

Software Quality Engineering

03. Unit Testing

Achiya Elyasaf

אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev



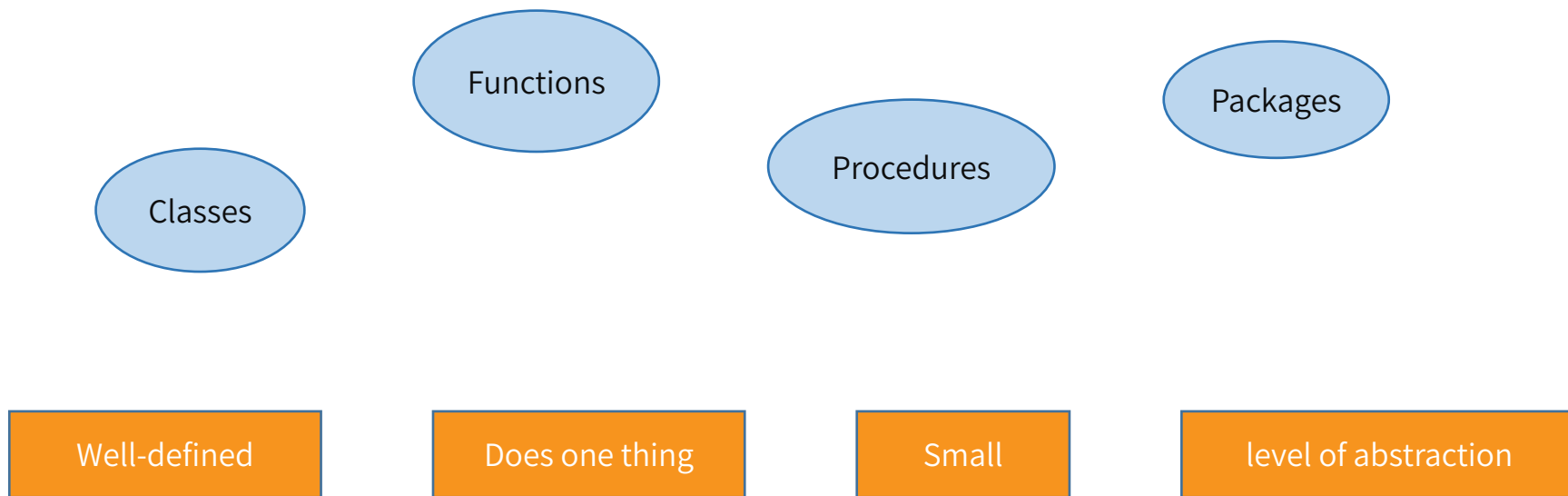
SISE הנדסת מערכות תוכנה ומידע
Software and Information
Systems Engineering

Today's Agenda

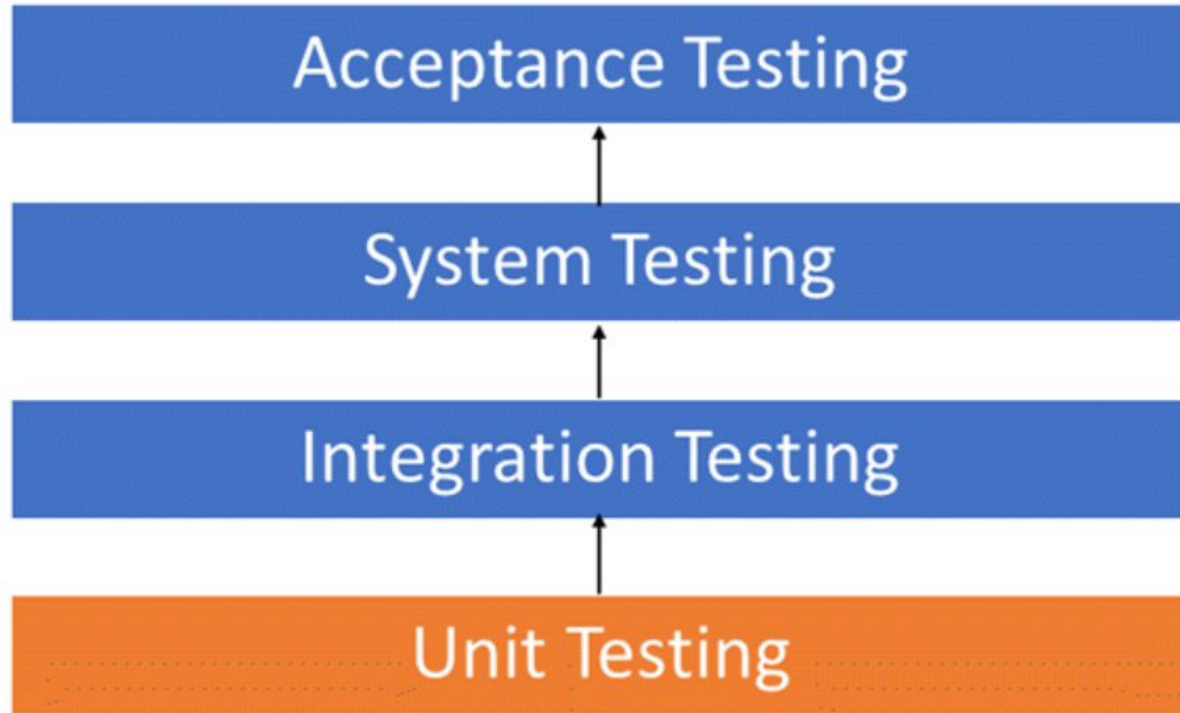
- \ Introduction to unit testing
- \ Static/Dynamic unit testing
- \ Unit Testing in OOP
- \ Mutation testing



What is a unit?



