1p due to a software glitch. They were left with heavy losses.
- Vulnerability in Window 10. This bug enables users to escape from security sandboxes through a flaw in the win32k system.
- In 2015, fighter plane F-35 fell victim to a software bug, making it unable to detect targets correctly.
- In May of 1996, a software bug caused the bank accounts of 823 customers of a major U.S. bank to be credited with 920 million US dollars.
List of SW Problems from Guru99 /3

- In April of 1999, a software bug caused the failure of a $1.2 billion military satellite launch, the costliest accident in history
- China Airlines Airbus A300 crashed due to a software bug on April 26, 1994, killing 264 persons
- In 1985, Canada's Therac-25 radiation therapy machine malfunctioned due to software bug and delivered lethal radiation doses to patients, leaving 3 people dead and critically injuring 3 others.
Cost of Testing / Cost of not Testing

2014 industrial survey of 1543 executives from 25 countries:
• Testing and quality assurance of software-intensive systems accounts for roughly 26% of IT budgets [1]

2013 study by researchers at the University of Cambridge:
• Global cost of locating and removing bugs from software has risen to $312 billion annually and it makes up half of the development time of the average project [2].

Sources:
Recall Exercise: A Pen

• Quality?

• Testing?
Software Quality – Definition

• **Software quality is the degree of conformance to explicit or implicit requirements and expectations**

Explanation:
• *Explicit*: clearly defined and documented
• *Implicit*: not clearly defined and documented but indirectly suggested
• *Requirements*: business/product/software requirements
• *Expectations*: mainly end-user expectations
Software Quality – Definition

• **Software quality is the degree of conformance to explicit or implicit requirements and expectations**

Explanation:

• *Explicit*: clearly defined and documented
• *Implicit*: not clearly defined and documented but indirectly suggested
• *Requirements*: business/product/software requirements
• *Expectations*: mainly end-user expectations
Software Product Quality Model
– ISO 25010 Standard
Software Quality Assurance (SQA)

- SQA is a set of activities for ensuring quality in software engineering processes (that ultimately result in quality in software products).

It includes the following activities:
- Process definition
- Process implementation
- Auditing
- Training

Processes could be:
- Software Development Methodology
- Project Management
- Configuration Management
- Requirements Development/Management
- Estimation
- Software Design
- Testing
- …
Software Quality Control (SQC)

- SQC is a set of activities for ensuring quality in software products.

It includes the following activities:
- Reviews
  - Requirement Review
  - Design Review
  - Code Review
  - Deployment Plan Review
  - Test Plan Review
  - Test Cases Review
- Testing
  - Unit Testing
  - Integration Testing
  - System Testing
  - Acceptance Testing
What is Software Testing?
What is Software Testing? (Static & Dynamic)

Confirm quality (**pass-test**) vs. Find defects (**fail-test**)
What is Software Testing?

The process of evaluating a program or a system

Verifying

Validating
What is Software Testing?

The process of evaluating a program or a system

**Verifying** that the product has been developed right

**Validating** that the right product has been developed
Verification

Definition

• The process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase.

Objective

• To ensure that the product is being built according to the requirements and design specifications. In other words, to ensure that work products meet their specified requirements.

Question

• Are we building the product right?
Validation

Definition
• The process of evaluating software during or at the end of the development process to determine whether it satisfies specified (or implicit) business requirements.

Objective
• To ensure that the product actually meets the user’s needs, and that the requirements were correct in the first place. In other words, to demonstrate that the product fulfills its intended use when placed in its intended environment.

Question
• Are we building the right product?

Evaluation Items:
- User requirements, Final product/software

Activities:
- Requirements review, Acceptance testing
Verification vs. Validation

Legend:
- Executable Work Products
- Tests

Source: SEI at CMU, Donald Firesmith
Testing in the Software Development Lifecycle (SDLC)

- Requ. Spec.
- Design
- Code
- Unit test
- Integration test
- System test
- Acceptance test

See Fig 1.6 Textbook
Software Testing Life Cycle (STLC)

- The STLC defines the steps/stages/phases in testing of software.
STLC integrated with SDLC
Test Techniques

Keyword testing
Exploratory testing

Load testing
Stress-testing
Black box security testing

Input space partitioning
Boundary values
Error guessing
All combinations/pairwise/n-wise
Coverage criterions
Mutation testing
Test Complexity – Quiz

Example:
• 30 variables, 2 levels
• Test all combinations

How long does it take to test, if 5 tests/sec can be executed automatically?

