# Test Case Design: Specifications and adequacy

Barbara Russo

SwSE - Software and Systems Engineering research group

unibz
Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

# Goal of testing

*Testing is the process of executing a program with the intent of finding errors*

Glen Myers

"The Art of Software Testing"

# Testing

*Software testing is the process of analysing a software item to detect the differences between* ***existing and required conditions*** *and to evaluate the features of the software item*

IEEE definition

# Limits of software testing

*Program testing can be used to show the presence of bugs, but never to show their absence!*

Dijkstra, 1969

# Limits of software testing

- If a failure is observed, then the software is a failure software, but
- If no failure has been observed, we cannot say that the software is correct
- **Exhaustive testing is not feasible**, but we can compute the probability that no failures occur in a given interval of time

—> **software reliability**

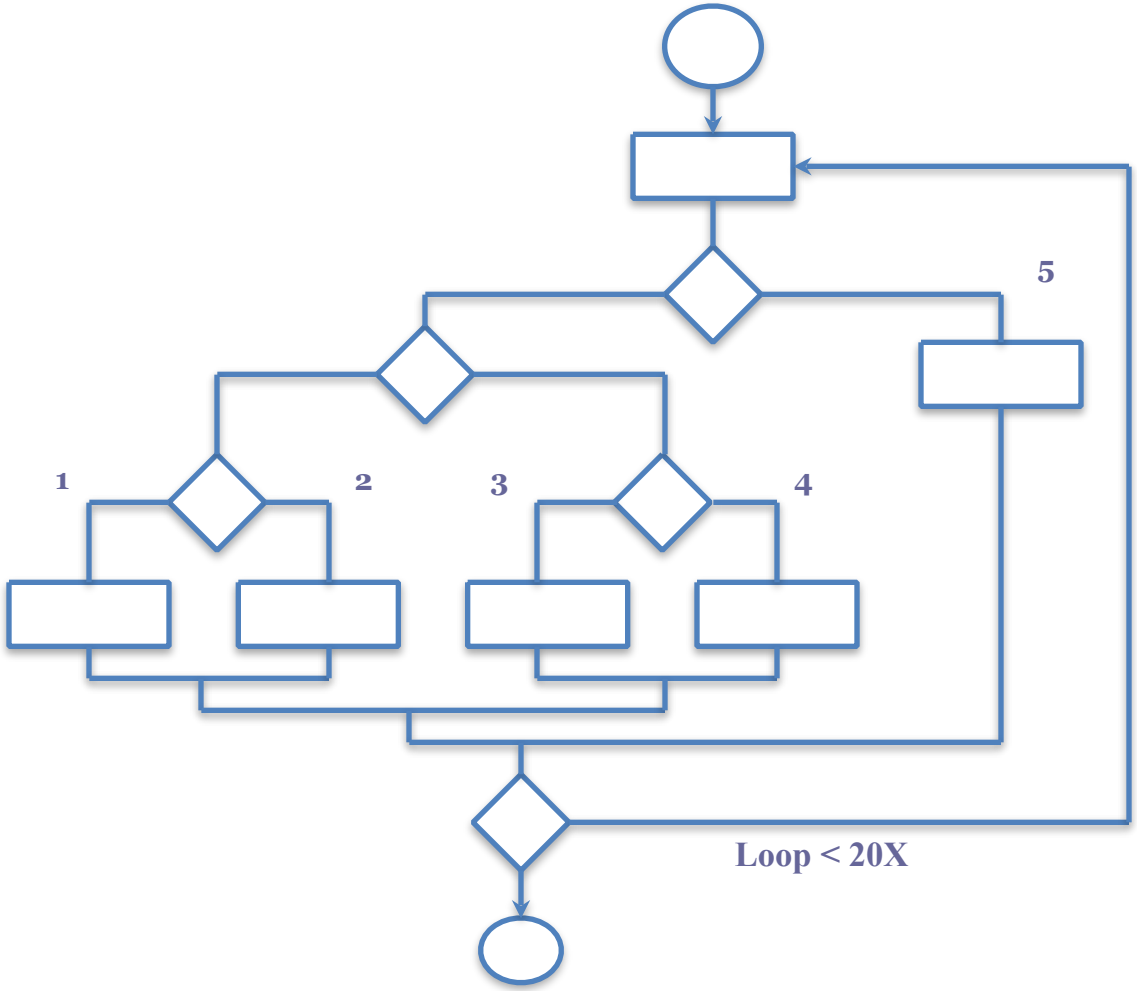
# Limits of software testing

*Beware of bugs in the above code; I have only proved it correct, not tried it*

Knuth, 1977

- We need to test for confidence not for proof of correctness

# Limits of software testing




if we execute one test per millisecond, it would take too much to test this program!!

