Software Engineering
Week 11: Programming

Lydie du Bousquet

Software engineering is not the same as programming

- Every software engineer knows how to program, but **not** every programmer is a software engineer.
- **Software engineering** is typically
  - a **group** effort,
  - with differing and often fluid roles and responsibilities
- Engineers develop software
  - **to meet specifications** set for their client, and
  - generally must adhere to specific standards and practices.
- Engineering projects have
  - **timelines, release dates**, and
  - considerable **interaction** between people responsible for various components.

http://www.wikihow.com/Become-a-Software-Engineer

Programming

- **Deadlines** and **realease dates**
  - Be efficient
  - Know tools you use
  - Program efficiently

- **Group effort** and **interaction**
  - **Code esthetic**
    - Working: means correct, meets specification
    - Easy to understand: well structured & documented
  - **Agreed upon specification**
    - Changing the spec is a team decision

Programming

- Use languages **properly**:
  Procedural, OOP, …

Make the right choices

- Use **Patterns**

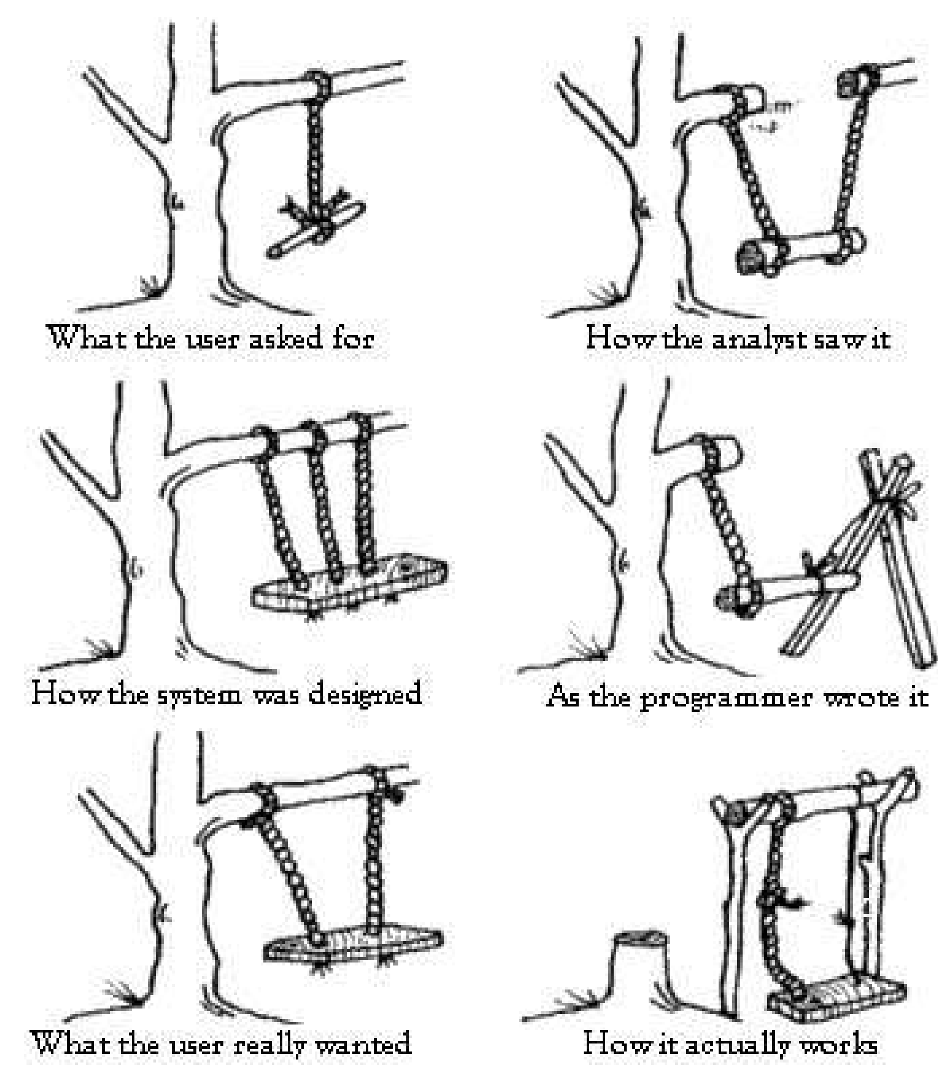
- Recognize the **programming style**
  eg. threads vs events