Answer choices:
1. Less than 10 sec
2. Less than 1 min
3. Less than 1 hour
4. Less than 1 day
5. Less than 1 year
6. More than 1 year
Test Complexity

Example:

- 30 variables, 2 levels
  -> $2^{30} \approx 10^9$
  combinations to test
- 5 tests/second ->
  214748364.8 sec or
  6.8 years of testing!
Structure of Lecture 1

• Introduction and Motivation
• Course Information
• Definitions and Principles
• Lab 1
Course Information/Overview

• Level: Course at bachelor's level (in English), 2\textsuperscript{nd} year
• Credits: 6 ECTS
• Prerequisite:
  – Compulsory: MTAT.03.094/LTAT.05.008 Software Engineering (6 ECTS)
  – Recommended: MTAT.03.130 Object-oriented Programming (6 ECTS)
• Work load:
  – Lectures (incl. practical work): 64 person-hours – incl. lab and exam sessions
  – Independent work (outside classroom): 92 person-hours
• Assessment:
  – 11 Homework Assignments (work in pairs) – 60% of grade (~5 ph per lab = 55 ph)
  – 10 Quizzes (individual) – 10% of grade (~10 ph)
  – Exam (written) – 30% of grade (~27 ph)
• Grade scale: A (90%+), B(80%+), C(70%+), D(60%+), E(50%+), F
Course Objectives

• The course
  – addresses the essential concepts of software quality control and testing
  – introduces various testing strategies and types of testing

• … and will also give an overview of
  – different software defects
  – test tools & automation
  – organizational aspects of software testing
Letter Grades

- **A** - An excellent performance, clearly outstanding. The candidate demonstrates excellent judgement and a high degree of independent thinking.
- **B** - A very good performance. The candidate demonstrates sound judgement and a very good degree of independent thinking.
- **C** - A good performance in most areas. The candidate demonstrates a reasonable degree of judgement and independent thinking in the most important areas.
- **D** - A satisfactory performance, but with significant shortcomings. The candidate demonstrates a limited degree of judgement and independent thinking.
- **E** - A performance that meets the minimum criteria, but no more. The candidate demonstrates a very limited degree of judgement and independent thinking.
- **F** - A performance that does not meet the minimum academic criteria. The candidate demonstrates an absence of both judgement and independent thinking.
Lectures (J. Liivi 2-111)

- Lecture 1 (14.02) – Introduction to Software Testing
- Lecture 2 (21.02) – Basic Black-Box and White-Box Testing Techniques (overview)
- Lecture 3 (28.02) – BBT advanced: Combinatorial Testing
- Lecture 4 (07.03) – WBT advanced: Control-Flow and Data-Flow Coverage Criteria
- Lecture 5 (14.03) – Test Lifecycle, Test Levels, Test Tools
- Lecture 6 (21.03) – BBT advanced: State-Transition Testing
- Lecture 7 (28.03) – Behavioural Testing / GUI Testing / Visual Testing
- Lecture 9 (11.04) – Test-Suite Effectiveness / Mutation Testing
- Lecture 11 (25.04) – Defect Estimation / Test Documentation, Organisation and Process Improvement (Test Maturity Model)
- 02.05 - no lecture
- Lecture 12 (09.05) – Industry Guest Lecture (to be announced)
- Lecture 13 (16.05) – Exam Preparation
**Lectures / Labs (HW) / Quiz Schedule**

<table>
<thead>
<tr>
<th>W24</th>
<th>W25</th>
<th>W26</th>
<th>W27</th>
<th>W28</th>
<th>W29</th>
<th>W30</th>
<th>W31</th>
<th>W32</th>
<th>W33</th>
<th>W34</th>
<th>W35</th>
<th>W36</th>
<th>W37</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lec 1</td>
<td>Lec 2</td>
<td>Lec 3</td>
<td>Lec 4</td>
<td>Lec 5</td>
<td>Lec 6</td>
<td>Lec 7</td>
<td>Lec 8</td>
<td>Lec 9</td>
<td>Lec 10</td>
<td>Lec 11</td>
<td>X</td>
<td>Lec 12</td>
<td>Lec 13</td>
</tr>
<tr>
<td>X</td>
<td>HW 1</td>
<td>HW 2</td>
<td>HW 3</td>
<td>HW 4</td>
<td>HW 5</td>
<td>HW 6</td>
<td>HW 7</td>
<td>HW 8</td>
<td>HW 9</td>
<td>HW 10</td>
<td>X</td>
<td>HW 11</td>
<td>X</td>
</tr>
<tr>
<td>Q 1</td>
<td>Q 2</td>
<td>Q 3</td>
<td>Q 4</td>
<td>Q 5</td>
<td>Q 6</td>
<td>Q 7</td>
<td>Q 8</td>
<td>Q 9</td>
<td>Q 10</td>
<td>Q 11</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Lab Assignments (HW) to be submitted via course wiki**
(1 week time / strict deadlines / 24 h grace period with penalty – afterwards 0 points)

