Introduction to Software Testing

https://www.youtube.com/playlist?list=PLF3-CvSRq2SaApl3Lnu6Tu_ecsBr94543

In the 1st lesson you learned the basics of software analysis.

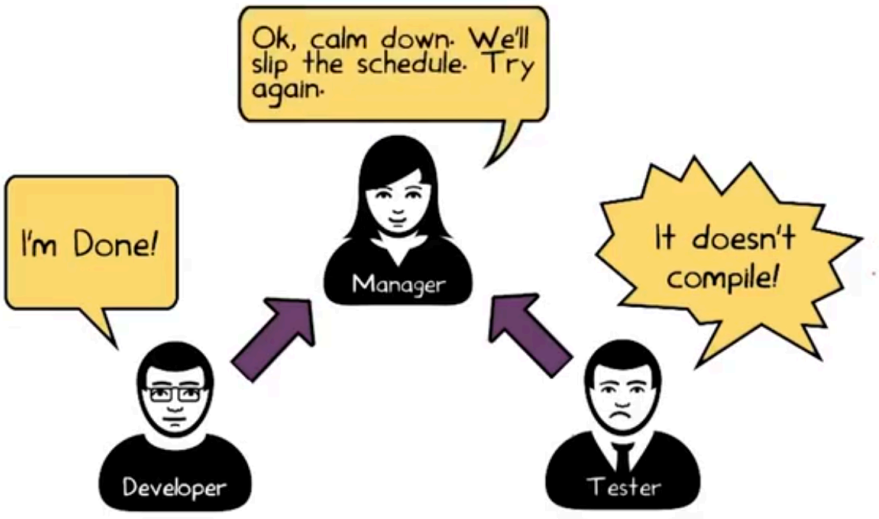
However, analysis is only one half of this course. The other half is software testing.

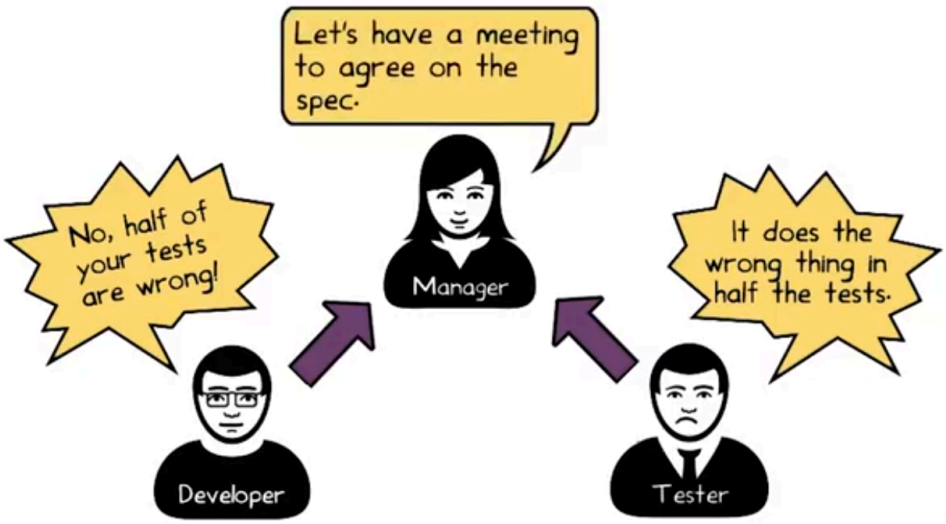
Software testing -> the process of checking the correctness