Testing Hierarchy



Why unit testing?

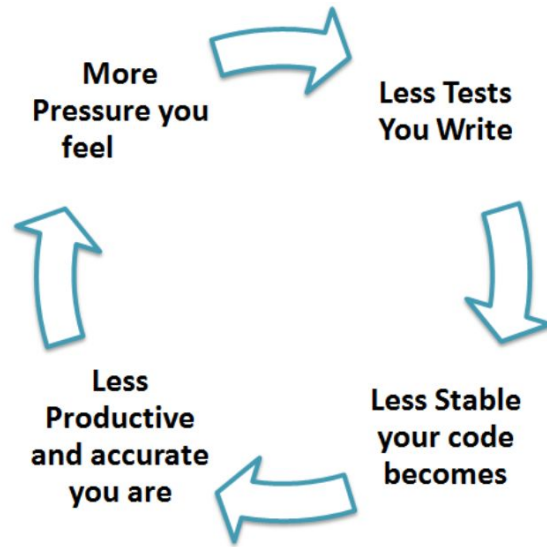
\ Fix bugs early

\ Understand better the code □ quick changes

\ Project documentation

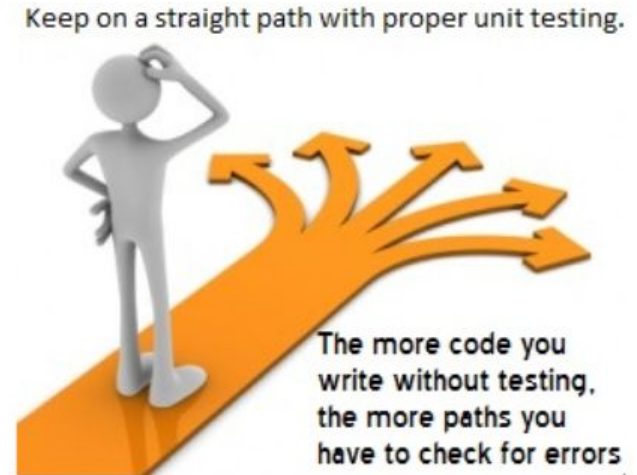
\ Code re-use

\ Migrate both your code **and** your tests to your new project.
Tweak the code till the tests run again.



Concepts of unit testing

- \ **What** to test?
 - \ Test units in isolation
- \ **When** to test?
 - \ Before integrating with other units
- \ **How** to test?
 - \ Expected outputs for inputs
- \ **Who** is performing the test? - developers
 - \ White / black box tests? - **white**
 - \ Is unit testing **sufficient**? – globally not ..



Intuitively...

\Execute **every line** of code

\Execute every predicate - **true** and **false** separately

\Observe that the unit performs its **intended function**

\Ensure that the unit contains **no known errors**



```

public Set<Integer> findMatchingItemsForUser(User user, Set<User> otherUsers){
    Set<User> matchingUsers = new HashSet<>();
    int userAge = user.age;
    for (User otherUser : otherUsers) {
        if (user.age < otherUser.age + 3 && user.age > otherUser.age - 3) {
            matchingUsers.add(otherUser);
        }
    }
    Set<Integer> itemIdsThatMatchingUsersBought = new HashSet<>();
    for (User matchingUser : matchingUsers) {
        Set<Integer> itemIds = matchingUser.itemIds;
        for (Integer itemId : itemIds) {
            itemIdsThatMatchingUsersBought.add(itemId);
        }
    }
    Iterator<Integer> iterator = itemIdsThatMatchingUsersBought.iterator();
    while (iterator.hasNext()) {
        Integer itemId = iterator.next();
        if (user.itemIds.contains(itemId))
            iterator.remove();
    }

    return itemIdsThatMatchingUsersBought;
}

```




```
public Set<Integer> findMatchingItemsForUser(User user, Set<User> otherUsers) {
    Set<User> matchingUsers = getMatchingUsers(user, otherUsers);
    Set<Integer> itemIdsThatMatchingUsersBought = getItemsThatMatchingUsersBought(matchingUsers);
    removeItemsThatUserAlreadyBought(user, itemIdsThatMatchingUsersBought);

    return itemIdsThatMatchingUsersBought;
}

private void removeItemsThatUserAlreadyBought(User user, Set<Integer> itemIdsThatMatchingUsersBought) {
    itemIdsThatMatchingUsersBought.removeIf(itemId -> user.itemIds.contains(itemId));
}

private Set<Integer> getItemsThatMatchingUsersBought(Set<User> matchingUsers) {
    Set<Integer> itemIdsThatMatchingUsersBought = new HashSet<>();
    for (User matchingUser : matchingUsers)
        itemIdsThatMatchingUsersBought.addAll(matchingUser.itemIds);

    return itemIdsThatMatchingUsersBought;
}

private Set<User> getMatchingUsers(User user, Set<User> otherUsers) {
    Set<User> matchingUsers = new HashSet<>();
    for (User otherUser : otherUsers)
        if (isUserInSimilarAge(user, otherUser))
            matchingUsers.add(otherUser);

    return matchingUsers;
}

private boolean isUserInSimilarAge(User user, User otherUser) {
    return user.age < otherUser.age + 3 && user.age > otherUser.age - 3;
}
```