Programming is a craft

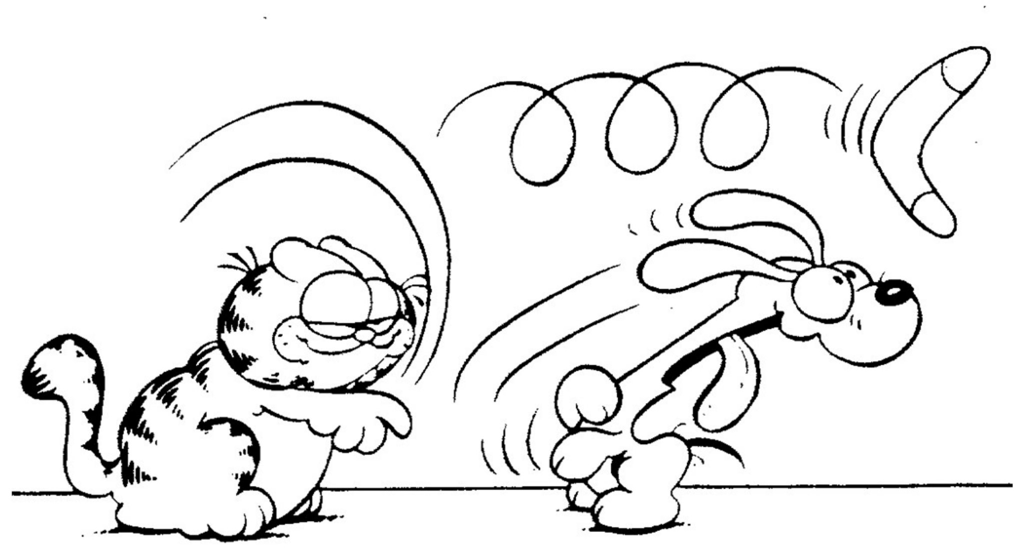
It needs practicing!

Software engineering is also a craft

- **Understand** the problems … and solutions
- **Apply** them during projects



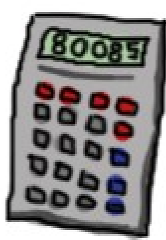
Software Engineering
Week 11: Testing



Code should be « **working** » not only « **running** »

- OK. Let’s test a program !

- Exercise 1
  Test the IntegerDivision Program



Exercise 1: test the program

```
public class IntegerDivision {
  public static int IntDiv(int x, int y) {
   int z = 0;
   int signe = 1;
   if (x < 0) {
     signe = -1;
     x = -x; }
   if (y < 0) {
     signe = -signe;
     y = -y;}
   if (y == 0) {
    throw new IllegalArgumentException("Arg nul:"+y);}
   while (x >= y) {
     x = x - y;
     z = z + 1;
   }
   z = signe * z;
   return z;
  }
}
```

Code should be « **working** » not only « **running** »

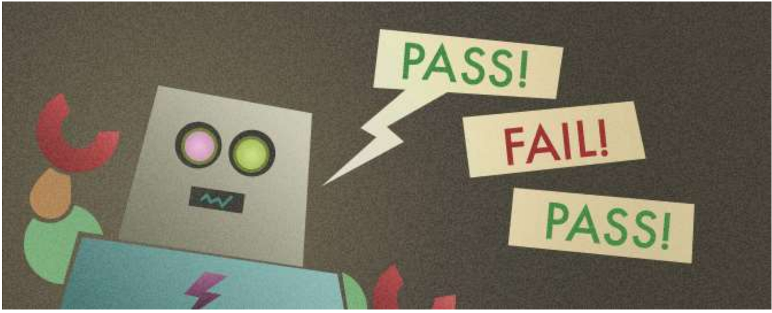
1. What is testing? What does it mean?
2. Why do you test?