**Quizzes to be done in Moodle**
(2 attempts / 15 min each / Fri 9:00 am– Mon 11:30 am / 10 best count)
Lab Sessions (J. Liivi 2-… various rooms)

Preparation, Execution, Report – **Work in Pairs**

1. Lab 1 (week 25: Feb 19 & 20) - Debugging (10 marks)
2. Lab 2 (week 26: Feb 26 & 27) - Basic Black-Box-Testing (10 marks)
3. Lab 3 (week 27: Mar 05 & 06) - Combinatorial Testing (10 marks)
4. Lab 4 (week 28: Mar 12 & 13) - Basic White-Box Testing (10 marks)
5. Lab 5 (week 29: Mar 19 & 20) - Automated Web-App Testing (10 marks)
6. Lab 6 (week 30: Mar 26 & 27) - Automated Integration Testing (10 marks)
7. Lab 7 (week 31: Apr 02 & 03) - Web-Application Testing in the CI/CD Pipeline (10 marks)
8. Lab 8 (week 32: Apr 09 & 10) - Automated GUI Testing (10 marks)
9. Lab 9 (week 33: Apr 16 & 17) - Mutation Testing (10 marks)
10. Lab 10 (week 34: Apr 23 & 24) - Static Code Analysis (10 marks)
11. No labs in week 35 (due to holiday on May 1)
12. Lab 11 (week 36: May 07 & 08) - Document Inspection and Defect Prediction (10 marks)

Send reports via submission button on course wiki before your next lab starts. Only PDF files will be accepted.
GO TO LABS !!!!!

(if you don’t, you will lose marks)
Final Exam

Written exam (30%)
• Based on textbook, lectures and lab sessions
• Multiple-choice part **closed book** / Other parts open book & open laptop

Dates:
• Exam 1: Tuesday, 21-May-2019 at 08:15-09:55, room J. Liivi 2-403/405 - capacity limits: 30/35
• Exam 2: Friday, 24-May-2019, 14:15-15:55, room J. Liivi 2-404/405 - capacity limit: 30/35
• Retake Exam (resit): Thursday, 13-June-2018 at 10:15-11:55, room J. Liivi 2-611
Literature

- **Practical Software Testing – A Process-Oriented Approach**
  - Author: I. Burnstein
  - Year: 2003

- **Introduction to Software Testing**
  - Authors: P. Ammann and J. Offutt
  - Year 2017 (2nd ed.)
Software Testing

Course Responsible / Instructor: Dietmar Pfahl (dietmar.pfahl at ut dot ee) - room: 302

Lab Supervisors (TAs):
- Ezequiel Scott (ezequielscott at gmail dot com)
- Claudia Kittask (claudiakittask at gmail dot com)
- Hanna Tagen (hanna dot tagen at gmail dot com)

Lectures (begin in week 24 of the academic year, on 14-Feb-2019):
- Thursday 10:15 - 12:00, J. Liivi 2-111

Labs (practice learning: begin in week 25):
- Group 1: Tuesday 08.15 - 10.00, r403 - Claudia
- Group 3: Tuesday 14.15 - 16.00, r224 - Ezequiel
- Group 4: Wednesday 08.15 - 10.00, r207 - Ezequiel
- Group 5: Wednesday 14.15 - 16.00, r206 - Hanna
- Group 6: Wednesday 14.15 - 16.00, r405 - Ezequiel
SIGN UP TO MESSAGE BOARD (Slack)

(if you don’t, you will miss up to date info)
Sign-Up Link for Slack (Check out email of Monday at 20:20)

• Before the first labs start next week, please sign up to the course Slack channel. You will get your homework feedback exclusively via Slack from the lab supervisors. Here is the sign-up link:

• https://join.slack.com/t/swt2019course/shared_invite/enQtNTQ4OTIyNDc0NzU5LTVjYjk2NzcyNTQ4NzJmZWQyMTU2MjRjZmVjMzIwNDI2NGEzNjA3Yzg5Zjg1OWE0M2Q5MWQzMmUwYzI1MjQwYml
Structure of Lecture 1

• Introduction and Motivation
• Course Information
• Definitions and Principles
• Lab 1
Definition 1: Error – Fault – Failure
(according to IEEE Standard & textbook)

- **Failure** is an event caused by a **fault**, and a **fault** is an anomaly of the software caused by an **error**
  - **Error** – mistake made by human (e.g., programmer)
  - **Fault** – wrong/missing statement in the software (code)
  - **Failure** – inability to perform the program’s required functions (correctly)
  - Defect ? – Bug ?