Two phases of unit testing

Static unit testing

Dynamic unit testing



Static unit testing

\ **Inspection** and **correction** at each milestone in a unit's life cycle

\ **Review** before execute

\ **Code review** - a systematic approach to examining source code in detail

\ Review the code, **not the author** of the code



Static unit testing – code review

1. Readiness - author of the unit's responsibility

- \ Completeness

- \ Minimal functionality

- \ Readability (meaningful names, code formatting, etc)

- \ Complexity (enough to justify a review)

- Complexity in terms of #inputs, #conditions, processing time, etc

- \ Requirement/Design documents are available

- To verify the code implements the expected functionalities

2. Preparation

- \ Reviewers review the code before the actual review, prepare questions and suggest changes/solutions



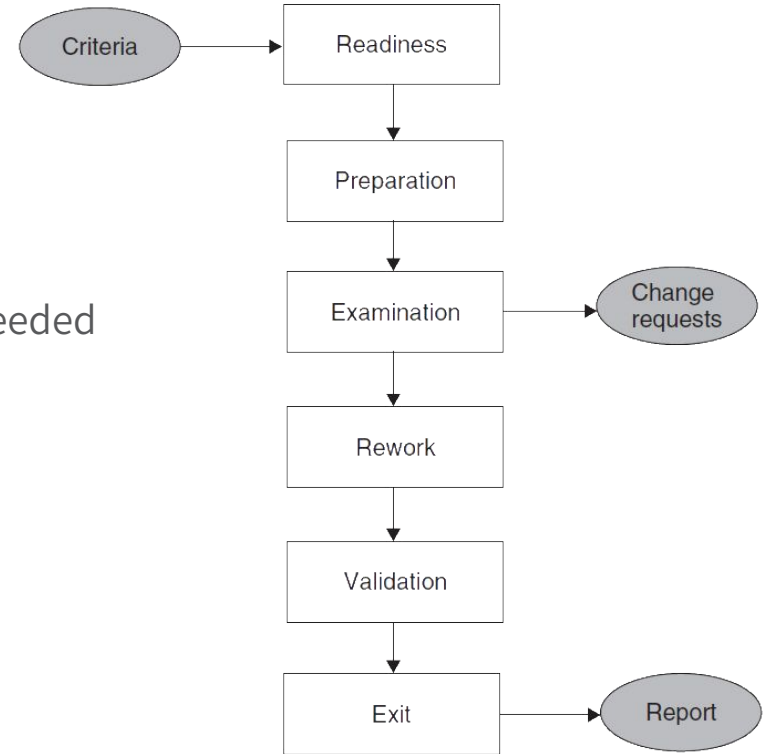
Static unit testing – code review

3. Examination

- \ Author presents the code
- \ Reviewers comment/suggest
- \ A decision is made whether another meeting is needed

4. Rework

- \ A summary of the examination is published
- \ Author is fixing the code



Steps in the code review process



Static unit testing – code review

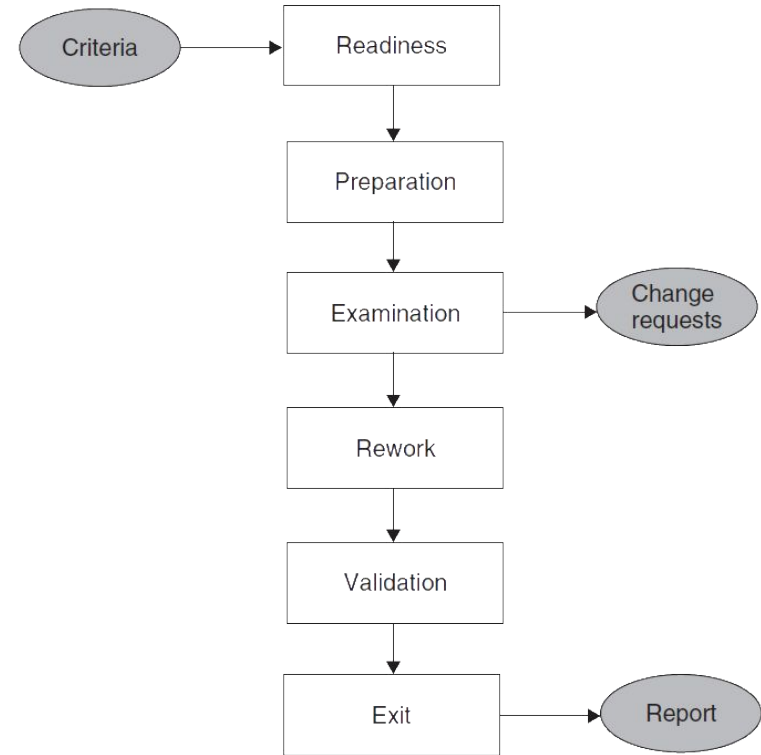
5. Validation

\ Fixes are reviewed, ensuring that the suggested improvements were implemented correctly

