Testing principles and unit testing

219343 Software Testing

Jittat Fakcharoenphol

Kasetsart University

January 24, 2013

Outline

Testing principles[1]

Principles provide a general guideline.

[1] Materials from Graham, van Veenendall, Evans, and Black, *Fundamentals of Software Testing: ISTQB Certification*, Thomson, 2007.

Testing shows presence of defect

Exhaustive testing is impossible

Early testing

Example: The Triangle Test[2]

- Given the lengths of three sides of a triangle, determine the type of that triangle.
  - Equilateral
  - Isosceles
  - Scalene

[2] Taken from Black, *Pragmatic Software Testing*, Wiley, 2007

The code that can be tested

Consider this code:

```
package lect02;

public class Triangle1 {

static void checkType(int a, int b, int c) {
  if((a==b) && (b==c))
    System.out.println("Equilateral");
  else if((a==b) ||(b==c) ||(a==c))
    System.out.println("Isosceles");
  else
    System.out.println("Scalene");
}

public static void main(String [] argv) {
  int a,b,c;
  // .... read input
  checkType(a,b,c);
}
}
```

The code that can be tested

Consider this code:

```
package lect02;

public class Triangle1 {

static void checkType(int a, int b, int c) {
  if((a==b) && (b==c))
   System.out.println("Equilateral");
  else if((a==b) ||(b==c) ||(a==c))
   System.out.println("Isosceles");
  else
   System.out.println("Scalene");
}

public static void main(String [] argv) {
  int a,b,c;
  // .... read input
  checkType(a,b,c);
}
}
```

- How can we test this program, in particular method checkType?

The code that can be tested

Consider this code:

```
package lect02;

public class Triangle1 {

static void checkType(int a, int b, int c) {
  if((a==b) && (b==c))
    System.out.println("Equilateral");
  else if((a==b) ||(b==c) ||(a==c))
    System.out.println("Isosceles");
  else
    System.out.println("Scalene");
}

public static void main(String [] argv) {
  int a,b,c;
  // .... read input
  checkType(a,b,c);
}
}
```

- How can we test this program, in particular method checkType?
- Automatically?

The code that can be tested

Consider this code:

```
package lect02;

public class Triangle1 {

static void checkType(int a, int b, int c) {
  if((a==b) && (b==c))
    System.out.println("Equilateral");
  else if((a==b) ||(b==c) ||(a==c))
    System.out.println("Isosceles");
  else
    System.out.println("Scalene");
}

public static void main(String [] argv) {
  int a,b,c;
  // .... read input
  checkType(a,b,c);
}
}
```

- How can we test this program, in particular method checkType?
- Automatically? Very very difficult. Because?

The code that can be tested

Consider this code:

```
package lect02;

public class Triangle1 {

static void checkType(int a, int b, int c) {
  if((a==b) && (b==c))
   System.out.println("Equilateral");
  else if((a==b) ||(b==c) ||(a==c))
   System.out.println("Isosceles");
  else
   System.out.println("Scalene");
}

public static void main(String [] argv) {
  int a,b,c;
  // .... read input
  checkType(a,b,c);
}
}
```

- How can we test this program, in particular method checkType?
- Automatically? Very very difficult. Because?
- It is hard to check the output of the method.

After some fix

```
package lect02;

public class Triangle {

enum Type { Equilateral, Isosceles, Scalene }

static Type checkType(int a, int b, int c) {
  if((a==b) && (b==c))
   return Type.Equilateral;
  else if((a==b) ||(b==c) ||(a==c))
   return Type.Isosceles;
  else
   return Type.Scalene;
}
}
```

Practice

Find as many interesting test cases for this method.
Write in this form:

Tester action and data	**Expected result**

Practice

Find as many interesting test cases for this method.
Write in this form:

Tester action and data	**Expected result**

We will use this example to talk about good test set again.

Unit testing: classic example[3]

- **John**: John works hard. He codes everyday. The project deadline is tomorrow. He types in about two hundred new lines per hour, and thinks that after 6 hours and roughly a thousand new lines added the program would work flawlessly.

[3] Materials regarding unit testing concepts are from Alberto Savoia's slides and George Necular's software engineering course

Unit testing: classic example[3]

- **John**: John works hard. He codes everyday. The project deadline is tomorrow. He types in about two hundred new lines per hour, and thinks that after 6 hours and roughly a thousand new lines added the program would work flawlessly.
- **Betty**: Betty works hard. She codes everyday. The project deadline is tomorrow. She types in about one hundred new lines per hour, and keeps testing each method she adds. She does not proceed to write new codes unless all previously written pieces work correctly.