# Basic questions

- When does testing start? When does it complete?
- What techniques should be applied during software development to get acceptable quality at acceptable cost?
- How can we assess the readiness of a product to release?
- How can we control the quality of a product to release?

# Test Case Design

- How do we design tests?

*Tests are defined in terms of their **adequacy** against certain criteria and according to **specifications***

# Test completeness and adequacy

- It is **impossible** to find a set of tests that **ensures the correctness** of a product
- We can only determine whether **test sets are not adequate** for a given criterion we set

# Examples - Inadequacy


- A test suite is inadeguate to *guard against faults* in:
  - **Specifications**. If in the specifications, we give different permissions to different actors of a system and a test suite does not check that the permissions are different
  - **Statements**. If the quality concerns statements' coverage and a test suite does not cover all the executable statements (except infeasible statements)

# Examples - Inadequacy

redundancy

Adequacy criterion

- A test suite is ina *ard against faults* in:
  - **Specifications**. If in the specifications, we give different permissions to different actors of a system and a test suite does not check that the permissions are different
  - **Statements**. If the quality concerns statements' coverage and a test suite does not cover all the executable statements (except infeasible statements)

# Examples - Inadequacy

redundancy

Adequacy criterion

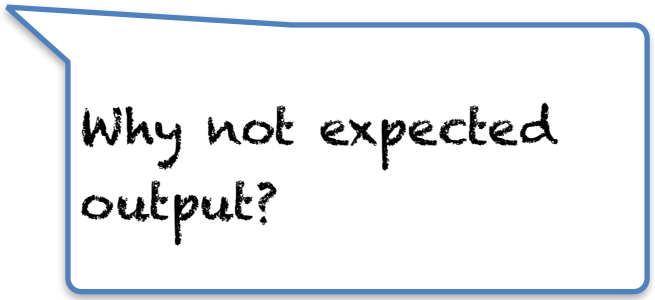
- A test suite is ina… *ard against faults* in:
  - **Specifications**. If i… …ions, we give different permissi… …ctors of a system and a test suite do… …t the permissions are different

Adequacy criterion

  - **Statements**. If the quality concerns statements' coverage and a test suite does not cover all the executable statements (except infeasible statements)

# Test Case

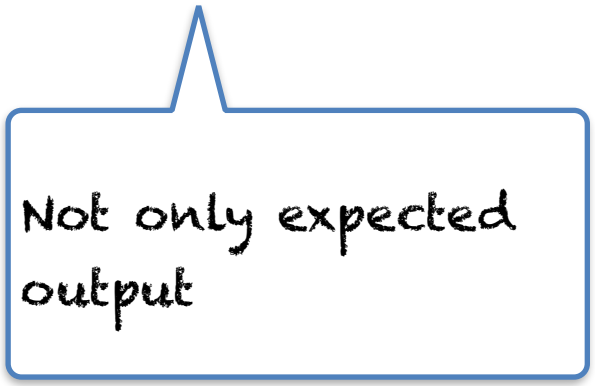
- A test case is a **choice of 1) Inputs, 2) Execution Conditions and 3) Pass / Fail Criterion**




- Example “Permission”.
  - I: {read, write},
  - EC: under a certain domain environment,
  - PF C:{when owner ->TRUE, when guest -> FALSE}

# Test Case

- Input: all kind of stimuli that contribute to a specific behaviour
- Output oracle: given against expected output **or** other peculiar way to determine that an output is correct (e.g., 100% coverage)



# Test case - notation

- **$T_{\text{Output value}}$ (input values)**

- $T_{\text{FileNotFoundException}}$(0, “Hello”, 0.3)
- $T_3$(0, “Home”, 3)
- $T_{(3,4)}$(1, “Home”, “Layout”)

