Schedule of Lectures

Week 01: Introduction to SE
Week 02: Requirements Engineering I
Week 03: Requirements Engineering II
Week 04: Analysis
Week 05: Development Infrastructure
Week 06: Continuous Development and Integration
Week 07: Project Estimation / Architecture and Design I
Week 08: Architecture and Design II

Week 09: Verification and Validation I
Week 10: Verification and Validation II
Week 11: Refactoring (and TDD)
Week 12: Agile/Lean Methods
Week 13: Industry Guest Lecture
Week 14: Course wrap-up, review and exam preparation
Week 15: Reserve time slot (no lecture scheduled as of today)
Structure of Lecture 10

- Testing Basics
- Testing Levels
- Testing Methods
- Testing Types
- Testing Artefacts
- Metrics
Recap from last week … (Terminology)

- **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)
  - **Defect (?)** – Bug ?

- **Debugging** / Fault localization – localizing, repairing, re-testing.
- **Test Case**, **Test Script**, **Test Oracle**, **Test Verdict**, …
- **Incident**, **Issue**
Recap from last week … (Terminology)

• **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)
  - **Defect (?)** – Bug ?

• **Debugging / Fault localization** – localizing, repairing, retesting.

• **Test Case, Test Script, Test Oracle, Test Verdict, …**

• **Incident, Issue**

  - “I fixed the bug!”
  - “Here comes my bug report”
Recap from last week … (Test Levels)

Actual Needs and Constraints

User Acceptance (alpha, beta test)

Delivered Package

System Specifications

Analysis / Review

System Test

Analysis / Review

Integration Test

Integration Test

System Integration

Module Test

User review of external behavior as it is determined or becomes visible

Subsystem

Unit/Component Specs

Unit/Components

Subsystem

Unit/Component Specs

Review
Recap from last week … (Unit Test)

For CalculatorApp:

```java
@Test
void additionTest() {
    System.out.println("inside additionTest");
    assertAll(
        () -> assertEquals(2, calc.add(1,1), "Doesn't add two positive numbers properly"),
        () -> assertEquals(0, calc.add(-1,1), "Doesn't add a negative and a positive number properly"),
        () -> assertNotEquals(calc, "The calc variable should be initialized")
    );
}

@Test
void divisionTest() {
    System.out.println("inside divisionTest");
    assertThrows(ArithmeticException.class, () -> calc.divide(2,0));
}
```

Why not check this:
(-1) + (-1)
0 + 0

Why not check this:
2 / 1
1 / 2
-1 / 0
Testing is difficult

Assume a ’magic’ Function M

\[ M(x, y) \rightarrow z \]
with \( x, y: \text{int (32 bit)} \)

Exhaustive testing:
How many test cases,
If only valid input (==int) used?

\[ M(x, y) = ? \]
Testing is difficult

Assume a 'magic' Function M

\[ M(x, y) \rightarrow z \]

with \( x, y: \text{int (32 bit)} \)

Exhaustive pos. testing:

\[ 2^{32} \times 2^{32} \]

\[ = 2^{64} \approx 1.8 \times 10^{19} \text{ test cases (input data + expected output)} \]
Testing is difficult

Assume a 'magic' Function M

\[ M(x, y) \rightarrow z = \text{SUM}(x, y) \]
with \( x, y : \text{int (32 bit)} \)

Exhaustive pos. testing:
\[ 2^{32} \times 2^{32} \]
\[ = 2^{64} \sim 1.8 \times 10^{19} \text{ test cases (input data + expected output)} \]
Testing is difficult

Assume a 'magic' Function M

\[ M(x, y) \rightarrow z = \text{SUM}(x, y) \]
with \( x, y: \text{int} \) (32 bit)

Possible approaches:
- ???

White Box

```plaintext
...  
if (x - 100 <= 0) {  
  if (y - 100 <= 0) {  
    if (x + y - 200 == 0) {  
      z = x / (y - 100);  
    }  
  }  
}  
}  
z = x + y; ...
```

Malicious code!
Testing is difficult

Assume a ’magic’ Function M

\[ M(x, y) \rightarrow z = \text{SUM}(x, y) \]

with \( x, y: \text{int (32 bit)} \)

Possible approaches:
- Execute each statement
- Read (review) code

White Box

\[
\ldots
\]

\[
\text{if ( } x - 100 \leq 0 \} \{ \\
\quad \text{if ( } y - 100 \leq 0 \} \{ \\
\quad \quad \text{if ( } x + y - 200 \equiv 0 \} \{ \\
\quad \quad \quad z = x / (y - 100) ; \\
\quad \quad \} \\
\quad \} \\
\} \quad z = x + y ; \ldots
\]
Testing is difficult

Assume a ’magic’ Function M

\[ M(x, y) \rightarrow z = \text{SUM}(x, y) \]

with \( x, y \): int (32 bit)

1st if = true: \( x \leq 100 \)
2nd if = true: \( y \leq 100 \)
3rd if = true: \( x + y = 200 \)

\[ M(100, 100) \rightarrow \text{crash} \]
Black-Box vs. White-Box

External/user view:
- Check conformance with specification -> function coverage

Abstraction from details:
- Source code not needed

Scales up:
- Different techniques at different levels of granularity

Internal/developer view:
- Allows tester to be confident about code coverage

Based on control and data flow:
- Easier debugging

Does not scale up:
- Most useful at unit & integration testing levels, as well as regression testing

USE BOTH!
Black-Box vs. White-Box

Gray-Box Testing

Combines black-box and white-box testing; typically, the focus is on input/output testing (black-box view) which is informed by structural information of the code (white-box view).