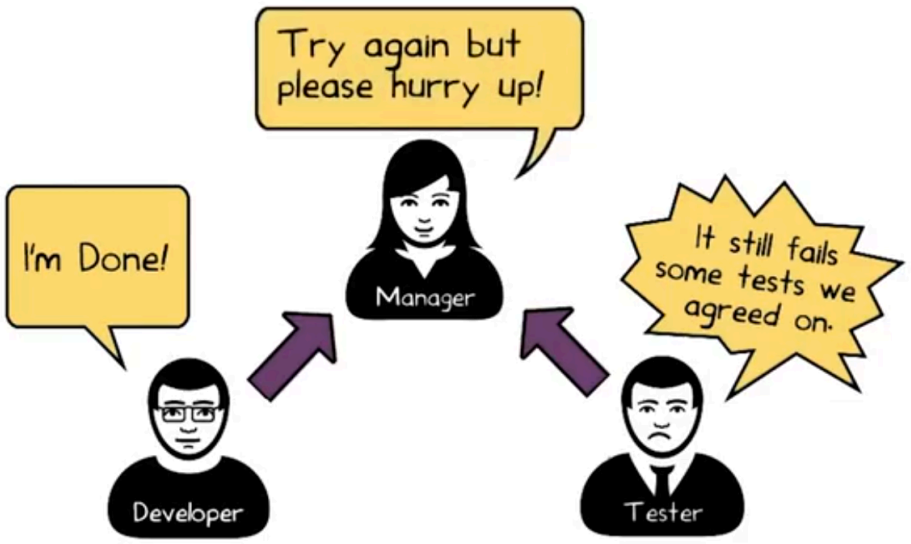
of a given piece of software.

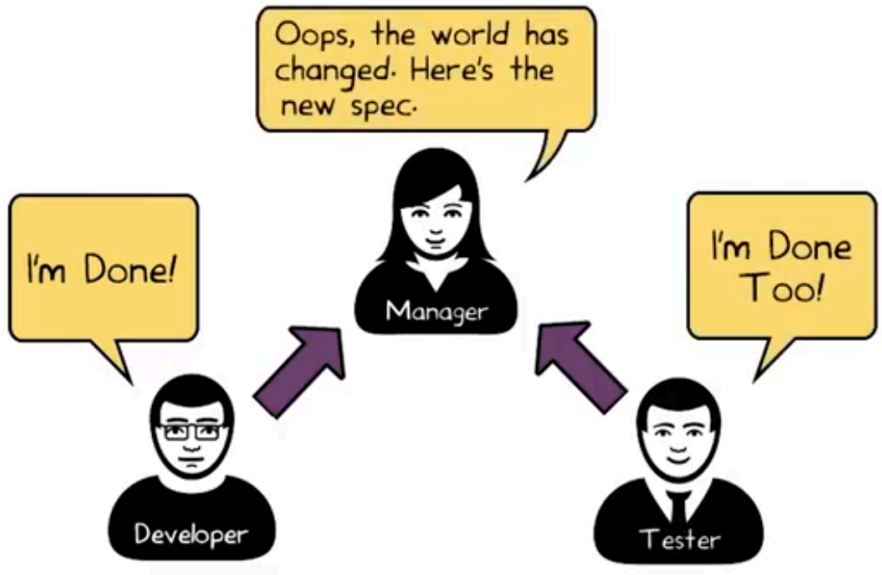
Software development today

- A team typically consists of at least one developer, at least one tester, and a manager to whom both the developers and the testers report.
- Typical scenario:









Key observations

- Specifications must be explicit
- Independent development and testing
- Resources are finite
- Specifications evolve over time

The need for specifications

- Testing checks whether program implementation agrees with program specification
- Without a specification there is nothing to test!
- Testing a form of **consistency checking** between **implementation** and **specification**
  - Recurring theme for software quality checking approaches
  - **What if both implementation and specification are wrong?**

Developer ! = Tester

- Developer writes implementation, tester writes specification
- Unlikely that both will independently make the same mistake
- Specifications useful even if written by developer itself
  - Much simpler than implementation
  - Specification unlikely to have same mistake as the implementation

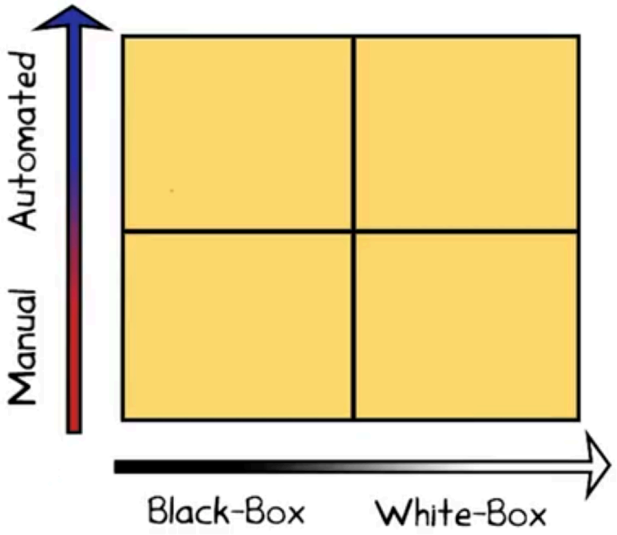
Other observations

- Resources are finite
  - Limit how many tests are written
- Specifications evolve over time
  - Tests must be updated over time
- An idea: **Automated testing**
  - **No need for testers?**