6. Exit


\ Summarizes the review process

\ A decision is made whether to schedule another review cycle



Steps in the code review process





**Unit testing -
done**

**Static testing is
not sufficient**



Dynamic unit testing

\Tests are actually executed in isolation

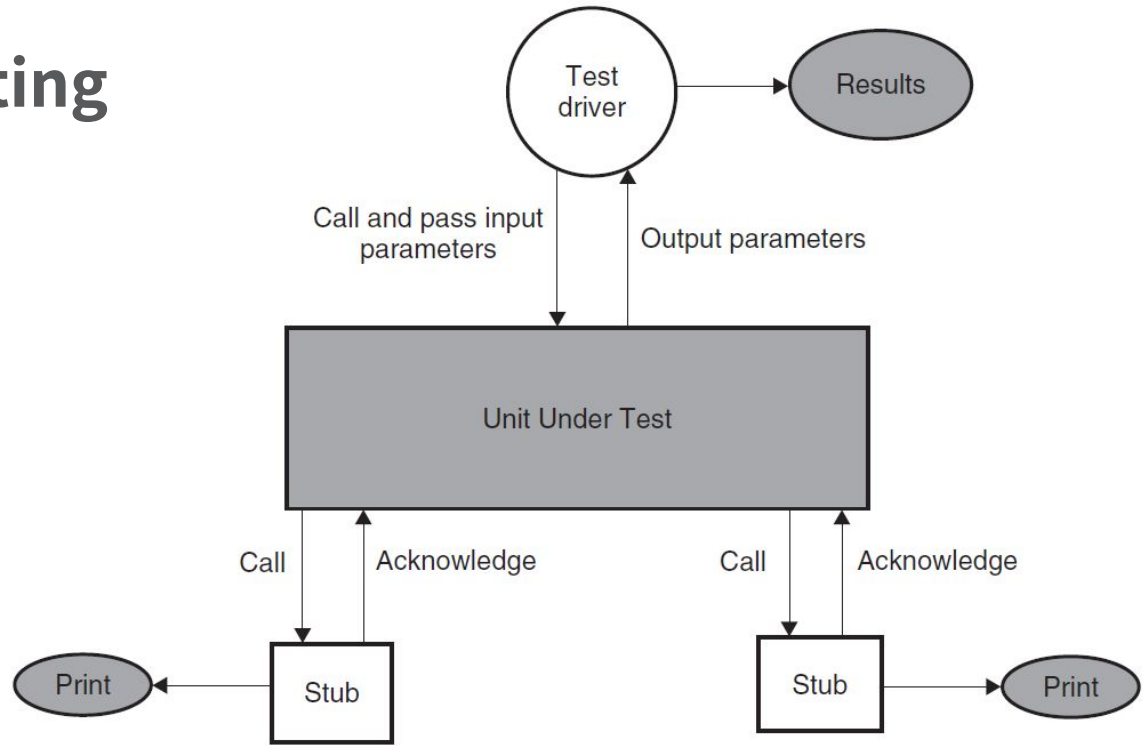
\Actual environment is emulated with drivers and stubs



Dynamic unit testing

Drivers

Stubs

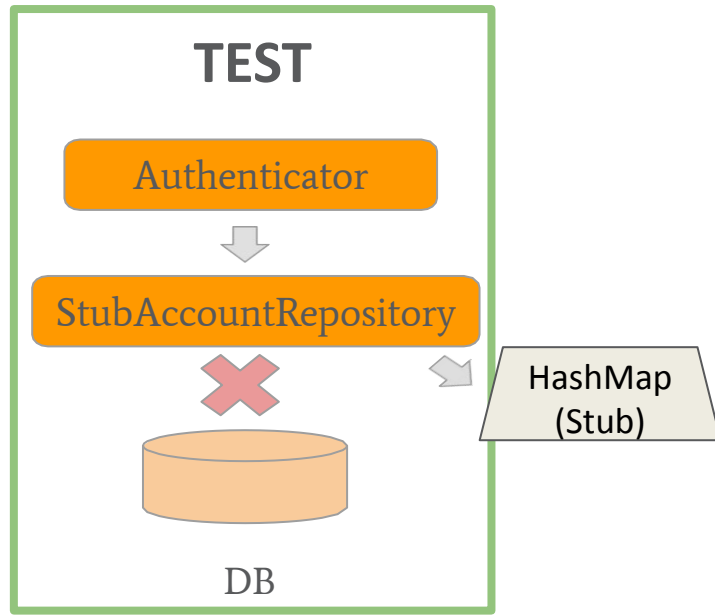
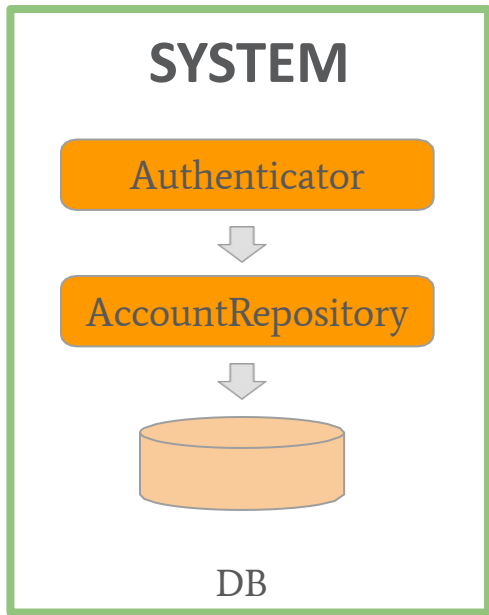


Isolating the code helps in revealing **unnecessary dependencies** between the code being tested and other units or data