# Exercise - create test cases

```
[1] int foo (int a, int b, int c, int d, float e) {
[2]    if (a == 0) {
[3]        return 0;
[4]    }
[5]    int x = 0;
[6]    if ( (a==b) II ( (c == d) && bug(a) ) ) {
[7]        x=1;
[8]    }
[9]    e = 1/x;
[10]    return e;
[11] }
```

bug(a) = TRUE if !a==0 else 0

What is the input space for the test case? Partition the space

Partitioning

# Exercise - create test cases

BuggyDates.java | *BufferOverFlow.java

```
1  package it.unibz;
2
3  public class BufferOverFlow {
4
5      public static void main(String[] args) {
6
7          int[] vals = new int[10];
8          try {
9              for (int i = 0; i < 20; i++) {
10                 vals[i] = i;
11             }
12         } catch (java.lang.ArrayIndexOutOfBoundsException e) {
13             System.out.println("do not do anything");
14         }
15     }
16 }
17
```

What do we test?
What is the input space for the test case? Partition the space

Problems | Javadoc | Declaration | Console

```
<terminated> BufferOverFlow [Java Application] /Library/Java/JavaVirtualMachines/jdk-12.0.1.jdk/Contents/Home/bin/java (Mar 9, 2021, 5:12:58 PM)
do not do anything[I@7cca494b
```

# Test Case

Search in the space of the input values

- A **good test case** has a high probability of finding an as-yet undiscovered error
- A **successful test case** is one that uncovers an as-yet undiscovered error

# Test Case Specification

- A specification to be satisfied by one or more test cases.
- Examples:
  - **TC specification:** A system has multiple actors.
    - **TC Input:** {owner, guest, administrator}
  - **TC specification:** Word processor must open one or more files;
    - **TC 1 Input:** one file; **TC2 Input:** 2 files

What is the corresponding functionality specification?

# Test Case Specification

- It may also describe **some aspects** of input and output. Example:
  - **TC specification**: Word processor requires some *recovery policy* while opening files

What is the difference with software specification? What in this case?

# How to derive Test Case Specifications

- It depends on types of testing
  - Functional testing (Black-box testing).
  - Structural testing (White-box)
  - Fault based testing
  - Fault-seeding testing

# Functional Testing (Black-box testing)

- Test Case specification can be derived
  - from *product specification*, which in turn can include description of input and output, or
  - from any system *observable behaviour*

# Exercise

**Auction:**

| **Title: Check price validity** |
| --- |
| **Test:** CheckPriceValidityTest |
| A registered user inputs his bid price. Inputted price is checked against auction's next price. If the price is higher, then the given bid price is valid for placing a bid, if not, then the price is not valid for placing a bid. |

| **Title: CheckPriceValidityTest** | | |
| --- | --- | --- |
| **Input** | **Description** | **Output** |
| Price as double value, which is not negative and higher than next auction's price. | On input of the price, it is checked whether it is in correct format and higher or lower than the auction's next price. | Given bid price is valid for placing a bid |
| Price as double value, which is negative or lower than next auction's price. | On input of the price, it is checked whether it is in correct format and higher or lower than the auction's next price. | Given bid price is not valid for placing a bid |

| **Title: Check auction's time validity** |
| --- |
| **Test:** CheckTimeValidityTest |
| A registered user checks the auction's time validity. If the auction has started and not yet expired, then the user can place bids, if it has not started or expired, then it is not possible to place bids. |

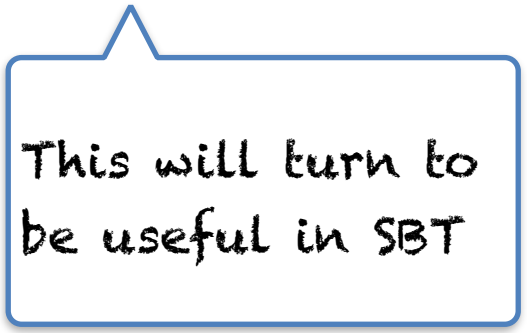
| **Title: CheckTimeValidityTest** | | |
| --- | --- | --- |
| **Input** | **Description** | **Output** |
| User opens auction when it has started and has not expired | System checks whether the auction has started and not yet expired. | The user can place bids. |
| User opens auction when has not started or has expired | System checks whether the auction has started and not yet expired. | The user cannot place bids. |