Outline

- Landcape of testing
- Specifications
  - Pre- and post-conditions
- Measuring test suite quality
  - Coverage metrics
  - Mutation analysis

Clasification of testing approaches



- **Black-box testing** refers to testing where the tester can see nothing about the tested programs internal mechanisms. As though the program is contained inside an opaque box.
  - The tester can only issue inputs to the program, observe the program's outputs, and determine whether the observed outputs meet the specifications required of the program.
- **White-box testing** refers to testing in which the internal details of the program being tested are fully available to the tester.
  - the tester can use these internal details to perform a more precise analysis of the tested program and uncover inputs that are more likely to trigger buggy behavior.
- **Testing approaches need not be strictly black-box or white-box**.
  Some internal details may be available to the tester, while others remain hidden.
  These sorts of testing approaches are called **gray box approaches**.
- Similarly, **testing approaches need not be fully manual or fully automated**.
- Examples
  - Exercise different GUI events of an Android app. This sort of testing would fall somewhere near the **bottom left of above diagram** because the tester is only issuing commands to and observing outputs from the program under testing.
  - If the tester looks at the source code, in order to determine what other routes there are through the app's GUI, then we have taken the original approach and we move it **rightward to the white-box side of the diagram**.
  - Instead of manually activating GUI events for the Android app, we might use an automated approach such as a fuzzer to automatically issue tap commands to random coordinates of the smartphone. This takes the original **black-box approach** and **moves** it **upward to the automated side of the diagram**.
  - We might also take approaches such as feedback directed random testing which issues random commands that change in response to the feedback issued by the GUI. Or we may **perform symbolic execution** that needs to inspect the source code, in order to test effectively, **that is, perform static analysis**. Or we may even monitor the code as it is being tested, that is, perform dynamic analysis in order to discover future tests intelligently.
    These approaches take us to the **top right quadrant of the diagram**, in which we are performing automated white box testing program.

Automated testing vs. manual testing

Automated testing

- Find bugs more quickly
- No need to write tests
- If software changes, no need to maintain tests

Manual testing

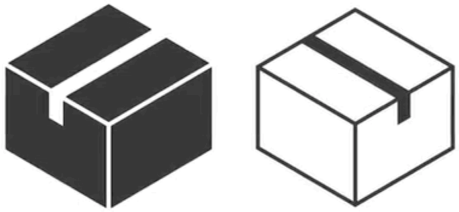
- Efficient test suite
- Potentially better coverage

The **ideal approach** will often lie in **the combination of automated and manual approaches**.

Black-box vs. white-box testing

Black-box testing

- Can work with code that cannot be modified
- Does not need to analyse or study code
- Code can be in any format (managed, binary, obfuscated)



White-box testing

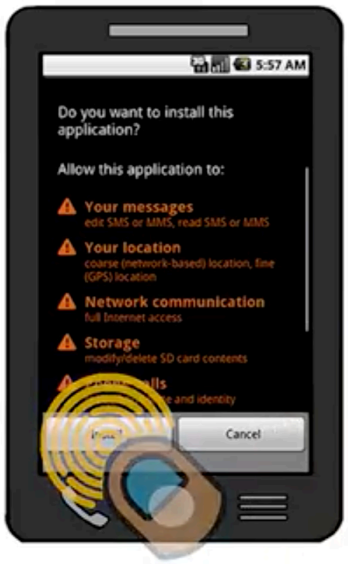
- Efficient test suite
- Potentially better coverage

Both kinds of approaches are useful. And typically a combination is needed.

An example: mobile app security

- Consider the DroidKungFu Malware, which was found to be in circulation on eight 3rd party app stores based out of China around 2011. Prior to installation, this app asks for the following permissions.