[3] Materials regarding unit testing concepts are from Alberto Savoia's slides and George Necular's software engineering course

Unit testing: classic example[3]

- **John**: John works hard. He codes everyday. The project deadline is tomorrow. He types in about two hundred new lines per hour, and thinks that after 6 hours and roughly a thousand new lines added the program would work flawlessly.
- **Betty**: Betty works hard. She codes everyday. The project deadline is tomorrow. She types in about one hundred new lines per hour, and keeps testing each method she adds. She does not proceed to write new codes unless all previously written pieces work correctly.
- Guess who will go to bed earlier?

[3] Materials regarding unit testing concepts are from Alberto Savoia's slides and George Necular's software engineering course

Developer Testing Revolution

- Developer testing is a key component in a hot paradigm: Agile/eXtreme Programming.

Developer Testing Revolution

- Developer testing is a key component in a hot paradigm: Agile/eXtreme Programming.
- The Developer Testing Trinity:
  - Test
  - Test early and often
  - Test well

Good reasons for developer testing

- Reduces unit-level bugs

Good reasons for developer testing

- Reduces unit-level bugs
- Forces you to slow down and think

Good reasons for developer testing

- Reduces unit-level bugs
- Forces you to slow down and think
- Improves design

Good reasons for developer testing

- Reduces unit-level bugs
- Forces you to slow down and think
- Improves design
- Makes development faster

Good reasons for developer testing

- Reduces unit-level bugs
- Forces you to slow down and think
- Improves design
- Makes development faster
- Tests are good documentation

Good reasons for developer testing

- Reduces unit-level bugs
- Forces you to slow down and think
- Improves design
- Makes development faster
- Tests are good documentation
- Tests constrain features

Good reasons for developer testing

- Reduces unit-level bugs
- Forces you to slow down and think
- Improves design
- Makes development faster
- Tests are good documentation
- Tests constrain features
- Tests allows safe refactoring and reduce the cost of change

Good reasons for developer testing

- Reduces unit-level bugs
- Forces you to slow down and think
- Improves design
- Makes development faster
- Tests are good documentation
- Tests constrain features
- Tests allows safe refactoring and reduce the cost of change
- Tests defend against other programmers

Good reasons for developer testing

- Reduces unit-level bugs
- Forces you to slow down and think
- Improves design
- Makes development faster
- Tests are good documentation
- Tests constrain features
- Tests allows safe refactoring and reduce the cost of change
- Tests defend against other programmers
- Tests reduce fear

Goals

- Does the code do what I want?

Goals

- Does the code do what I want?
- Does the code do what I want all the time?

Goals

- Does the code do what I want?
- Does the code do what I want all the time?
- Can I depend on it?

Goals

- Does the code do what I want?
- Does the code do what I want all the time?
- Can I depend on it?
- Also: get a document for the code.