Example: The tester knows that certain constraints on the input are checked by the unit under test.

Application, e.g., in regression testing: apply (or update) black-box test cases only where code has been changed;

Different techniques at different levels of granularity

Most useful at unit & integration testing levels, as well as regression testing

USE BOTH!
How do Black-Box and White-Box Testing relate to one another?

Develop an initial Test suite using BB techniques

Analyze the parts of the code uncovered by BB test suite

Enhance the Test suite using WB techniques

Apply BB coverage criteria to enhance it

Apply WB coverage criteria to enhance it
Basic Black-Box Testing Techniques

Following are two techniques that can be used for designing black box tests

• **Equivalence Class Partitioning**: It is a software test design technique “that divides the input data of a software unit into partitions of equivalent data from which test cases can be derived. In principle, test cases are designed to cover each partition at least once. (…) Equivalence partitioning is typically applied to the inputs of a tested component, but may be applied to the outputs in rare cases. The equivalence partitions are usually derived from the requirements specification for input attributes that influence the processing of the test object.” (Source: Wikipedia)

• **Boundary Value Analysis**: It is a software test design technique that involves determination of boundaries for input values and selecting values that are at the boundaries and just inside/outside of the boundaries as test data.
Equivalence Class Partitioning (ECP)

• Split input space into classes which the software handles equivalently with regards to the output produced

  green area = valid
  white area = invalid

• Select test cases to cover each class
ECP – Simple Example

```java
public static boolean adultFunction(int age) {
    boolean adult;
    if (age >= 18)
        adult = true;
    else
        adult = false;
    return adult;
}
```

What are the ECs?

- **green area** = valid
- **white area** = invalid
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

What are the ECs?

green area = valid
white area = invalid
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

Note that this spec is rather vague:
• it is unclear at what age one is an adult
• It is unclear what happens, if invalid input is entered
• It is unclear whether certain plausibility checks about feasible ages are made, e.g.:
  • Can a person be older than 150 years?
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

Note that this spec is rather vague:

• it is unclear at what age one is an adult
• It is unclear what happens, if invalid input is entered
• It is unclear whether certain plausibility checks about feasible ages are made, e.g.:
  • Can a person be older than 150 years?

Use own domain knowledge (adult age starts with 18)
Talk to developers and ask for clarification
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

Output 1 = ‘adult’ $\Rightarrow$ age $\geq$ 18
Output 2 = ‘not adult’ $\Rightarrow$ age < 18

green area = valid
white area = invalid
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

Output 1 = ‘adult’ → age in [18, 150]
Output 2 = ‘not adult’ → age in [0, 18)
Output 3 = ‘invalid input’ → age not in [0, 150]

green area = valid
white area = invalid

[18, 150]  not in [0, 150]
[0, 18]
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

Output 1 = ‘adult’ \(\Rightarrow\) age in \([18, 150]\)
Output 2 = ‘not adult’ \(\Rightarrow\) age in \([0, 18)\)
Output 3 = ‘invalid input’ \(\Rightarrow\) age not in \([0, 150]\)

Green area = valid
White area = invalid

Could be refined into:
age < 0
age > 150
age not an int
ECP – Simple Example

Look at specification:

Based on the age of a person, the program decides whether the person is an adult or not.

Output 1 = ‘adult’ \(\Rightarrow\) age in \([18, 150)\)
Output 2 = ‘not adult’ \(\Rightarrow\) age in \([0, 18)\)
Output 3 = ‘invalid input’ \(\Rightarrow\) age not in \([0, 150]\)

Output 3 was not mentioned in the specification but it’s good practice to think about this possibility (programmers hopefully do!).
Also the maximum age was not mentioned in the spec; the tester would have to talk to the developers to find out whether there is an age limit implemented (e.g., as plausibility check).
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
- EC1: integer in [0, 18)
- EC2: integer in [18, 150]
- EC3: integer not in [0, 150] or not an integer

Output variable ‘adult’:
- EC4: true
- EC5: false

Output variable ‘error’:
- EC6: ‘invalid input’

This is a variable derived based on reasoning of the tester
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in [0, 18)
EC2: integer in [18, 150]
EC3: integer not in [0, 150] or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

These are ECs derived based on reasoning of the tester
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in \([0, 18)\)
EC2: integer in \([18, 150]\)
EC3: integer not in \([0, 150]\) or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

Test cases (minimum):

TC1: age = 10; adult = false; error = <empty>
TC2: age = 20; adult = true; error = <empty>
TC3: age = ‘x’; adult = <empty>; error = ‘invalid input’
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in [0, 18)
EC2: integer in [18, 150]
EC3: integer not in [0, 150] or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

Test cases (minimum):
TC1: age = 10; adult = false; error = <empty>
TC2: age = 20; adult = true; error = <empty>
TC3: age = ‘x’; adult = <empty>; error = ‘invalid input’
Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in [0, 18)
EC2: integer in [18, 150]
EC3: integer not in [0, 150] or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

Test cases (minimum):
TC1: age = 10; adult = false; error = <empty>
TC2: age = 20; adult = true; error = <empty>
TC3: age = ‘x’; adult = <empty>; error = ‘invalid input’
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in \([0, 18)\)
EC2: integer in \([18, 150]\)
EC3: integer not in \([0, 150]\) or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

Test cases (minimum):
TC1: age = 10; adult = false; error = <empty>
TC2: age = 20; adult = true; error = <empty>
TC3: age = ‘x’; adult = <empty>; error = ‘invalid input’
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in [0, 18)
EC2: integer in [18, 150]
EC3: integer not in [0, 150] or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

Coverage of ECs:

<table>
<thead>
<tr>
<th></th>
<th>EC1</th>
<th>EC2</th>
<th>EC3</th>
<th>EC4</th>
<th>EC5</th>
<th>EC6</th>
<th>EC7</th>
<th>EC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>TC2</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>TC3</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
ECP – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in [0, 18)
EC2: integer in [18, 150]
EC3: integer not in [0, 150] or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

Coverage of ECs:

<table>
<thead>
<tr>
<th></th>
<th>EC1</th>
<th>EC2</th>
<th>EC3</th>
<th>EC4</th>
<th>EC5</th>
<th>EC6</th>
<th>EC7</th>
<th>EC8</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>TC2</td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>TC3</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>
ECP Guidelines

Possible inputs/outputs must be known from specification or derived from exploration of the system.