- Once installed, the app attempts to collect sensitive information from the compromised device and reports it to remote command and control servers at multiple web locations, such as this one.




```
HttpPost localHttpPost = new HttpPost(...);
(new DefaultHttpClient()).execute(localHttpPost;
```

```
http://[...]search.gongfu-android.com:8511/[...]
```

- **Black-box testing** would detect this malware by merely starting the app and monitoring the network activity of the phone.
  Thereby capturing their attempt to connect to this suspicious web location.


```
HttpPost localHttpPost = new HttpPost(...);
(new DefaultHttpClient()).execute(localHttpPost;
```

```
http://[...]search.gongfu-android.com:8511/[...]
```

- **White-box testing** on the other hand, would involve inspecting the source of binary code of the app, instead of observing the app's input/output behavior in the case of black-box testing. This inspection in turn would reveal this call in the code to connect to the suspicious web location.

- Both the approaches detect the same malicious behavior, but they do so in fundamentally different ways. Of course, either of these approaches could fail to detect this malicious behavior if they are not careful enough. **A combined approach would reduce the chance of such failure**.

The automated testing problem

- Automated testing is hard to do
  - the number of pathways through a program increases exponentially with the number of branch points in the program.
  - if a program has a loop that could potentially be an infinite number of routes through the code.
- Probably impossible for entire systems
- Certainly impossible without specifications

Pre- and post-conditions

- A **pre-condition** is a predicate
  - Assumed to hold before a function executes
- A **post-condition** is a predicate
  - Expectes to hold after a function executes, whenever the pre-condition also holds
- Pre- and post-conditions can be considered as **a special case of assertions**, which we saw in the first lesson.

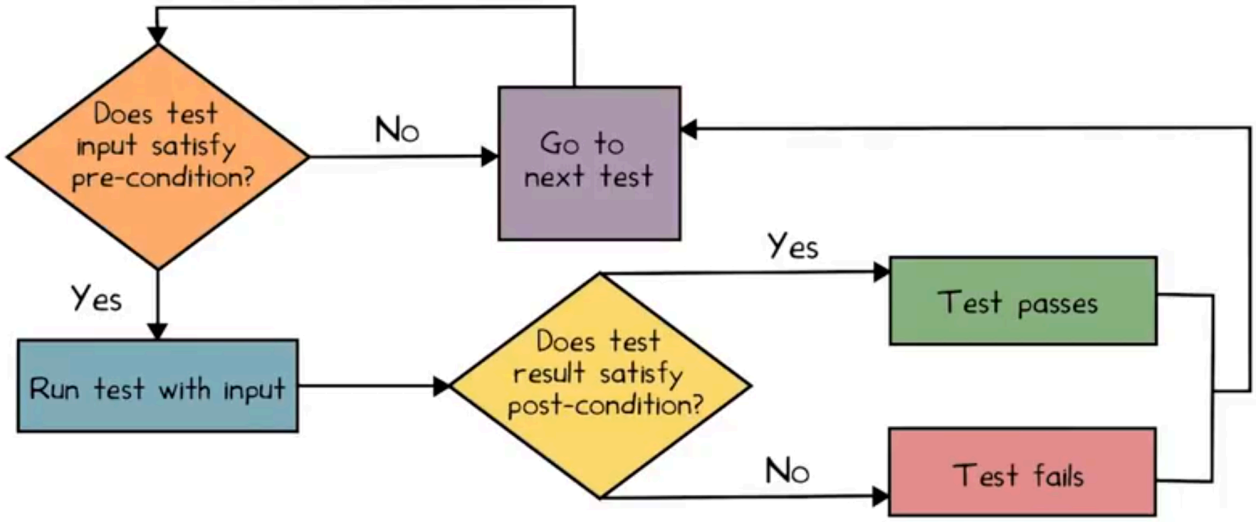
Example

```
Class Stack<T> {
      T[] array;
      int size;        frame conditions

      Pre: s.size() > 0
      T pop() { return array[--size]; }
      Post: s'.size() == s.size() - 1

      int size() { return size; }
```

More pre- and post-conditions

- Most useful if they are executable
  - Written in the programming language itself
  - A special case of **assertions**
- **Need not be precise**
  - May become more complex than the code
  - But useful even if they do not cover every situation