- **Debugging** / Fault localization – localizing, repairing, re-testing.
Origins and Impact of Faults

Fault sources
- Lack of skills/training
- Oversight
- Poor communication
- ‘Lost in translation’
- Immature process

Impact on / of software program

Errors
Faults
Failures

User’s point of view
- Poor quality software
- User dissatisfaction

Fig 3.1
Definition 2: Error – Fault – Failure
(as it is often used in IDEs/tools)

- **Failure** is an event caused by an **error**, **error** is a state of the program caused by a **fault** in the code
- **Fault** – wrong/missing statement in code (resulting in error)
- **Error** – incorrect program state (may result in a failure)
- **Failure** – inability to perform its required functions (correctly)

- **Debugging** / Fault localization – localizing, repairing, re-testing.
Definition 2: Error – Fault – Failure

Example:

Input: x = [2, 7, 0]

Expected (=correct) result?

Actual result?

```java
public static int numZero (int[] x) {
// Effects: if x==null throw NullPointerException
//          else return the number of occurrences of 0 in x
    int count = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i] == 0) {
            count++;
        }
    }
    return count;
}
```
Definition 2: Error – Fault – Failure

Example:

Input: \( x = [2, 7, 0] \)
Expected (=correct) result: 1
Actual result: 1

Failure? Error? Fault?

```java
public static int numZero (int[] x) {
    // Effects: if x==null throw NullPointerException
    //          else return the number of occurrences of 0 in x
    int count = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i] == 0) {
            count++;
        }
    }
    return count;
}
```
Definition 2: Error – Fault – Failure

Example:

Input: \( x = [2, 7, 0] \)
Expected (=correct) result: 1
Actual result: 1

Failure=no Error? Fault?

```java
public static int numZero (int[] x) {
    // Effects: if x==null throw NullPointerException
    //          else return the number of occurrences of 0 in x
    int count = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i] == 0) {
            count++;
        }
    }
    return count;
}
```
Definition 2: Error – Fault – Failure
Example:

```
public static int numZero (int[] x) {
    // Effects: if x==null throw NullPointerException
    //          else return the number of occurrences of 0 in x
    int count = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i] == 0) {
            count++;
        }
    }
    return count;
}
```

Program state: x, i, count, PC

Input: x = [2, 7, 0]  Error?

<table>
<thead>
<tr>
<th>State 1:</th>
<th>State 2:</th>
<th>State 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC=public static ... x=[2, 7, 0] count=? i=?</td>
<td>PC=int count = ... x=[2, 7, 0] count=0 i=?</td>
<td>PC=for (int i = ... x=[2, 7, 0] count=0 i=1</td>
</tr>
</tbody>
</table>
Definition 2: Error – Fault – Failure
Example:

```
public static int numZero (int[] x) {
    // Effects: if x==null throw NullPointerException
    //          else return the number of occurrences of 0 in x
    int count = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i] == 0) {
            count++;
        }
    }
    return count;
}
```

Input: x = [2, 7, 0]  Error=yes Fault?

<table>
<thead>
<tr>
<th>State 1: PC=public static …</th>
<th>State 2: PC=int count = …</th>
<th>State 3: PC=for (int i = …</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=[2, 7, 0] count=? i=?</td>
<td>x=[2, 7, 0] count=0 i=?</td>
<td>x=[2, 7, 0] count=0 i=1</td>
</tr>
</tbody>
</table>
Definition 2: Error – Fault – Failure

Example:

State 1:
- PC=public static
- x=[2, 7, 0]
- count=？
- i=？

Input: x = [2, 7, 0]  Error=yes  Fault=yes

```
public static int numZero (int[] x) {
    // Effects: if x==null throw NullPointerException
    //          else return the number of occurrences of 0 in x
    int count = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i] == 0) {
            count++;
        }
    }
    return count;
}
```

Program state: x, i, count, PC

State 2:
- PC=int count = ...
- x=[2, 7, 0]
- count=0
- i=？

State 3:
- PC=for (int i = ...
- x=[2, 7, 0]
- count=0
- i=1
Definition 2: Error – Fault – Failure

Example:

Input: $x = [2, 7, 0]$
Expected (=correct) result: 1
Actual result: 1
Failure=no Error=yes Fault=yes

Program state: $x, i, count, PC$

```java
public static int numZero (int[] x) {
    // Effects: if x==null throw NullPointerException
    // else return the number of occurrences of 0 in x
    int count = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i] == 0) {
            count++;
        }
    }
    return count;
}
```
Definition 2: Error – Fault – Failure

Example:

Input: \( x = [0, 7, 2] \)
Expected (=correct) result: ?
Actual result: ?
Failure=? Error=? Fault=?