Take a pen and write your answers

Definition of Testing

- Testing is the process of **executing** the software (system) **in order to find bugs**

- It consists in
  - Identifying **relevant input** data (to find bugs)
  - **Executing** the Software with the data
  - Observing and **judging** outputs



Developping vs Testing

Developping

Testing

Developping vs Testing

- It is **difficult** to test its **own** programs
- It was difficult to build them (usually)
  - You don’t want to destroy what you have built
  - You will avoid the critical points

- Use methods to select data systematically

Code should be « **working** » not only « **running** »

3. How did you select input data for ex1?
How can we do in general?

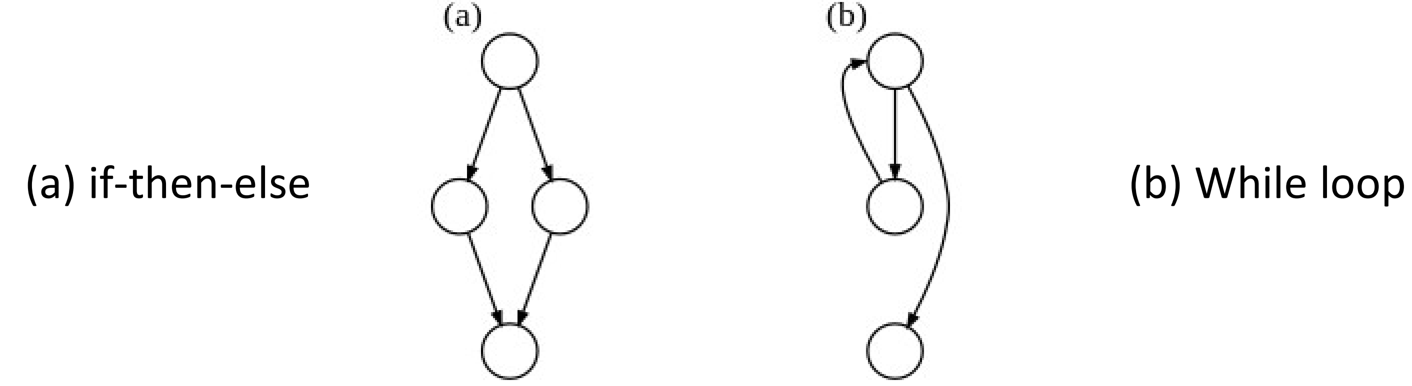
Take a pen and write your answers

2. Selecting test data

- From the **code structure** vs from the **specification**
- On purpose or at random
- To check
  - Normal behaviors
  - Limits,
  - Outside the limits
- Functional and non-functional

Control-flow graph

- **Representation** of **all** paths that **might** be traversed through a program during its **execution**



"Some types of control flow graphs" by JMP EAX - Own work. Licensed under CC BY-SA 4.0 via Commons – https://commons.wikimedia.org/wiki/File:Some_types_of_control_flow_graphs.svg