If input is a range or an ordered list of values
→ one in-range/list and two out-of-range/list classes are defined
→ Example: x in [0, 9] → EC1: [0, 9], EC2: (-infinity, 0), EC3: (9, +infinity)

If input s a set or a “must be” condition (boolean)
→ one in-set and one out-of-set class are defined;
→ Example: EC1: {car}, EC2: empty (or: {horse})

If analysis of spec indicates that elements of input classes result in specific output classes (i.e., are treated equivalently) then additional classes may be defined.
→ Example: vehicle is in {car, motorcycle, truck} → EC1: {car}, EC2: {motorcycle}, EC3: {truck}, EC4: empty or anything not in the set.

If there is anything else (e.g., invalid inputs) → partition further
ECP Guidelines

Possible inputs/outputs must be known from specification or derived from exploration of the system.

If input is a range or an ordered list of values
→ one in-range/list and two out-of-range/list classes are defined
→ Example: x in [0, 9] → EC1: [0, 9], EC2: (-infinity, 0), EC3: (9, +infinity)

If input s a set or a “must be” condition (boolean)
→ one in-set and one out-of-set class are defined;
→ Example: EC1: {car}, EC2: empty (or: {horse})

If analysis of spec indicates that elements of input classes result in specific output classes (i.e., are treated equivalently) then additional classes may be defined.
→ Example: vehicle is in {car, motorcycle, truck} → EC1: {car}, EC2: {motorcycle}, EC3: {truck}, EC4: empty or anything not in the set.

If there is anything else (e.g., invalid inputs) → partition further
ECP in case of more than one input

This figure is a metaphor for the union set of ECs of all input variables.
From ECP to Test Cases
From ECP to Test Cases

Remember:
We must cover all ECs with Test Cases (TCs). We try to do this with as few TCs as possible!
From ECP to Test Cases

Step 1: Number each EC

Step 2: Apply the following rules ->

• Rule 1 – valid ECs: Combine as many valid ECs in one test case as possible

• Rule 2 – invalid ECs: Pick one invalid input EC and take a value from it; then combine it exclusively with values from valid input ECs

  • Reason: for each invalid EC there should be some dedicated error handling; this must be checked!

Remember:
We must cover all ECs with Test Cases (TCs). We try to do this with as few TCs as possible!
Boundary Value Analysis (BVA)

- Adds to the equivalence partitioning method
- Select test cases to represent each side of the class boundaries
BVA – Simple Example

Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in [0, 18)
EC2: integer in [18, 150]
EC3: integer not in [0, 150] or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

Test cases (from ECP):
TC1: age = 10; adult = false; error = <empty>
TC2: age = 20; adult = true; error = <empty>
TC3: age = ‘x’; adult = <empty>; error = ‘invalid input’
Summary of Equivalence Classes (ECs):

Input variable ‘age’:
EC1: integer in $[0, 18)$
EC2: integer in $[18, 150]$  
EC3: integer not in $[0, 150]$ or not an integer

Output variable ‘adult’:
EC4: true
EC5: false
EC6: <empty>

Output variable ‘error’:
EC7: <empty>
EC8: ‘invalid input’

Test cases (from ECP):
TC1: age = 10; adult = false; error = <empty>
TC2: age = 20; adult = true; error = <empty>
TC3: age = ‘x’; adult = <empty>; error = ‘invalid input’

New Test cases (from BVA):
TC4: age = 0; …
TC5: age = 18; …
RC6: age = 150; …
BVA Guidelines

• Range $a..b \Rightarrow a, b, \text{just above } a, \text{just below } b$

• List of ordinal values $\Rightarrow \text{max, min, just below min, just above max}$

• Boundaries of externally visible data structures shall be checked (e.g. ordered sets, arrays)

• Output bounds should be checked
Black Box Testing (BBT) – Wrap-Up

• BBT, also known as Behavioral Testing, is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester.
  • These tests can be functional or non-functional

Aims to detect these types of issues:
- Incorrect or missing functions
- (User) Interface problems
- Problems in data structures or external database access
- Behavior or performance problems
- Initialization and termination problems
BBT Techniques – Wrap-Up

Following are some techniques that can be used for designing black box tests:

- *Equivalence Class Partitioning*

- *Boundary Value Analysis*
BBT – Advantages & Disadvantages

Advantages
• Tests are done from a user’s point of view and will help in exposing discrepancies in the specifications.
• Tester need not know programming languages or how the software has been implemented.
• Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias.
• Test cases can be designed as soon as the specifications are complete.