Goals

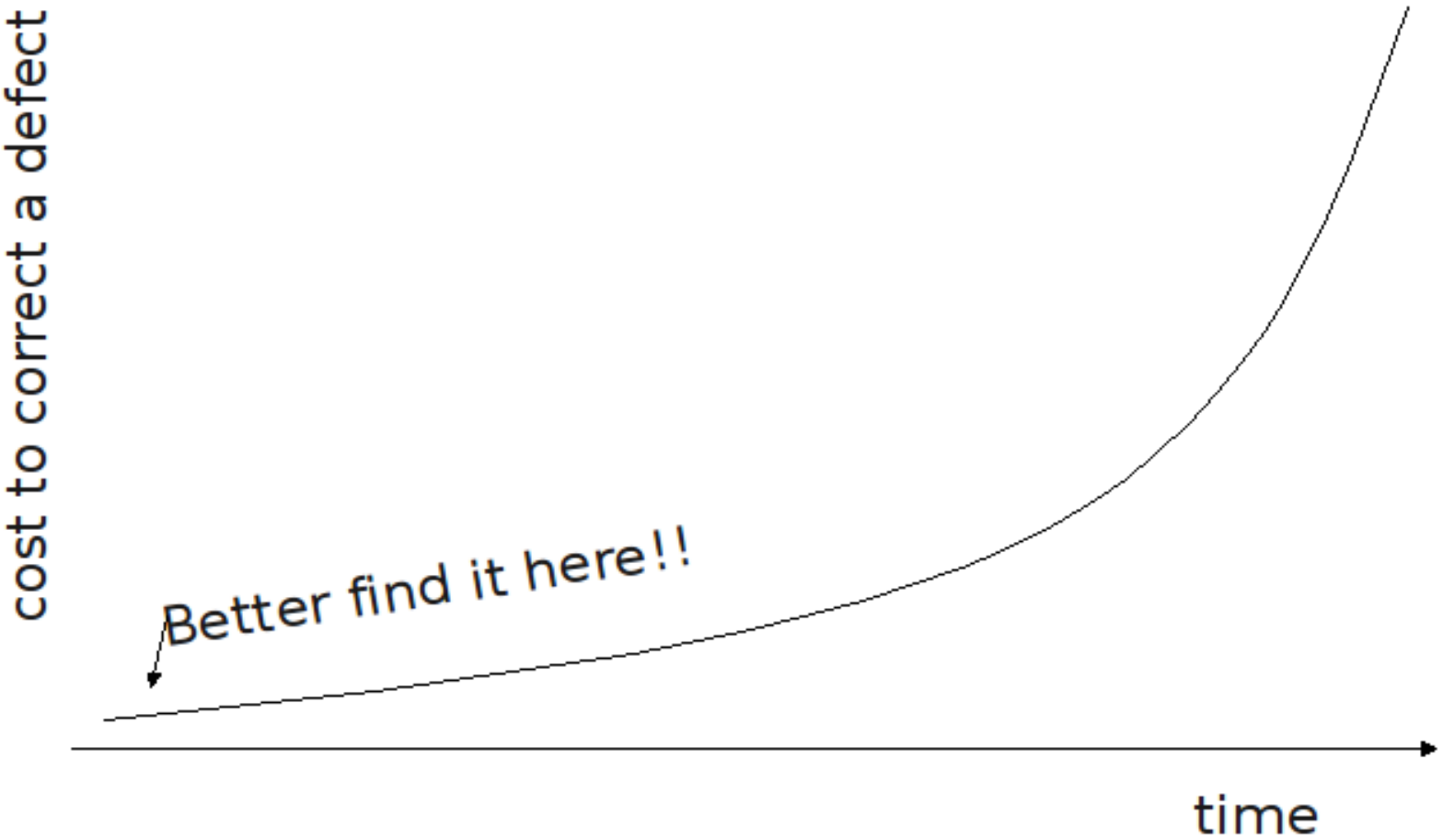
- Does the code do what I want?
- Does the code do what I want all the time?
- Can I depend on it?
- Also: get a document for the code.
  Plus: Always correct documentation for your intention.