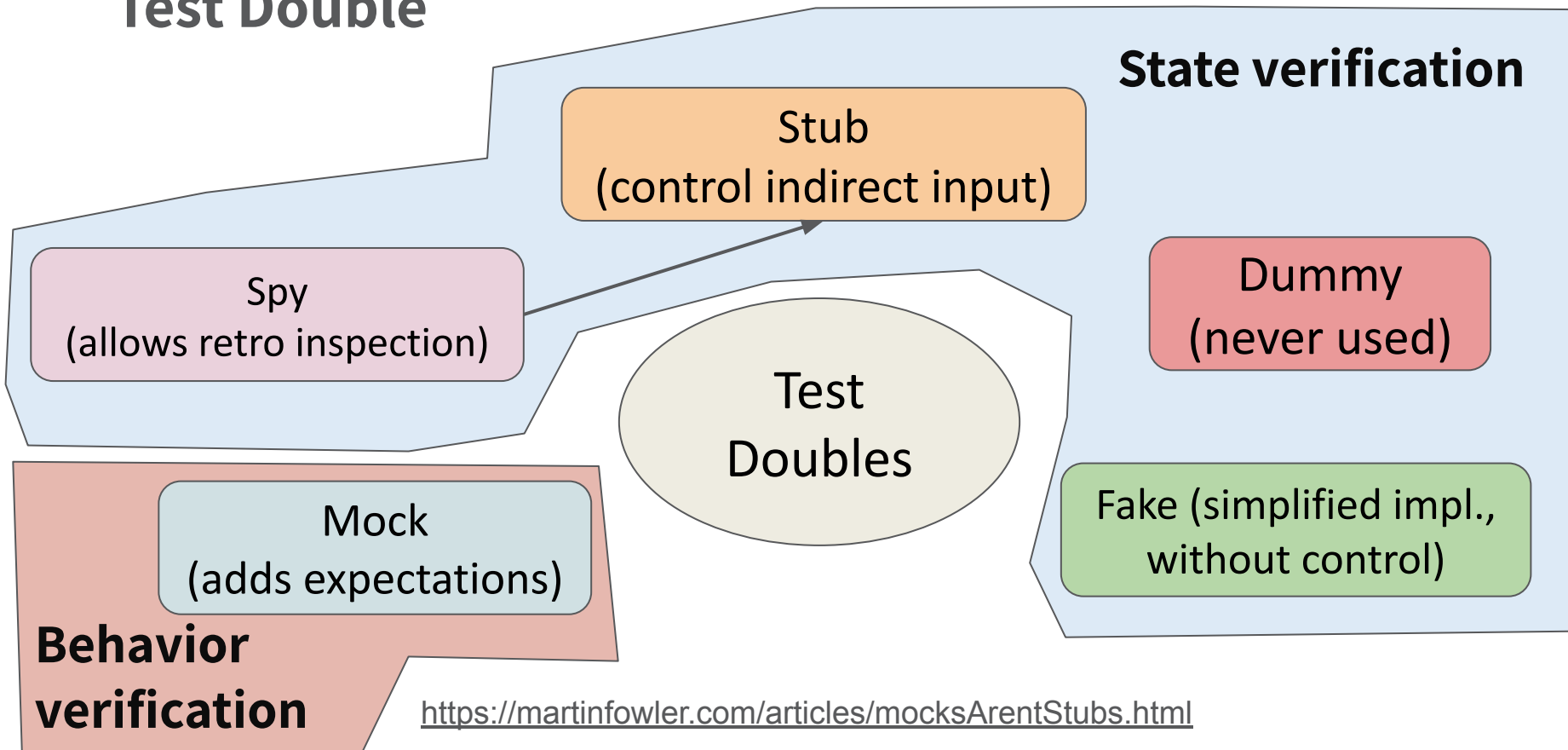


Stubs

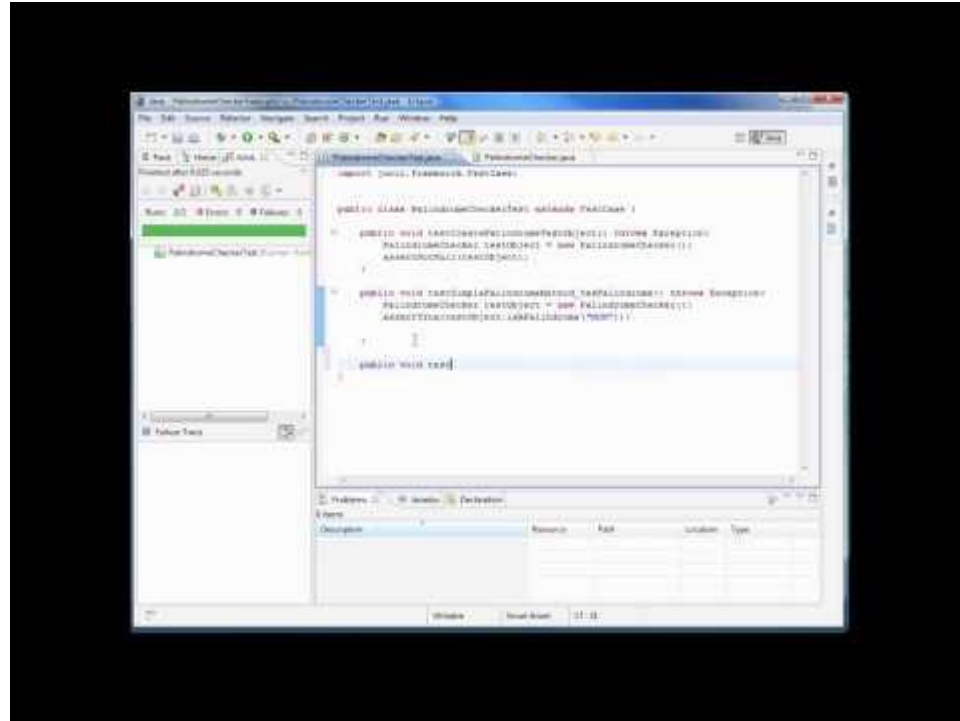
Lightweight implementations that mimic real objects and provide only the necessary methods and responses required for the test.