Disadvantages
• Only a small number of possible inputs can be tested and many program paths will be left untested.
• Without clear specifications, which is the situation in many projects, test cases will be difficult to design.
• Tests can be redundant if the software designer/developer has already run a test case.
• Ever wondered why a soothsayer closes the eyes when foretelling events? So is almost the case in Black Box Testing.
Other BBT Methods

• Cause Effect Graphing (CEG)
• Combinatorial Testing
• Fuzzing
• Exploratory Testing
• Model-driven Testing
• …

Spring 2021: “Software Testing” course
Testing Strategies

Black Box Testing

White Box Testing
White Box Testing (WBT)

- WBT, also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing is a testing method the internal structure/design/implementation of the item being tested is known to the tester.
  - The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential.

Aims to detect these types of issues:
- Control flow issues
- Data flow issues
- Algorithms
How do Black-Box and White-Box Testing relate to one another?

- Develop an initial Test suite using BB techniques
- Apply BB coverage criteria to enhance it
- Analyze the parts of the code uncovered by BB test suite
- Enhance the Test suite using WB techniques
- Apply WB coverage criteria to enhance it
In-Class Exercise
Exercise: ECP

Assume a ’magic’ Function M

Spec:
The program accepts integers x, y
The program calculates sum = x + y
The program displays the result ‘sum’

M (x, y)  \rightarrow  \text{sum} = x + y
with x, y: int (32 bit)

Exhaustive (positive) testing: ?
Exercise: ECP

Assume a 'magic' Function M

Spec:
The program accepts integers x, y
The program calculates sum = x + y
The program displays the result ‘sum’

M (x, y) → sum = x + y
with x, y: int (32 bit)

Exhaustive (positive) testing:
$2^{32} \times 2^{32}$
$= 2^{64} \approx 1.8 \times 10^{19}$ test cases (input data + expected output)
Exercise: ECP

Input: x & y | Output: sum

Classes

C1: InputX: ?
C2: InputY: ?
C3: OutputSum: ?
Exercise: ECP

Input: x & y | Output: sum

Classes

C1: InputX: [MinInt, MaxInt]
C2: InputY: [MinInt, MaxInt]
C3: OutputSum: [MinInt, MaxInt]
Exercise: ECP

Input: x & y | Output: sum

Classes

C1: InputX: [MinInt, MaxInt]
C2: InputY: [MinInt, MaxInt]
C3: OutputSum: [MinInt, MaxInt]

Is that all?
Exercise: ECP

Input: x & y | Output: sum

Classes

C1: InputX: [MinInt, MaxInt]
C2: InputY: [MinInt, MaxInt]
C3: OutputSum: [MinInt, MaxInt]

C4: InputX: notInt
C5: InputY: notInt
C6: OutputSum: exception

Black Box

Input x:
Input y:
Output sum:
Exercise: ECP

Input: x & y | Output: sum

Classes

C1: InputX: [MinInt, MaxInt]
C2: InputY: [MinInt, MaxInt]
C3: OutputSum: [MinInt, MaxInt]

C4: InputX: notInt
C5: InputY: notInt
C6: OutputSum: exception
## Exercise: ECP

### Input: x & y | Output: sum

#### Classes

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>C1: InputX</td>
<td>[MinInt, MaxInt]</td>
</tr>
<tr>
<td>C2: InputY</td>
<td>[MinInt, MaxInt]</td>
</tr>
<tr>
<td>C3: OutputSum</td>
<td>[MinInt, MaxInt]</td>
</tr>
<tr>
<td>C4: InputX</td>
<td>notInt</td>
</tr>
<tr>
<td>C5: InputY</td>
<td>notInt</td>
</tr>
<tr>
<td>C6: OutputSum</td>
<td>exception</td>
</tr>
</tbody>
</table>

#### Test Cases

- **Data:** x, y, sum
- **TC1:** 0, 0 -> 0
- **TC2:** notInt, 0 -> WrongInputException
- **TC3:** 0, notInt -> WrongInputException

If we consider output=exception to be an error message caused by invalid input (notInt), then it’s good practice to check for the effect of each invalid input class independently.
Exercise: ECP

Input: x & y | Output: sum

Test Cases

Data: x, y, sum

TC1: 0, 0 -> 0
TC2: notInt, 0 -> WrongInputException
TC3: 0, notInt -> WrongInputException

Classes covered:

C1, C2, C3
C4, C2, C6
C1, C5, C6

minimal,
TCs cover all classes
Exercise: ECP + BVA

Input: x & y | Output: sum

Classes

C1: InputX: [MinInt, MaxInt]
C2: InputY: [MinInt, MaxInt]
C3: OutputSum: [MinInt, MaxInt]

C4: InputX: notInt
C5: InputY: notInt
C6: OutputSum: exception

Test Cases

Data: x, y, sum

TC1: 0, 0 -> 0
TC2: notInt, 0 -> WrongInputException
TC3: 0, notInt -> WrongInputException
TC4: MinInt, MinInt -> ArithmeticException
TC5: MaxInt, MaxInt -> ArithmeticException
TC6: MaxInt/2, MaxInt/2 -> MaxInt
TC7: MinInt/2, MinInt/2 -> MinInt
Example: WBT

Assume a 'magic' Function $M$

$M(x, y) \rightarrow \text{sum} = x + y$
with $x, y: \text{int (32 bit)}$

TC1: 0, 0 -> 0

White Box

... 

if ( $x - 100 \leq 0$ ) {
    if ( $y - 100 \leq 0$ ) {
        if ( $x + y - 200 == 0$ ) {
            crash(); }}
    print($x + y$);

How many statements covered with BBT suite?
Example: WBT

Assume a 'magic' Function $M$

$M(x, y) \rightarrow \text{sum} = x + y$

with $x, y: \text{int (32 bit)}$

TC1: 0, 0 -> 0

White Box

... 

if ( $x - 100 \leq 0$ ) {
    if ( $y - 100 \leq 0$ ) {
        if ( $x + y - 200 = 0$ ) {
            crash(); }}
    print($x + y$);

How many statements covered with BBT suite?