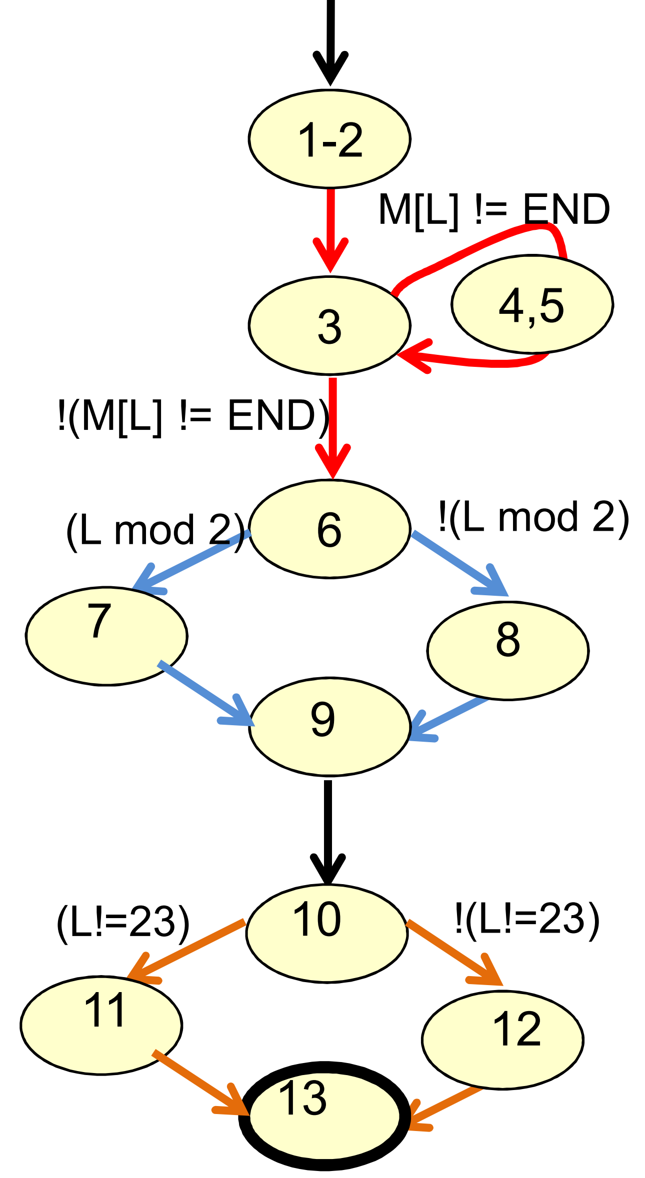
Example : is the size of a word even ?

```
1.    M <- ReadWord
2.    L = 0
3.    while  M[L] != END do
4.        L = L + 1
5.    end
6.    if (L mod 2) = 0
7.    then print("L is even")
8.    Else print("L is odd")
9.    Print(".")
10.   if (L!=23)
11.   then print(".")
12.   else print( "..")
13.   end
```

What is the control-flow graph of this program?

Example : is the size of a word even ?

```
1.    M <- ReadWord
2.    L = 0
3.    while  M[L] != END do
4.        L = L + 1
5.    end
6.    if (L mod 2) = 0
7.    then print("L is even")
8.    Else print("L is odd")
9.    Print(".")
10.   if (L!=23)
11.   then print(".")
12.   else print( "..")
13.   end
```



Exercise 2

- For the “word even program” propose test data to achieve:
  - 100% line coverage
  - 100% condition coverage
  - 100% path coverage
- What is the difference between line and condition coverage ?

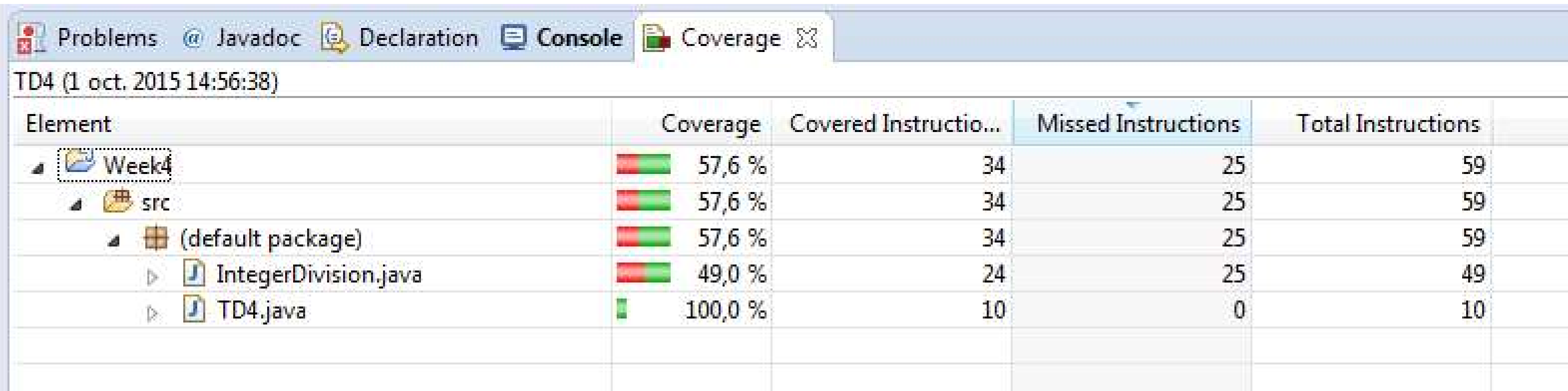
Example : is the size of a word even ?

```
1.    M <- ReadWord
2.    L = 0
3.    while  M[L] != END do
4.        L = L + 1
5.    end
6.    if (L mod 2) = 0
7.    then print("L is even")
8.    Else print("L is odd")
9.    Print(".")
10.   if (L!=23)
11.   then print(".")
12.   end
```

What is the control-flow graph of this program?