Test your code

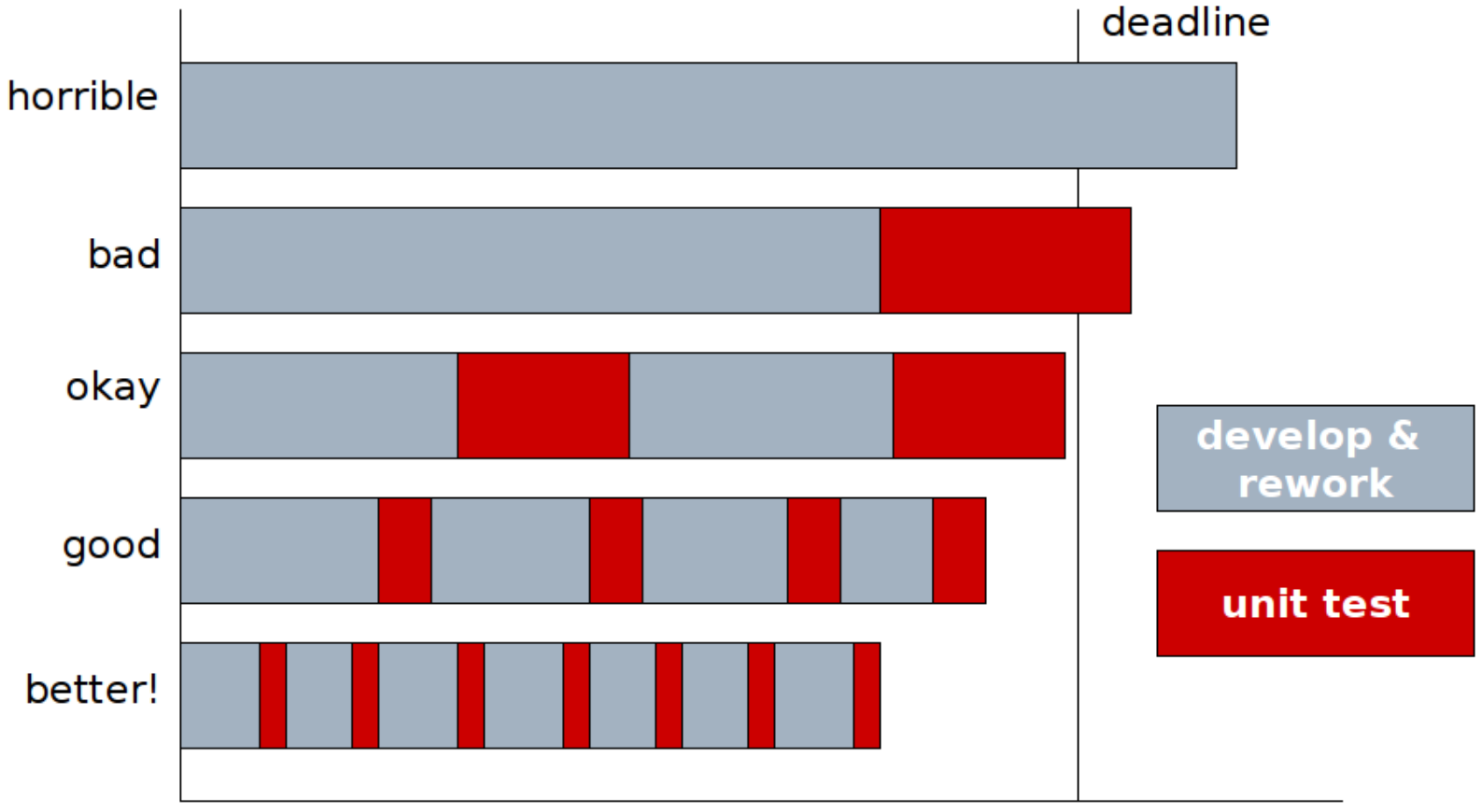
It is your code, and your responsibility

- Do it for your current colleagues
- Do it for future generation of colleagues
- Do it for yourself

Test early and often



Test early and often



Heaven!

Heaven!

- Every class has unit tests

Heaven!

- Every class has unit tests
- The tests are executed many times each day

Heaven!

- Every class has unit tests
- The tests are executed many times each day
- The tests are thorough, up to date, and easy to maintain and analyze

Heaven!

- Every class has unit tests
- The tests are executed many times each day
- The tests are thorough, up to date, and easy to maintain and analyze
- **In this class, we shall aim for that.**

A unit

- What's a unit?
  - A single method/function/procedure
  - A collection of related methods/functions/procedues

A unit

- What's a unit?
  - A single method/function/procedure
  - A collection of related methods/functions/procedues
- Ideal world:

A unit

- What's a unit?
  - A single method/function/procedure
  - A collection of related methods/functions/procedues
- Ideal world: independent, self-sufficient, standalone

A unit

- What's a unit?
  - A single method/function/procedure
  - A collection of related methods/functions/procedues
- Ideal world: independent, self-sufficient, standalone
- Real world:

A unit

- What's a unit?
  - A single method/function/procedure
  - A collection of related methods/functions/procedues
- Ideal world: independent, self-sufficient, standalone
- Real world: lots of dependence

Basic structure

- Setup
  - Create initial states
  - Initialize method parameters
  - Store pre-execute values

Basic structure

- Setup
  - Create initial states
  - Initialize method parameters
  - Store pre-execute values
- Execute code

Basic structure

- Setup
  - Create initial states
  - Initialize method parameters
  - Store pre-execute values
- Execute code
- Compare results

Practice: Triangle

Write junit test cases for Triangle.

Discussion: test cases for Triangle

What are your test cases?

Test-driven development

- Traditional steps:
  design, code, test, design, code, test, ...

Test-driven development

- Traditional steps:
  design, code, test, design, code, test, ...
- TDD:
  test, code, refactor, test, code, refactor, ...

Test-driven development

- Traditional steps:
  design, code, test, design, code, test, ...
- TDD:
  test, code, refactor, test, code, refactor, ...
- We'll discuss more about TDD later.

Test-driven development

- Traditional steps:
  design, code, test, design, code, test, ...
- TDD:
  test, code, refactor, test, code, refactor, ...
- We'll discuss more about TDD later. For now, let's do it a little.

TDD: Triangle

See demo.

Practice: Median