Program state: \( x, i, count, PC \)

```java
public static int numZero (int[] x) {
    // Effects: if x==null throw NullPointerException
    //          else return the number of occurrences of 0 in x
    int count = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i] == 0) {
            count++;
        }
    }
    return count;
}
```
Definition 2: Error – Fault – Failure

Example:

Input: \( x = [0, 7, 2] \)
Expected (=correct) result: 1
Actual result: 0
Failure=yes Error=yes Fault=yes

Program state: x, i, count, PC

```java
public static int numZero (int[] x) {
    // Effects: if x==null throw NullPointerException
    //          else return the number of occurrences of 0 in x
    int count = 0;
    for (int i = 1; i < x.length; i++) {
        if (x[i] == 0) {
            count++;
        }
    }
    return count;
}
```
Test Case

• A test case is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly.

• Templates and examples of formal test case documentation can be found here:

  http://softwaretestingfundamentals.com/test-case/
A **Test Case** consists of:

- A set of inputs + expected outputs
- Execution conditions

  Example of ‘execution condition’:
  
  When pressing the ‘save’ button of a word processor, what happens depends on what you did previously (e.g., what you typed in or deleted)

---

<table>
<thead>
<tr>
<th>Seq.No</th>
<th>Condition to be tested</th>
<th>Test Data</th>
<th>Expected result</th>
<th>successful</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Test Suite** = set of Test Cases

**Test Data** = input to a Test Case

**Test Oracle** = condition that determines whether a test case passed or failed (-> fail happens if actual output is different from expected output)

**Test Verdict** = decision of whether a test passed or failed
**Test Case – Examples**

- A **test case** consists of:
  - A set of inputs
  - Execution conditions
  - Expected outputs

<table>
<thead>
<tr>
<th>TC ID</th>
<th>TC Name</th>
<th>Description</th>
<th>Steps</th>
<th>Expected result</th>
</tr>
</thead>
</table>
| TC1   | TC1_FileUpdate Scenario1 | Validate that system updates the file when first character is A and second character is a digit. | 1. Open the application.  
2. Enter first character as “A”  
3. Enter second character as a digit | File is updated. |
| TC2   | TC2_FileUpdate Scenario2 | Validate that system updates the file when first character is B and second character is a digit. | 1. Open the application.  
2. Enter first character as “B”  
3. Enter second character as a digit | File is updated. |
Test Oracle

= a mechanism used for determining whether a test has passed or failed

- It provides the expected result for a test, e.g. from
  - Specification document, mathematical formula, program model, other program, historical data, belief of tester, …
- **Test Verdict** = the actual judgment after a test case terminates – pass/fail/warning/don’t know

Sometimes it is difficult/impossible to find an oracle
- Non-functional quality aspects / Scientific computing
Test Script

- A **Test Script** is a set of instructions (written using a scripting/programming language) that is performed on a system under test to verify that the system performs as expected. Test scripts are used in automated testing.

- Examples of Test Frameworks supporting test scripting:
  - JUnit, Selenium, Sikuli, …
Test Script – Examples

JUnit

@Test
class TestRental {
  public void shortRegularRental() {
    Customer customer = new Customer("Cust");
    Movie movie = new Movie("Groundhog Day", REGULAR);
    Rental rental = new Rental(movie, 2); // 2 days rental
    customer.addRental(rental);

    String expected = "Rental Record for Cust\n";
    expected += "\tGroundhog Day\t2.0\n";
    expected += "Amount owed is 2.0\n";
    expected += "You earned 1 frequent renter points";

    Assert.assertEquals(expected, customer.statement());
  }
}

Sikuli

def sample_test_script (self):
type ("TextA")
click (ImageButtonA)
assertExist (ImageResultA)

@Test
public void shortRegularRental() {
  Customer customer = new Customer("Cust");
  Movie movie = new Movie("Groundhog Day", REGULAR);
  Rental rental = new Rental(movie, 2); // 2 days rental
  customer.addRental(rental);

  String expected = "Rental Record for Cust\n";
  expected += "\tGroundhog Day\t2.0\n";
  expected += "Amount owed is 2.0\n";
  expected += "You earned 1 frequent renter points";

  Assert.assertEquals(expected, customer.statement());
}
Why do faults occur in software? (Ch 3)

- Software is written by humans
  - Who know something, but not everything
  - Who have skills, but aren’t perfect
  - Who don’t usually use rigorous methods
  - Who do make mistakes (errors)

- Under increasing pressure to deliver to strict deadlines
  - No time to check, assumptions may be wrong
  - Systems may be incomplete

- Software is complex, abstract and invisible
  - Hard to understand
  - Hard to see if it is complete or working correctly
  - No one person can fully understand large systems
  - Numerous external interfaces and dependencies
Principle 1 – Purpose of testing

• Testing is the process of exercising a software component using a selected set of test cases, with the intent of
  – Revealing defects
  – Evaluating quality

• “The testing paradox”

  Testing can’t show that defects don’t exist
  --
  The more you test software, the more immune it becomes to your tests
Principles (cont.)