EclEmma

- Code coverage tool
  Show which statements have been executed
- Eclipse plug-in
- Adds a so called *launch mode*



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
Week4	57,6 %	34	25	59
src	57,6 %	34	25	59
(default package)	57,6 %	34	25	59
IntegerDivision.java	49,0 %	24	25	49
TD4.java	100,0 %	10	0	10

EclEmma

- **Line coverage**
  - green for fully covered lines
  - yellow for partly covered lines
  - red for lines that have not been executed at all
- **Decision branches**
  - ♦ green for fully covered branches,
  - ♦ yellow for partly covered branches and
  - ♦ red when no branches in the particular line have been executed.

Experiment at home

1. Install Junit and EclEmma,
2. Construct a JUnit test file for `IntegerDivision.java`
3. Execute it with EclEmma
4. Answer the following questions

   Q1. What do you observe?

   Q2. What does it means?

   Q3. Achieve 100% line coverage

   Q4. Achieve 100% condition coverage

   Q5. Fell the difference between line and condition coverage

Selecting test data

- *From the **code structure** vs*
  from the **specification**
- On purpose or at random
- To check
  - Normal behaviors
  - Limits,
  - Outside the limits
- Functional and non-functional

Input Space Partition testing

- Identify **input partition** in the input set from the specification
  - Valid values
  - Boundaries
  - Normal uses
  - And (if relevant) invalid values & extreme uses
- Choose at least **one input data** in each equivalence partition

Combination strategies criteria

F(x, y, z)

$x \in \{a,b,c\}$ $y \in \{1,2,3\}$ $z \in \{true, false\}$

- **All combination coverage**
  - 3*3*2
- **Each choice coverage**
  - (a,1,true) (b,2,false) (c,3,true)
- **Pairwise coverage**

Combination strategies criteria

Pairwise coverage

- Pairs to be covered
  - (a,1) (a,2) (a,3) (b,1) (b,2) (b,3) (c,1) (c,2) (c,3)
  - (a,t) (a,f) (b,t) (b,f) (c,t) (c,f)
  - (1,t) (1,f) (2,t) (2,f) (3,t) (3,f)
- Possible tests

  (a,1,t) (a,2,f) (a,3,t)
  (b,1,f) (b,2,t) (b,3,f)
  (c,1,t) (c,2,f) (c,3,t)

9 tests instead of 18
With all-combinations

Partitions and combinations

Applying **partition** and **combination** strategies, means applying **hypotheses**

All behaviours in a partition are « **equivalent** » w.r.t. finding errors

The chosen combination type is « **adequat** » to find errors

THINKING CAP

Whoa.

Do it on purpose!

Exercise 3

Q1. Apply the input partition technique and all-combination strategy to select tests for the IntegerDivision program

Q2. Complete your tests

Q3. What can you say about your tests now?
Is the program free of bugs? Why?

Code should be « **working** » not only « **running** »

4. How can I be convinced that my tests are good enough to find bugs?

Take a pen and write your answers

Fault injection and mutation testing

- A test is good if it is **able** to find bug
- Mutation testing introduces systematically elementary fault in the code
  - Long
  - Tedious
- Inject some faults to evaluate your tests (and remove them after!)