$\rightarrow 80\%$
Example: WBT

Assume a 'magic' Function $M$

$M(x, y) \rightarrow \text{sum} = x + y$

with $x, y: \text{int (32 bit)}$

White Box

```java
... if ( x - 100 <= 0 ) {
    if ( y - 100 <= 0 ) {
        if ( x + y - 200 == 0 ) {
            crash(); }
    }
}

print(x + y);
```

How many statements covered with BBT suite?

$\rightarrow 80\%$

TC1: 0, 0 -> 0
TC2: notInt, 0 -> WrongInputException
TC3: 0, notInt -> WrongInputException
TC4: MinInt, MinInt -> ArithmeticException
TC5: MaxInt, MaxInt -> ArithmeticException
TC6: MaxInt/2, MaxInt/2 -> MaxInt
TC7: MinInt/2, MinInt/2 -> MinInt
Example: WBT

Assume a ‘magic’ Function M

M (x, y) → sum = x + y
with x, y: int (32 bit)

Possible approaches:
- Execute each statement
- Execute paths based on:
  - Control-flow (decisions, conditions, loops, independent paths, etc.)
  - Data-flow (definition and usage of variables)
- Read (review) code

White Box
...
if ( x - 100 <= 0 ) {
  if ( y - 100 <= 0 ) {
    if ( x + y - 200 == 0 ) {
      crash();
    }
  }
}

print(x + y);

How many statements covered with BBT suite?
Control-Flow Testing: Statement Coverage

- If we try to cover all statements, we must find input data such that all three if-statements are 'true':

Traverse code and combine conditions:
(x<=100) and (y<=100) and (x+y=200) ->
(200-y<=100) and (y<=100) ->
(y>=100) and (y<=100) ->
y = 100 ->
x = 100

White Box

...  
if ( x - 100 <= 0 ) {
    if ( y - 100 <= 0 ) {
        if ( x + y - 200 == 0 ) {
            crash(); }}
    }
print(x + y);
Control-Flow Testing: Statement Coverage

Assume a 'magic' Function M

\[ M(x, y) \rightarrow \text{sum} = x + y \]
with \( x, y \): int (32 bit)

TC1: \( M(0, 0) \rightarrow 0 \)
TC2: \( M(100, 100) \rightarrow \text{crash}() \)

1st if = true: \( x \leq 100 \)
2nd if = true: \( y \leq 100 \)
3rd if = true: \( x + y = 200 \)

\( \rightarrow \) 100\% Statement Coverage

White Box

\[
\ldots
\text{if ( x - 100 } \leq 0 ) \{
\text{\quad if ( y - 100 } \leq 0 ) \{
\text{\quad \quad if ( x + y - 200 } == 0 ) \{
\text{\quad \quad \quad crash(); } \}}
\text{\quad \}}
\text{\}}
\text{print(x + y);}
\]
Control-Flow Testing: Branch Coverage

- If we try to cover all branches, we must find input data such that all three if-statements are once evaluated to 'true' and once to 'false':

TC1: M(0, 0) -> 0
TC2: M(100, 100) -> crash()

How much branch coverage?
Control-Flow Testing: Branch Coverage

Assume a 'magic' Function M

TC1: M(0, 0) -> 0
TC2: M(100, 100) -> crash()

-> 66% Branch (or Decision) Coverage

TC1: if1 = true / if2 = true / if3 = false
TC2: if1 = true / if2 = true / if3 = true

Missing: if1=false and if2=false -> 2 additional TCs needed
WBT – Advantages & Disadvantages

Advantages
• Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.
• Testing is more thorough, with the possibility of covering most paths.

Disadvantages
• Since tests can be very complex, highly skilled resources are required, with thorough knowledge of programming and implementation.
• Test script maintenance can be a burden if the implementation changes too frequently.
Other WBT Methods

• Data Flow Testing
• Mutation Testing
• Symbolic Execution
• ...

Spring 2021: “Software Testing” course
Other Testing Methods
Other Testing Methods

Gray Box Testing
• Combines BBT and WBT

Ad-hoc Testing (also known as Monkey Testing)
• A testing method without planning (and usually without documentation)
• The tests are conducted informally and randomly without any specified expected results
• Success relies highly on skills and creativity of the testers
• If automated → Random Testing (e.g., randoop)

Agile Testing
• Context: Test-Driven Development (TDD) and Behavior Driven Development (BDD)
• Exploratory Testing
TDD and BDD

• Developer TDD => Unit Tests

Acceptance TDD => Acceptance Tests
also called: Behavior-driven testing (BDD)
TDD

1. Write a test
2. See it fail
3. Make it run

Developer TDD

Combining Acceptance TDD/BDD and Developer TDD

Institute of Computer Science
University of Tartu

© Dietmar Pfahl 2020
TDD

1. Write a test
2. See it fail
3. Make it run
4. Make it right (refactor)
BDD

Acceptance TDD => Acceptance Tests
also called: Behavior-driven testing (BDD)
TDD and BDD

• Developer TDD => Unit Tests

Acceptance TDD => Acceptance Tests
also called:
Behavior-driven testing (BDD)
Exploratory Testing

• Not the same as ‘random testing’ or ‘monkey testing’
• Always worth doing (on top of “regular” / automated testing)
• Can trigger failures that systematic techniques miss
• Consider
  • ”What is the craziest thing we can do?”
  • Intuition / Experience / Brainstorming
  • Past failures / Lists in literature
• Tools
  • http://www.softwaretestinghelp.com/tools/top-17-exploratory-testing-tools/
Structure of Lecture 10

• Testing Basics
• Testing Levels
• Testing Methods
• Testing Types
• Testing Artefacts
• Metrics
Overview of Testing Types

• Smoke Testing
• Functional Testing
• Usability Testing
• Security Testing
• Performance Testing
• Regression Testing
• Compliance Testing
Smoke Testing

• Smoke Testing, also known as “Build Verification Testing”, is a type of software testing that covers most of the major functions of the software but none of them in depth.