2: A good test case – When the test objective is to detect defects, then a good test case is one that has high probability of revealing a yet undetected defect(s)

3: Test result – The results should be inspected meticulously

4: Expected output – A test case must contain the expected output
Principles (cont.)

5: **Input** – Test cases should be developed for both valid and invalid input conditions

6: **Fault content estimation** – The probability of the existence of additional defects in a software component is proportional to the number of defects already detected in that component

7: **Test organization** – Testing should be carried out by a group that is independent of the development group
Principles (cont.)

8: **Repeatable** – Tests must be repeatable and reusable

9: **Planned** – Testing should be planned

10: **Life cycle** – Testing activities should be integrated into the software life cycle

11: **Creative** – Testing is a creative and challenging task
Recommended Textbook Exercises

- Chapter 1
  - 2, 3
- Chapter 2
  - 1, 3, 5
- Chapter 3
  - 1, 2, 3, 6
Structure of Lecture 1

• Introduction and Motivation
• Course Information
• Definitions and Principles
• Lab 1
Lab 1 – Debugging

System 1

- Issue 1
- Issue 2
- Issue 3

Faults?
Faults?
Faults?

System 2

- Issue 1
- Issue 2

Faults?
Faults?

Localization steps & heuristics for each fault

Use IntelliJ Debugger

Submission:
At 23:59 on day before next lab
Lab 1 – Debugging

- Thought process for setting breakpoints and deciding where to step in next to see the program state.
Starting Point: Issue Report

- Example Report
  - Admin Data
  - Short Description
  - Reproduction Steps (input)
  - Expected vs Actual Result
  - Additional Information (screen shots, stack traces, etc.)
  - Comments / Discussion
Issue Report – Sys_1: HeapSort

• Issue report 1:

Description:
The program should heapify any given list of positive integers but the resulting tree (and list) does not meet the max binary heap structure.

Input:
heapifying a list of integers - [1, 2, 5, 7, 11, 10, 3, 4, 9, 1, 0]

Input code:
List<Integer> heapList = new ArrayList();
heapList.add(1);
heapList.add(2);
heapList.add(5);
heapList.add(7);
heapList.add(6);
heapList.add(8);
heapList.add(11);
heapList.add(10);
heapList.add(3);
heapList.add(4);
heapList.add(9);
heapList.add(1);
heapList.add(0);
System.out.println("List before heapifying:");

Expected output:
List before heapifying:
[1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0]
After heapifying:
[11, 10, 8, 7, 9, 1, 5, 2, 3, 4, 6, 1, 0]

Actual output:
List before heapifying:
[1, 2, 5, 7, 6, 8, 11, 10, 3, 4, 9, 1, 0]
After heapifying:
[11, 9, 5, 3, 9, 7, 1, 6, 3, 4, 2, 1, 0]
# System 1: HeapSort – Build the Heap

<table>
<thead>
<tr>
<th>Heap</th>
<th>New element</th>
<th>Swap elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6, 5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6, 5, 3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6, 5, 3, 1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>6, 5, 3, 1, 8</td>
<td>5, 8</td>
<td></td>
</tr>
<tr>
<td>6, 8, 3, 1, 5</td>
<td>6, 8</td>
<td></td>
</tr>
<tr>
<td>8, 6, 3, 1, 5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8, 6, 3, 1, 5, 7</td>
<td>7, 3</td>
<td></td>
</tr>
<tr>
<td>8, 6, 7, 1, 5, 3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>8, 6, 7, 1, 5, 3, 2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8, 6, 7, 1, 5, 3, 2, 4</td>
<td>1, 4</td>
<td></td>
</tr>
<tr>
<td>8, 6, 7, 4, 5, 3, 2, 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

List to be sorted in ascending order:

\[ [6 \ 5 \ 3 \ 1 \ 8 \ 7 \ 2 \ 4] \]