Tests might not find bugs or find bugs that are *not* bugs

2 possible reasons

- Test data are not well-chosen
- Test oracle is not correct

Oracle: mechanism for determining whether a test has **passed** or **failed** (assert in Junit)

This is **not** a test... Why?

```
@Test
public void test1() {
  int res= IntegerDivision.IntDiv(4, 2);
}
```

This is a (very simple) test

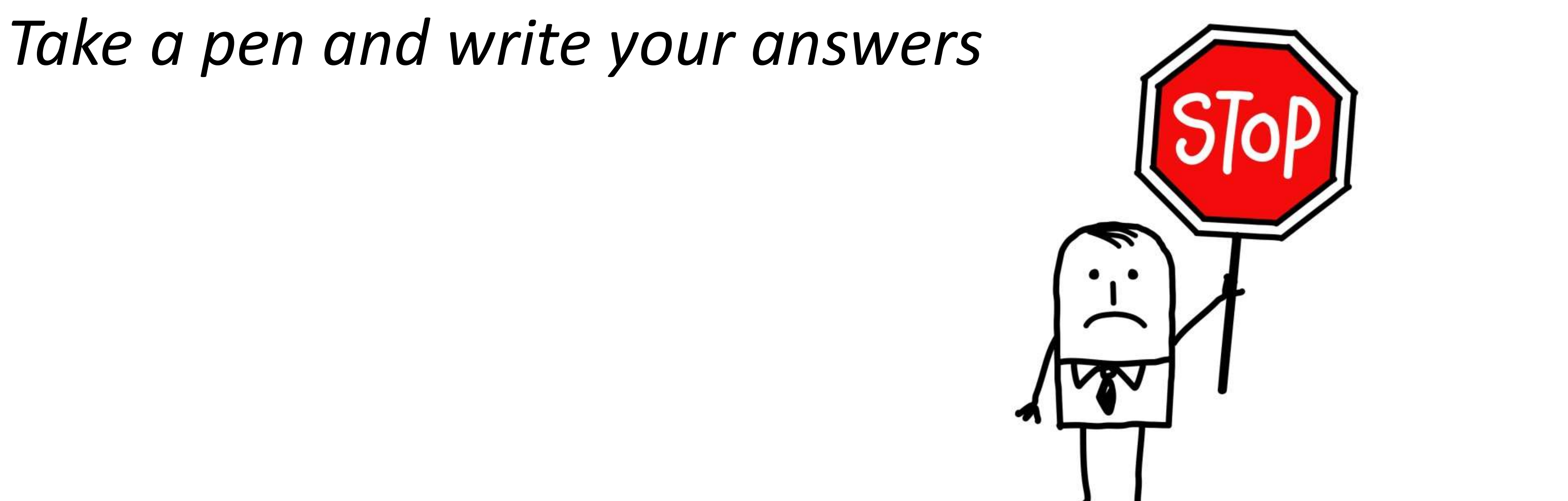
```
@Test
public void test1() {
  int res= IntegerDivision.IntDiv(4, 2);
  assertEquals(res, 2);

}
```

« Oracle »

Code should be « **working** » not only « **running** »

5. When should testing stop?

Take a pen and write your answers



When should testing stop?

- One of the **most difficult** questions to a tester

- Test is to reveal failures
- Test everything is impossible

- Need to choose a **compromise** between
  - Test many behaviours but it is expensive
  - Test too few behaviours and failing finding errors