Test Double



Unit Testing in eXtreme Programming



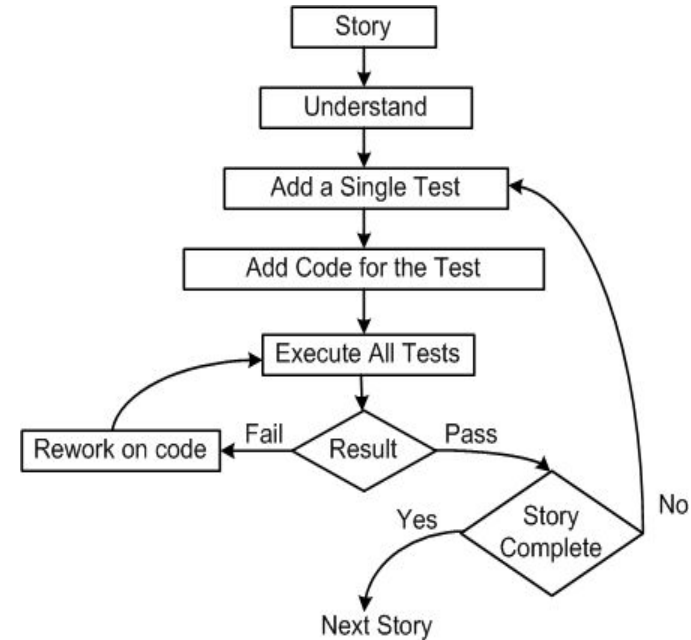
https://youtu.be/O-ZT_dtIrR0



Unit Testing in eXtreme Programming

1. Pick a requirement, i.e., a story
2. Write a test case that verifies a small part of the story and assign a fail verdict to it
3. Write the code that implements a particular part of the story to pass the test
4. Execute all test
5. Rework on the code, and test the code until all tests pass
6. Repeat step 2 to step 5 until the story is fully implemented

Main advantage: the test and code are aligned. When code is written without thinking of “how to test it” - it will be hard to test it.



Unit Testing in eXtreme Programming

\ XP uses a **Test Driven development** approach

\ Programmer writes tests prior to production code:

- \ Understand the story

- \ Write a test case (that fails) to verify a part of the story

- \ Write the code to implement that part of the story

- \ Execute and fix until all tests pass

- \ Repeat until all story is implemented



Unit Testing in eXtreme Programming

Three laws of **Test Driven development** (TDD)

- \ One may not write production code unless the **first failing unit test** is written
- \ One may not write more of a unit test than is **sufficient to fail**
- \ One may not write more production code than is sufficient to make the failing unit test pass

Pair programming:

- \ In XP code is being developed by two programmers working side by side
- \ One person develops the code tactically and the other one inspects it methodically by keeping in mind the story they are implementing



Unit Tests Frameworks

Why do we want to use unit tests frameworks (e.g., JUnit)?

- \ Make writing tests more simple
- \ Help writing individual test cases
- \ ... and organize multiple test cases into a test suite
- \ Initialize a test environment
- \ Execute the test suite
- \ Cleanup the test environment
- \ Record the tests results
- \ Should be supported by modern IDEs
(IDEs - Integrated development environment)



Unit-Testing Frameworks

\ Java

- \ TestNG

- \ JUnit

- \ Mock with Mockito

- \ Frameworks have different attributes, like:

- Lifecycle annotations
- DataProvider

\ Javascript

- \ Jasmine

- \ Mocha

\ Web

- \ Karma – unit tests

- \ Selenium - end-to-end

\ Mobile

- \ Appium

- \ Espresso - only for android



Tools For Unit Testing

- \ Code auditor (basic writing standards)
- \ Bound checker (illegal memory access checker)
- \ Documenters (auto docs generation)
- \ Interactive debuggers
- \ Memory leak detectors
- \ Static code (path) analyzer (e.g., infinite loop detection)
- Software inspection support
- Test coverage analyzer
- Test data generator
- Test harness
- Performance monitors
- Network analyzers
- Simulators and emulators
- Traffic generators
- Version control



Unit Testing in Object Oriented Design

- \ Object-Oriented Testing

- \ Issues in OO Testing

- \ OO Integration Testing

- \ OO System Testing



Hopes and Realities of Testing OO Software

\ Hope

\ Encapsulation: Classes are implemented and tested once and then reused many times as “black boxes”

\ Reality: OO software is more difficult to test than traditional (procedural) software



Hopes and Realities of Testing OO Software

\ Hope

- \ Encapsulation: Classes are implemented and tested once and then reused many times as “black boxes”

\ Reality: OO software is more difficult to test than traditional (procedural) software

\ Positive Sides

- \ A unifying and widely accepted framework for object-oriented technology (UML)

- \ Unit testing = class testing

\ Levels of Object-Oriented Testing

- | | |
|-----------------------|-----------|
| 1. Operation / Method | 2. Class |
| 3. Integration | 4. System |



Two possible definitions of “unit”

1. A *unit* is the **smallest software component** that can be compiled and executed
2. A *unit* is a software component that would **never** be assigned to **more than one developer**

\ The main question for testing a class is whether the **class itself**, or its **methods**, are considered as units

\ Best choice: First bullet – a *class* is a *unit*



What causes the problems for testing?