Heapified list:

\[ [8 \ 6 \ 7 \ 4 \ 5 \ 3 \ 2 \ 1] \]
## System 1: HeapSort – Sorting /1

<table>
<thead>
<tr>
<th>Heap</th>
<th>Swap elements</th>
<th>Delete element</th>
<th>Sorted array</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>8, 6, 7, 4, 5, 3, 2, 1</td>
<td>8, 1</td>
<td></td>
<td></td>
<td>Swap 8 and 1 to delete 8 from heap</td>
</tr>
<tr>
<td>1, 6, 7, 4, 5, 3, 2, 8</td>
<td>8</td>
<td></td>
<td></td>
<td>Delete 8 from heap &amp; add to sorted array</td>
</tr>
<tr>
<td>1, 6, 7, 4, 5, 3, 2</td>
<td>1, 7</td>
<td></td>
<td>8</td>
<td>Swap 1 and 7 as they are not in order</td>
</tr>
<tr>
<td>7, 6, 1, 4, 5, 3, 2</td>
<td>1, 3</td>
<td></td>
<td>8</td>
<td>Swap 1 and 3 as they are not in order</td>
</tr>
<tr>
<td>7, 6, 3, 4, 5, 1, 2</td>
<td>7, 2</td>
<td></td>
<td>8</td>
<td>Swap 7 and 2 to delete 7 from heap</td>
</tr>
<tr>
<td>2, 6, 3, 4, 5, 1, 7</td>
<td>7</td>
<td></td>
<td>8</td>
<td>Delete 7 from heap &amp; add to sorted array</td>
</tr>
<tr>
<td>2, 6, 3, 4, 5, 1</td>
<td>2, 6</td>
<td></td>
<td>7, 8</td>
<td>Swap 2 and 6 as they are not in order</td>
</tr>
</tbody>
</table>

...
## System 1: HeapSort – Sorting

<table>
<thead>
<tr>
<th>Heap</th>
<th>Swap elements</th>
<th>Delete element</th>
<th>Sorted array</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>3, 2, 1</td>
<td>3, 1</td>
<td></td>
<td>4, 5, 6, 7, 8</td>
<td>Swap 3 and 1 to delete 3 from heap</td>
</tr>
<tr>
<td>1, 2, 3</td>
<td></td>
<td>3</td>
<td>4, 5, 6, 7, 8</td>
<td>Delete 3 from heap &amp; add to sorted array</td>
</tr>
<tr>
<td>1, 2</td>
<td>1, 2</td>
<td></td>
<td>3, 4, 5, 6, 7, 8</td>
<td>Swap 1 and 2 as they are not in order</td>
</tr>
<tr>
<td>2, 1</td>
<td>2, 1</td>
<td></td>
<td>3, 4, 5, 6, 7, 8</td>
<td>Swap 2 and 1 to delete 2 from heap</td>
</tr>
<tr>
<td>1, 2</td>
<td>2</td>
<td></td>
<td>3, 4, 5, 6, 7, 8</td>
<td>Delete 2 from heap &amp; add to sorted array</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>2, 3, 4, 5, 6, 7, 8</td>
<td>Delete 1 from heap &amp; add to sorted array</td>
</tr>
</tbody>
</table>

Completed: 1, 2, 3, 4, 5, 6, 7, 8

Issue Report 1– Sys_2: 8-Queens Problem

- Issue report 1 (Hint: corresponds to 3 bugs)

  **Description:**
  When running the program, it should return a list of generations and the correct solution that it found. Instead, it throws an exception after 1000 generations.

  **Input:**
  Running the code with the population size of 100.

  **Input code:**
  ```java
  Population pop = new Population(100);
  runAlgorithm(pop);
  ```

  **Expected output:**
  Generation: 1  Current highest fitness: <?>
  Generation: 2  Current highest fitness: <?>
  ...
  Found suitable board state on generation <?): <[x1,x2,x3,x4,x5,x6,x7,x8]>
  Here is the found solution as a board where . marks an empty spot and X marks a queen
  <printout of 1 correct solution of 92 possible>

  **Actual output:**
  Generation: 1  Current highest fitness: 22
  Generation: 2  Current highest fitness: 22
  Generation: 3  Current highest fitness: 26
  ...
  Generation: 1000  Current highest fitness: 38
Issue Report 1 – Sys_2: 8-Queens Problem

- Issue report 1 (Hint: corresponds to 3 bugs)

  **Description:**
  When running the program, it should return a list of generations and the correct solution
  that it found. Instead, it throws an exception after 1000 generations.

  **Input:**
  Running the code with the population size of 100.

  **Input code:**
  ```java
  Population pop = new Population(100);
  runAlgorithm(pop);
  ```

  **Expected output:**
  - Generation: 1  Current highest fitness: <?
  - Generation: 2  Current highest fitness: <?
  ...  
  Found suitable board state on generation <?: <[x1,x2,x3]>
  Here is the found solution as a board where . marks an empty space
  <printout of 1 correct solution of 92 possible>

  **Actual output:**
  - Generation: 1  Current highest fitness: 22
  - Generation: 2  Current highest fitness: 22
  - Generation: 3  Current highest fitness: 26
  ...  
  - Generation: 1000  Current highest fitness: 38

  **Exception in thread "main" java.lang.Exception: Didn't find solution in 1000 generations at Algorithm.generation(Algorithm.java:119)
  at Algorithm.generation(Algorithm.java:129)**

  **Comments:**
  This issue might not be reproducible line-to-line due to randomness in the algorithm,
  meaning the current highest fitness can vary. But the core of the issue is reproducible
  (exception).