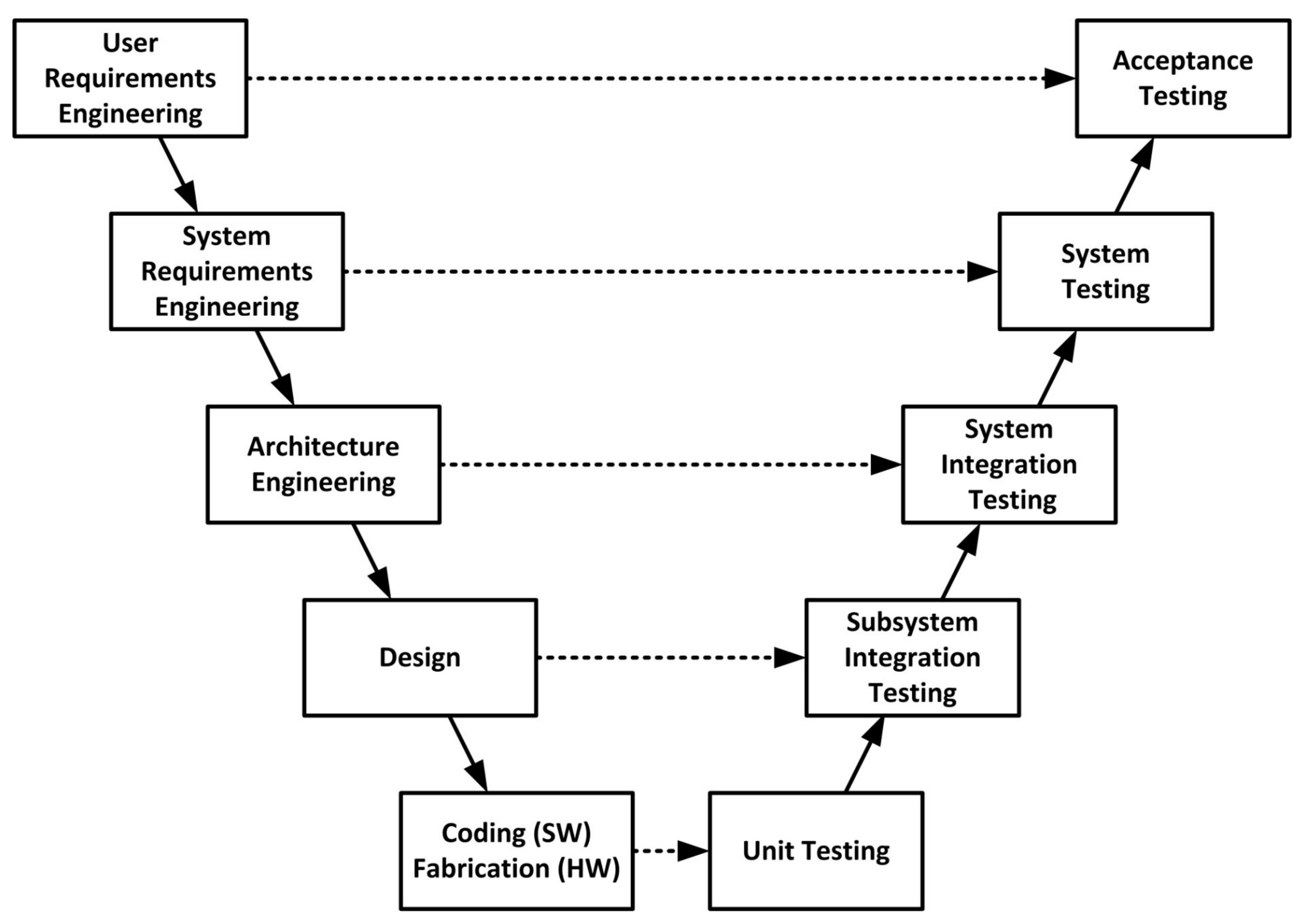
When should testing stop?

- Few of the common Test Stop criteria
  1. All the high priority bugs are fixed
  2. The **rate** at which bugs are found is too small
  3. The testing **budget** is exhausted
  4. The project **duration** is completed
  5. The **risk** in the project is under acceptable limit.

Practical considerations

- Testing levels
- Testing purpose
- Test automation
- Test code esthetics
- Test and validation

Testing levels



Testing purposes

- Functional properties
- Non-functional properties
  - Usability
  - Robustness
  - Reliability
  - Efficiency
  - Portability, Compatibility
  - Load, Performance, Efficiency
  - Security
  - …

Manual or automated tests?

- Automate testing has a cost
- It should be useful
  - Continuous integration
  - Regression testing
  - …
- It is not a guarantee that all bugs will be found

Test code esthetic

- Test programs are also programs!
  - Shared, maintained, have to evolve
  - Need to be commented
- Given-When-Then style
  - template intended to guide the writing of acceptance tests for a User Story
  - Idea beyond
    - (Given) some context
    - (When) some action is carried out
    - (Then) a particular set of observable consequences should obtain

Test vs Validation

- Validation is more general than test
- Process of evaluating a system during or at the end of the development process to determine whether it satisfies specified requirements [IEEE]
- Validation may consist in
  - Code review
  - Static analysis
  - Verification
  - Testing

Competence and Knowledge which will be evaluated

- Be able to
  - draw the **control-flow graph** of a simple program
  - select data to **achieve line/branch coverage**
  - ~~use **input partition** method and **combination strategies**~~
- know
  - the testing **philosophy** and
  - ~~**practical considerations**~~