\ Encapsulation/Composition

\ Inheritance

\ Polymorphism

\ Testing has to be made carefully considering the various objects' relations



Encapsulation/Composition

How to test private method and fields?

Designing for Testability

- \ Dependency Injection
- \ Avoid private (package / public / protected / @VisibleForTesting)
- \ Use Interfaces and Abstract Classes



Polymorphism: Definition

\ “Many forms”

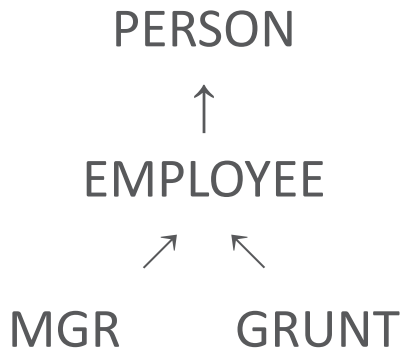
\ An object can be many types at once

\ From before: A class has state and behavior, plus all the states and behaviors of every class it inherits from

\ Methods and operations can be overridden



Polymorphism



If a method P takes Type Person as a parameter, then calling it with an Employee or a Grunt is allowed and must be tested

What if a new inheriting class is added after unit test is written?



Polymorphism: Effects on testing

\ An inherited method that is overridden must be retested in the subclass

\ If the implementation is changed, new white box tests are needed

\ The reason for overriding almost certainly means new or different external behavior, so new black box tests are needed also



Example

Class C has methods M and N, and method M calls method N.

D inherits from C, overriding M; E inherits from D, overriding N.

M for sure must be tested in all 3 contexts.

C M(), N()

↑

D overrides M(), inherits N()

↑

E inherits M(), overrides N()



Debugging

- \ Determining the **cause** of a failure
- \ Preferably conducted by the **programmer** who wrote the code
- \ A consequence of a test revealing a failure
- \ Approaches in the book “*The Art of Software Testing*”:
 - \ Brute force (print statements, debugging tools, logs, memory dump)
 - \ Cause elimination (induction and deduction of behavior and symptoms to isolate the causes)
 - \ Backtracking (begin at a point in the code where a failure was observed and traces back)



Mutation Testing

How do I know that my unit tests are good enough



Mutation testing

\ Measure the quality of test cases

\ Mutation: a **small modification** of the code.

\ The modified code is called a mutant

- Example: $x > 1$ \square $x \geq 1$

\ A test suite is determined to distinguish a program from its mutants



Mutation testing

\ **Killed** (test failed)

\ **Equivalent** (behaves like the original)

\ $\text{res} = x + y$ \square Mutant: $\text{res} = y + x$

\ **Stubborn** (test set is insufficient)

\ The existing set of test cases is insufficient



Mutation test - process

Step 1: Begin with a program P and a set of test cases T known to be correct.

Step 2: Run each test case in T using program P. If a test case fails, thus, the **output is incorrect**, program P must be **modified** and **retested**. If there are no failures, then continue with step 3.



Mutation test - process

\ **Step 3:** Create a set of mutants $\{P_i\}$, each differing from P by a simple, syntactically correct modification of P .

\ **Step 4:** Execute each test case in T using each mutant P_i .

\ P_i is killed.

\ P_i and P are equivalent.

\ P_i is killable (stubborn).



Mutation test - process

Step 5: Calculate the mutation score = $100 \times D / (N - E)$, where D is the number of **dead mutants**, N is the total number of mutants, and E is the number of equivalent mutants.

Step 6:

- \ Design a new test case, add it to T, and go to step 2.
- \ Accept T as a good measure of the correctness of P with respect to the set of mutant programs P_i .



Mutation testing - assumptions

\ Competent Programmer Hypothesis — only create “simple errors”

\ The coupling effect

\ *“If the software **contains a fault**, there will be usually a set of mutants that can only be **killed by a test case** that also detect that fault.”* Geist et al.

\ In other words: complex faults are coupled to a set of simple faults

=> a test suite detecting all simple faults in a program will detect most of the complex faults



Mutation testing

Test cases:

<{1, 1}, 1>

<{2, 1}, 2>

<{3, 1}, 2>

```
function foo(a, b) {  
    let z = null;  
    if (a === b)  
        z = 1;  
    else  
        z = 2;  
}
```

```
function fooMutation(a, b) {  
    let z = null;  
    if (a === b + 1)  
        z = 1;  
    else  
        z = 2;  
}
```



State of Mutation Testing at Google

Goran Petrović
Google Inc.

goranpetrovic@google.com

Marko Ivanković
Google Inc.

markoi@google.com

```
namespace testing {  
namespace mutation {  
namespace example {  
  
int RunMe(int a, int b) {  
    if (a == b || b == 1) {
```

The cost of executing tests for all mutants is prohibitive, therefore, new techniques are needed.

▼ Mutants
14:25, 28 Mar

Changing this 1 line to
if (a != b || b == 1) {

does not cause any test exercising them to fail.

Consider adding test cases that fail when the code is mutated to ensure those bugs would be caught.