• The results of this testing is used to decide if a build is stable enough to proceed with further testing.
  • If the smoke test passes, go ahead with further testing.
  • If it fails, halt further tests and ask for a new build with the required fixes.
  • If an application is badly broken, detailed testing might be a waste of time and effort.
Functional Testing

- Functional Testing is a type of software testing whereby the system is tested against the functional requirements/specifications.
- This type of testing is not concerned with how processing occurs, but rather, with the results of processing.

- Functional testing is normally performed during the levels of System and Acceptance Testing
- During functional testing, BBT techniques are used

- Typically, functional testing involves the following steps:
  1. Identify functions that the software is expected to perform
  2. Create input data based on the function’s specifications
  3. Determine the output based on the function’s specifications
  4. Execute the test case
  5. Compare the actual and expected outputs
Usability Testing

• Usability Testing is a type of testing done from an end-user’s perspective to determine if the system is easily usable

• Usability Testing is normally performed during System and Acceptance Testing levels

• Tips:
  • Understand who the users of the system are.
  • Understand what their business needs are.
  • Try to mimic their behavior.
Usability Test Types + Environment

Rubin’s Types of Usability Tests (Rubin, 1994, p. 31-46)

**Exploratory test** – early product development

**Assessment test** – most typical, either early or midway in the product development

**Validation test** – confirmation of product’s usability

**Comparison test** – compare two or more designs; can be used with other three types of tests

Think aloud

Audio/video recording

Observe
Usability Testing – Comparison A versus B
Usability Testing – Example of a Test Task

Let’s say a user needs to print a Financial Update Report, every 30 minutes, and he/she has to go through the following steps:

1. Login to the system
2. Click Reports
3. From the groups of reports, select Financial Reports
4. From the list of financial reports, select Financial Update Report
5. Specify the following parameters
   • Date Range, Time Zone, Departments, Units
6. Click Generate Report
7. Click Print
8. Select an option
   • Print as PDF
   • Print for Real

Good or Bad Usability?
Testing Usability Requirements

How to test:
- Define several (typical) usage scenarios involving tasks Q and R
- Select test users and classify as 'novice' and 'experienced'
- Let 5 (or better 10, 15) novices perform the scenarios
- Observe what problems they encounter
- Classify and count observed problems

<table>
<thead>
<tr>
<th>Problem counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1: At most 1 of 5 novices shall encounter critical problems during tasks Q and R. At most 5 medium problems on list.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task time</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2: Novice users shall perform tasks Q and R in 15 minutes. Experienced users tasks Q, R, S in 2 minutes.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Keystroke counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3: Recording breakfast shall be possible with 5 keystrokes per guest. No mouse.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opinion poll</th>
</tr>
</thead>
<tbody>
<tr>
<td>R4: 80% of users shall find system easy to learn. 60% shall recommend system to others.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Score for understanding</th>
</tr>
</thead>
<tbody>
<tr>
<td>R5: Show 5 users 10 common error messages, e.g. Amount too large. Ask for the cause. 80% of the answers shall be correct.</td>
</tr>
</tbody>
</table>
Usability Testing – What? How?

- Test Focus
  - Understandability
    - Easy to understand?
  - Ease of learning
    - Easy to learn?
  - Operability
    - Matches purpose & environment of operation?
    - Ergonomics: color, font, sound,...
  - Communicativeness
    - In accordance with psychological characteristics of user?

- Test Environments
  - Free form tasks
  - Procedure scripts
  - Paper screens
  - Mock-ups
  - Field trial
Heuristic Evaluation by Inspection

List of 10 Heuristics according to (Nielsen, 2005):

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Visibility of system status</td>
</tr>
<tr>
<td>2</td>
<td>Match between the system and the real world</td>
</tr>
<tr>
<td>3</td>
<td>User control and freedom</td>
</tr>
<tr>
<td>4</td>
<td>Consistency and standards</td>
</tr>
<tr>
<td>5</td>
<td>Error prevention</td>
</tr>
<tr>
<td>6</td>
<td>Recognition rather than recall</td>
</tr>
<tr>
<td>7</td>
<td>Flexibility and efficiency of use</td>
</tr>
<tr>
<td>8</td>
<td>Aesthetic and minimalist design</td>
</tr>
<tr>
<td>9</td>
<td>Help users recognize, diagnose, and recover from errors</td>
</tr>
<tr>
<td>10</td>
<td>Help and documentation</td>
</tr>
</tbody>
</table>

List violations of heuristics:

Rank by severity: 0...4

0: positive (or neutral) aspect of system

4: major, catastrophic aspect of system

---


Security Testing

• Security Testing is a type of software testing that intends to uncover vulnerabilities of the system and determine that its data and resources are protected from possible intruders.