  **Hints:**
  1. There are 3 bugs that correspond to this issue
  2. You can consider this issue fixed only when all 3 bugs have been fixed. In order to be
     sure that you have fixed the correct bugs, run the program multiple times. The expected
     output may sometimes be seen because of the randomness of the data, make sure that the
     correct output appears every time you run the program.
  3. While you have not yet fixed all the bugs, depending on the order in which you find and
     fix them, you might see any of the following output:
     - The described exception is thrown.
     - The program outputs a board state that it claims to be correct. However, the
       board state is not correct as all the queens are positioned in one single diagonal
       on the board.
     - The program outputs a board state that it claims to be correct. However, the
       board state is not correct as there is at least one clash visible on the board.
Issue Report 2– Sys_2: 8-Queens Problem

- Issue report 2 (Hint: corresponds to 1 bug) (This issue appears after Issue 1 has been fixed)

Description:
Based on past projects using genetic algorithms, the average amount of generations should be less than 87 and the program should produce the correct output in less than 100 generations on at least 75% of the runs. However, the performance is much worse, the average amount of generations is over 100 and it only gets the solution in under 100 generations in less than 62% of the time. On very few runs, the program throws an exception of not finding a solution in under 1000 generations.

Input:
Calculated average amount of generations and percentage of runs where solution was found in under 100 generations with population size 100 and 1000 runs.

Input code:
```java
public static List<Integer> generationCounts = new ArrayList<>();
public static void main(String[] args) throws Exception {
    for (int i = 0; i<1000; i++) {
        pop = new Population(100);
        generation(pop);
        generationCounts.add(counter+1);
        counter = 0;
    }
    System.out.println(calculateAverage(generationCounts));
    System.out.println(calculatePercent(generationCounts));
    generationCounts.removeAll(generationCounts); }
```
Issue Report 2– Sys_2: 8-Queens Problem

* Issue report 2 (Hint: corresponds to 1 bug) (This issue appears after Issue 1 has been fixed)

Description:
Based on past projects using genetic algorithms, the average amount of generations should be less than 87 and the program should produce the correct output in less than 100 generations on at least 75% of the runs. However, the performance is much worse, the average amount of generations is over 100 and it only gets the solution in under 100 generations in less than 62% of the time. On very few runs, the program throws an exception of not finding a solution in under 1000 generations.

Input:
Calculated average amount of generations and percentage in under 100 generations with population size 100

Input code:
```java
public static List<Integer> generationCounts = new ArrayList<Integer>();
public static void main(String[] args) throws Exception {
    for (int i = 0; i<1000; i++) {
        pop = new Population(100);
        generation(pop);
        generationCounts.add(counter+1);
        counter = 0;
    }
System.out.println(calculateAverage(generationCounts));
System.out.println(calculatePercent(generationCounts).removeAll(generationCounts));
```

Expected output:
Average generation count < 87
P(generation count<=100) >75%

Actual output generalized and specific:
Average generation count > 87
P(generation count<=100) < 75%
Average generation count: 109.916
P(generation count <100): 57.7%

Comments and hints:
As performance can be affected by many things, the issue reporter has provided their own insight as a hint. You may use this, but don’t have to.

a) Genetic algorithms and their performance are strongly based on evaluations of states and fitness calculations.
b) It is important to check that the code does what the developer has intended it to do. To know what is intended, use Appendix A and helpful methods in the program (main class, the run<method_name>() methods)
Representation of Chess Board

Queen

[2 4 1 6 7 0 2 3]

5th row, 8th position
8th position = index 7
Genetic Algorithm
Genetic Algorithm

Size = 4

Fitness of first individual? +1 in the fitness function per each queen per each clash
Genetic Algorithm

Generate new population:
- Sort by fitness value
- Take upper half
- Then generate new board allocations based on allocation pairs in the upper half
Genetic Algorithm

Generate new population:

Size = 4
To Do & Next Week

• Quiz 1 (in Moodle!):
  – Opens tomorrow morning – closes on Monday before noon!

• Lab 1:
  – Debugging

• Lecture 2:
  – Basic Black-Box and White-Box Testing Techniques (intro)

• In addition to do:
  – Read Burnstein textbook chapters 1-3

Next Week!