Mutants ran because goranpetrovic is whitelisted

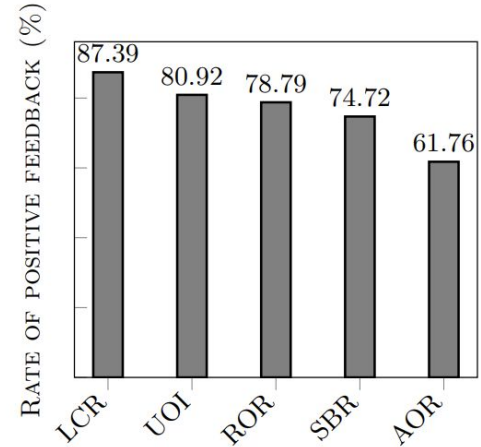
[Please fix](#)

[Not useful](#)

```
    return 1;  
}  
    return 2;  
}  
  
} // namespace example  
} // namespace mutation  
} // namespace testing
```

...ants of the original code, and judges the effectiveness of
...ect those faults. Mutation
...e best method of evaluating
...esen
...n de
...ctin

...ing a powerful tool, it is often con
...thermore, the cost of develop
...mitigating the results of the a
...can be exorbitant even for med
...to leverage mutation analysis in
...tem like Google's, in this work
...roach to mutation analysis. Mor
...be a method of transitive mutati
...ng, arid lines based on develop
...F. A diff-based approach greatly
...s in which mutants are created
...lines cuts the number of potent
...ed, these two approaches make n
...n for colossal complex systems (C
...y contains approximately 2 billion



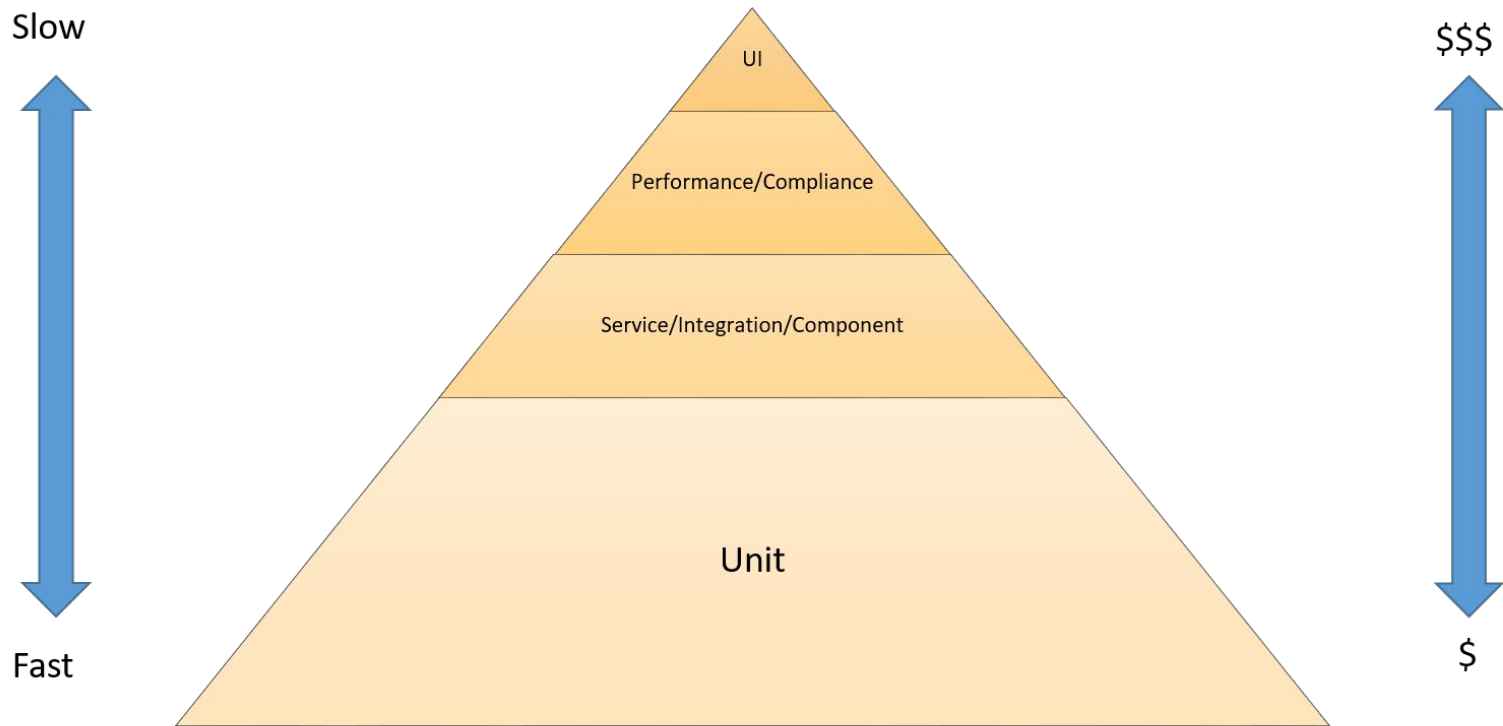
(a) Usefulness by mutation operator

	NAME	SCOPE
AOR	Arithmetic operator replacement	$a + b \rightarrow \{a, b, a - b, a * b, a / b, a \% b\}$
LCR	Logical connector replacement	$a \&\& b \rightarrow \{a, b, a b, true, false\}$
ROR	Relational operator replacement	$a > b \rightarrow \{a < b, a \leq b, a \geq b, true, false\}$
UOI	Unary operator insertion	$a \rightarrow \{a ++, a --\}; b \rightarrow !b$
SBR	Statement block removal	$stmt \rightarrow \emptyset$

Figure 3: Mutation operators



Conclusion



Ref: Mike Cohn, Succeeding with Agile