• 4 Focus Areas:
  • Network security: This involves looking for vulnerabilities in the network infrastructure (resources and policies).
  • System software security: This involves assessing weaknesses in the various software (operating system, database system, and other software) the application depends on.
  • Client-side application security: This deals with ensuring that the client (browser or any such tool) cannot be manipulated.
  • Server-side application security: This involves making sure that the server code and its technologies are robust enough to fend off any intrusion.
Security Testing

Example of a basic security test:
• Log into the web application.
• Log out of the web application.
• Click the BACK button of the browser, then check if you are asked to log in again or if you are provided the logged-in application.

• Most types of security testing involve complex steps and out-of-the-box thinking but, sometimes, it is simple tests like the one above that help expose the most severe security risks.
• The Open Web Application Security Project (OWASP) is a great resource for software security professionals. Be sure to check out the Testing Guide: https://www.owasp.org/index.php/Category:OWASP_Testing_Project

OWASP Top 10 security threats for 2013 were:
• Injection
• Broken Authentication and Session Management
• Cross-Site Scripting (XSS)
• Insecure Direct Object References
• Security Misconfiguration
• Sensitive Data Exposure
• Missing Function Level Access Control
• Cross-Site Request Forgery (CSRF)
• Using Known Vulnerable Components
• Unvalidated Redirects and Forwards
<table>
<thead>
<tr>
<th>OWASP Top 10 - 2013</th>
<th>OWASP Top 10 - 2017</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2 – Broken Authentication and Session Management</td>
<td>A2:2017-Broken Authentication</td>
</tr>
<tr>
<td>A3 – Cross-Site Scripting (XSS)</td>
<td>A3:2017-Sensitive Data Exposure</td>
</tr>
<tr>
<td>A6 – Sensitive Data Exposure</td>
<td>A6:2017-Security Misconfiguration</td>
</tr>
<tr>
<td>A8 – Cross-Site Request Forgery (CSRF)</td>
<td>A8:2017-Insecure Deserialization [NEW, Community]</td>
</tr>
<tr>
<td>A9 – Using Components with Known Vulnerabilities</td>
<td>A9:2017-Using Components with Known Vulnerabilities</td>
</tr>
</tbody>
</table>
How to avoid SQL injection vulnerability?

Instead of:

```java
String query = "SELECT * FROM Users WHERE Username= "
+ request.getParameter("username")
+ " AND Password= "
+ request.getParameter("password");

try {
    Statement statement = connection.createStatement();
    ResultSet results = statement.executeQuery(query);
}
```

Which might result in a SQL query string like this:

```
SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND
            Password='1' OR '1' = '1'
```
How to avoid SQL injection vulnerability?

Use java ‘prepared statement’:

```java
String username = request.getParameter("username");
String password = request.getParameter("password");
// perform input validation to detect attacks

String query = "SELECT * FROM Users WHERE Username= ? AND Password= ?";
PreparedStatement pstmt = connection.prepareStatement(query);
pstmt.setString( 1, username);
pstmt.setString( 2, password);
ResultSet results = pstmt.executeQuery( );
```

Example with Hibernate Query Language (HQL) can be found here: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
Performance Testing

• Performance Testing is a type of software testing that intends to determine how a system performs in terms of responsiveness and stability under a certain load.

Types:
• **Load Testing** is a type of performance testing conducted to evaluate the behavior of a system at increasing workload.
• **Stress Testing** a type of performance testing conducted to evaluate the behavior of a system at or beyond the limits of its anticipated workload.
• **Endurance Testing** is a type of performance testing conducted to evaluate the behavior of a system when a significant workload is given continuously.
• **Spike Testing** is a type of performance testing conducted to evaluate the behavior of a system when the load is suddenly and substantially increased.
Regression Testing

- Regression testing is a type of software testing that intends to ensure that changes (enhancements or defect fixes) to the software have not adversely affected it.

How much?
- In an ideal case, a full regression test is desirable but oftentimes there are time/resource constraints. In such cases, it is essential to do an impact analysis of the changes to identify areas of the software that have the highest probability of being affected by the change and that have the highest impact to users in case of malfunction and focus testing around those areas.
Regression Testing – Retest All

• Assumption:
  • Changes may introduce faults anywhere in the code

• BUT: expensive, prohibitive for large systems

• Reuse existing test suite
• Add new tests as needed
• Remove obsolete tests

[Skoglund, Runeson, ISESE05]
Regression Testing – Selective Testing

- Conduct impact analysis
  - Only code impacted by change needs to be retested
  - Select tests that exercise such code
- Add new tests if needed
- Remove obsolete tests
Compliance Testing

- Compliance Testing, also known as conformance testing, regulation testing, standards testing, is a type of testing to determine the compliance of a system with internal or external standards.

Checklist:

<table>
<thead>
<tr>
<th>Lorem ipsum dolor sit amet</th>
<th>✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>consectetur adipiscing elit</td>
<td>✓</td>
</tr>
<tr>
<td>sed do eiusmod tempor incididunt ut labore et dolore magna aliqua</td>
<td>✓</td>
</tr>
<tr>
<td>Ut enim ad minim veniam</td>
<td>X</td>
</tr>
<tr>
<td>quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat</td>
<td>✓</td>
</tr>
</tbody>
</table>
Structure of Lecture 10

• Testing Basics
• Testing Levels
• Testing Methods
• Testing Types
• Testing Artefacts
• Metrics
Test Documentation

IEEE 829-2008: Standard for Software and System Test Documentation

FIG. 7.4
Test Plan

• A Software Test Plan is a document describing the testing scope and activities. It is the basis for formally testing any software/product in a project.