Using pre- and post-conditions



- Doesn't help write tests, but helps run them

Pre-conditions

- The **weakest possible pre-condition** that prevents any in-built exceptions from being thrown in the following Java function

```
int foo(int[] A, int[] B) {
      int r = 0;
      for (int i = 0; i < A.length; i++) {
            r += A[i] * B[i];
      }
      return r;
}
```

is:

```
A != null && B != null && A.length <= B.length
```

Post-conditions

- Consider a sorting function in Java which takes a non-null integer array A and returns an integer array B. The checked items from the following statements specify the **strongest possible post-condition**.
  - ✓ B is non-null
  - ✓ B has the same length as A
  - The elements of B do not contain any duplicates
  - ✓ The elements of B are a permutation of the elements of A
  - ✓ The elements of B are in sorted order
  - The elements of A are in sorted order
  - The elements of A do not contain any duplicates

Executable post-condition

- How the starting post-condition look like if we wrote it in executable code?
  - B is non-null
  - B has the same length as A

    ```
    B != null;
    ```
  - he elements of B are in sorted order

    ```
    B.length == A.length;
    ```
  - The elements of B are a permutation of the elements of A

    ```
    for (int i = 0; i < B.length - 1; i++)
          B[i] <= B[i+1];
    ```

How good is your test suite?

```
// count number of occurrences of
// each number in each array and
// then compare these counts
```

- How do we know that our test suite is good?
  - Too few tests: may miss bugs
  - Too many tests: costly to run, bloat and redundancy, harder to maintain

- Approaches to measure the quality of the test suite:
  - **Code coverage** metrics
  - Mutation analysis (or mutation testing)

Code coverage

- Metric to quantify extent to which a program's code is tested by a given test suite
- Given as **percentage** of some aspect of the program executed in the tests
- 100% coverage rare in practice: e.g., inaccesible code
  - Often required for safety-critical applications

Types of code coverage

- Function coverage: which **functions** were called?
- Statement coverage: which **statements** were executed?
- Branch coverage: which **branches** were taken?
- Many others: line coverage, condition coverage, basic block coverage, path coverage, …

Code coverage metrics

```
int foo(int x, int y) {
      int z = 0;
      if (x <= y) {
            z = x;
      } else {
            z = y;
      }
      return z;
}
```

- Test Suite: foo(1,0)
  - Statement coverage 80%
  - Branch coverage 50%
- Another call to foo to increase both coverages to 100% is foo(1,1)

Mutation analysis

- A key advantage of code coverage metrics is their simplicity, they are not difficult to measure, however they are not perfect. It is possible to attain high code coverage with the test suite, yet not discover potential bugs.

  A more complex process called mutation analysis can be used to provide more confidence in one's test suite.
- Founded on the "competent programmer assumption"

  The program is close to right to begin with

- Key idea: test variations (mutants) of the program
  - Replace `x > 0` by `x < 0`
  - Replace `w` by `w + 1, w – 1`
- If test suite is good, should report failed tests in the mutants
- Find set of test cases to distinguish original program from its mutants

Mutation analysis

```
int foo(int x, int y) {
      int z = 0;
      if (x <= y) {
            z = x;
      } else {
            z = y;
      }
      return z;
}
```

the boxes marked with X indicate a passed test	**Test 1** assert: foo(0,1) == 0	**Test 2** assert: foo(0,0) == 0
Mutant 1 x <= y -> x > y		X
Mutant 2 x <= y -> x != y	X	X

- The test suite is not adequate with respect to both mutants. We need a test case in which the 2nd mutant fails, but the original code passes. E.g., foo(1,0) == 0.

A problem

- While mutation analysis is a powerful tool for measuring the quality of a test suite, one problem that could arise is
  - What if a mutant is equivalent to the original?
- Then no test will kill it
  - If we have such a persistent mutant, it becomes difficult to tell a lack of robustness in our testing or whether it is an equivalent mutant, in which case we can safely ignore the mutant.
- In practice, this is a real problem
  - Not easily solved
  - Try to prove **program equivalence** automatically
  - Often requires manual intervention

What have we learned?

- Landscape of testing

    - Automated vs. manual
    - Black-box vs. white-box
- Specifications: pre- and post-conditions
- Measuring test suite quality

Reality

- Many proposals for improving software quality
- But the world test
    - > 50% of the cost of software development
- Conclusion: **Testing is important**

Notă[1]

[1]