One can have the following types of test plans:
• **Master Test Plan:** A single high-level test plan for a project/product that unifies all other test plans.
• **Testing Level Specific Test Plans:**
  • Unit Test Plan
  • Integration Test Plan
  • System Test Plan
  • Acceptance Test Plan
• **Testing Type Specific Test Plans:** Plans for major types of testing like Performance Test Plan and Security Test Plan
## Test Case – Template

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Suite ID</td>
<td>The ID of the test suite to which this test case belongs.</td>
</tr>
<tr>
<td>Test Case ID</td>
<td>The ID of the test case.</td>
</tr>
<tr>
<td>Test Case Summary</td>
<td>The summary / objective of the test case.</td>
</tr>
<tr>
<td>Related Requirement</td>
<td>The ID of the requirement this test case relates/traces to.</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>Any prerequisites or preconditions that must be fulfilled prior to executing the test.</td>
</tr>
<tr>
<td>Test Procedure</td>
<td>Step-by-step procedure to execute the test.</td>
</tr>
<tr>
<td>Test Data</td>
<td>The test data, or links to the test data, that are to be used while conducting the test.</td>
</tr>
<tr>
<td>Expected Result</td>
<td>The expected result of the test.</td>
</tr>
<tr>
<td>Actual Result</td>
<td>The actual result of the test; to be filled after executing the test.</td>
</tr>
<tr>
<td>Status</td>
<td>Pass or Fail. Other statuses can be ‘Not Executed’ if testing is not performed and ‘Blocked’ if testing is blocked.</td>
</tr>
<tr>
<td>Remarks</td>
<td>Any comments on the test case or test execution.</td>
</tr>
<tr>
<td>Created By</td>
<td>The name of the author of the test case.</td>
</tr>
<tr>
<td>Date of Creation</td>
<td>The date of creation of the test case.</td>
</tr>
<tr>
<td>Executed By</td>
<td>The name of the person who executed the test.</td>
</tr>
<tr>
<td>Date of Execution</td>
<td>The date of execution of the test.</td>
</tr>
<tr>
<td>Test Environment</td>
<td>The environment (Hardware/Software/Network) in which the test was executed.</td>
</tr>
</tbody>
</table>
## Test Case – Example

<table>
<thead>
<tr>
<th>Test Suite ID</th>
<th>TS001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case ID</td>
<td>TC001</td>
</tr>
<tr>
<td>Test Case Summary</td>
<td>To verify that clicking the Generate Coin button generates coins.</td>
</tr>
<tr>
<td>Related Requirement</td>
<td>RS001</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>User is authorized; Coin balance is available.</td>
</tr>
<tr>
<td>Test Procedure</td>
<td>Select the coin denomination in the Denomination field. Enter the number of coins in the Quantity field. Click Generate Coin.</td>
</tr>
<tr>
<td>Test Data</td>
<td>Denominations: 0.05, 0.10, 0.25, 0.50, 1, 2, 5 Quantities: 0, 1, 5, 10, 20</td>
</tr>
<tr>
<td>Expected Result</td>
<td>Coin of the specified denomination should be produced if the specified Quantity is valid (1, 5) A message ‘Please enter a valid quantity between 1 and 10’ should be displayed if the specified quantity is invalid.</td>
</tr>
<tr>
<td>Actual Result</td>
<td>If the specified quantity is valid, the result is as expected. If the specified quantity is invalid, nothing happens; the expected message is not displayed</td>
</tr>
<tr>
<td>Status</td>
<td>Fail</td>
</tr>
<tr>
<td>Remarks</td>
<td>This is a sample test case.</td>
</tr>
<tr>
<td>Created By</td>
<td>John Doe</td>
</tr>
<tr>
<td>Date of Creation</td>
<td>01/14/2020</td>
</tr>
<tr>
<td>Executed By</td>
<td>Jane Roe</td>
</tr>
<tr>
<td>Date of Execution</td>
<td>02/16/2020</td>
</tr>
<tr>
<td>Test Environment</td>
<td>OS: Windows Y, Browser: Chrome N</td>
</tr>
</tbody>
</table>
Incident/Issue Report

- In most companies, an issue reporting tool is used and the elements of a report can vary. However, in general, an issue report can consist of the following elements.
Structure of Lecture 10

- Testing Basics
- Testing Levels
- Testing Methods
- Testing Types
- Testing Artefacts
- Metrics
Code Coverage

- The relative amount of covered items with regards to a coverage criterion, e.g., statement, branch, condition, ...

\[
\text{statement\_coverage} = \frac{\text{executed\_statements}}{\text{total\_number\_statements}}
\]

\[
\text{branch\_coverage} = \frac{\text{executed\_branches}}{\text{total\_number\_branches}}
\]

Usage
- To control the comprehensiveness of a test suite
- Often used as test stopping criterion
Defect Density

- Defect Density is the number of confirmed defects detected in software/component during a defined period of development/operation divided by the size of the software/component.

\[ \text{defect\_density} = \frac{\text{number\_confirmed\_defects}}{\text{size}} \]

Usage:
- For comparing the relative number of defects in various software components (or software products) so that high-risk components can be identified and resources focused towards them.
Defect Coverage

- Defect Coverage is the number of confirmed defects detected in software/component during a defined period of development/operation divided by the total number of defects.

\[
\text{defect\_coverage} = \frac{\text{number\_confirmed\_defects}}{\text{total\_number\_defects}}
\]

Usage:
- To assess the effectiveness of a test suite.
- Might be applied for certain types of defects (e.g., severity=major; priority=high)
Next Lecture

• Date/Time:
  • Friday, 13-Nov, 10:15-12:00
• Topic:
  • Refactoring (and TDD)
• Labs:
  • Continue working on Homework 5
  • Go to Assessment Labs next week!