# Exercise

- Read the specifications and discuss the input space. Define your input test cases

# Examples

- **Observation:** a DB system requires robust failure recovery in case of power loss TC specification: Removing power at certain critical point in processing queries
- **Finite State Machine:** If the system is described as a control flow graph, a test case specification can be a selection of feasible execution paths



# Structural Testing (White-box Testing)

- Test cases are derived from the structure of the code. Example: statement coverage

```
1. public static void String collapseSpaces(String argStr){
2.      char last = argSrt.charAt(0);
3.      StringBuffer argBuf = new StringBuffer();
4.      for(int cldx=0; cldx<argStr.length(), clds++){
5.             char ch = argStr.charAt(cldx);
6.             if(ch!= ' ' || last!= ' '){
7.                   argBuf.append(ch)
8.                   last=ch;
9.              }
10.     }
11.}
```

# Structural Testing (White-box Testing)

- Test case specification for **general rules**: Example: Empty string must be tested
- Test case specification for **conditions**: Example: test the two conditions of the if -clause separately

# Fault-base Testing

- Test cases are derived from reported faults
- Example
  - **Reported:** Race condition experienced in multi threads
  - **Test case specification:** test for synchronisation in multi threads

# Exercise

**Deadlock:** Error condition of the system due to two programs or tools waiting of each other signal

```
Date: 2003-10-28 19:37
Sender: o_sukhodolsky
Logged In: YES
user_id=746148

This is deadlock. The situation is as follows.
TestResult calls TestListeners while keeps lock on itself
(methods
addError() and addFailure() are synchronized). After that
BaseTestRunner tries to get lock on the runner
(junit.swingui.TestRunner).

But on other thread BaseTestRunner.endTest() gets lock on
the runner
(swingui), then calls TestRunner.testEnded(), which, in turn,
calls
TestRunner.synchUI() which call SwingUtilities.invokeAndWait
() with
empty Runnable (I'm not sure why it does this, but it does).
So, this
thread waits event dispatch thread (EDT), and at this time on
EDT
TestRunner calls TestResult.runCount() which tries to get lock
on
itself, which already obtained by TestListener.add
(Error|Failure).

So, we have a deadlock.  To fix it it's enought do not call
TestListeners under TestResult lock.
I would recomend to remove invokeAndWait() too, but I'm not
sure why
it's used.
```

# Example of fault-base Testing

- Fault seeding testing
- Mutation Testing

# Fault Seeding Testing

- Fault-seeding testing is fault-base testing that deliberately **seeds faults** and **define test case specifications to test them**

# Fault Seeding Testing

- Let **S** is the total number of seeded faults, and **s(t)** is the number of seeded faults that have been discovered at time t.
  - **s(t)/S** is the *seed-discovery effectiveness* of testing to time t.

# Issues

- Inserting faults into software involves the obvious risk of leaving them there
- Thus, faults injected are "typical" faults
- Not awakes a powerful technique to discover as-yet undiscovered faults

1. Injects fault and pass this code to another group

2. Design test cases to discover the seeded faults

3. Compute the *seed-discovery effectiveness*

```
package it.unibz;

import java.io.*;
/**
 * This is the class we use to try shell commands. The class has a logic
error
 * @author Barbara Russo
 */
public class Dates {
      /**
       * @param month the month number in a year
       * @return  the length of a month in a  year, no bissextile
       */
      public static int daysInMonth (int month) {
          if ((month == 9) || (month == 4) || (month == 6) || (month == 11))
{
            return 30;
          }
          else if (month == 2)
            return 28;
          else return 31;
      }
      public static void main (String[] args) {
          int someMonth, someDay;
          int laterMonth, laterDay;
          int aMonth;
          someMonth = Integer.parseInt(args[0]);
          someDay = Integer.parseInt(args[1]);
          laterMonth = Integer.parseInt(args[2]);
          laterDay = Integer.parseInt(args[3]);
          /* Used to record what day in the year the first day  */
          /* of someMonth and laterMonth are. */
          int someDayInYear = 0;
          int laterDayInYear = 0;
          for (aMonth = 1; aMonth < someMonth; aMonth = aMonth + 1) {
              someDayInYear = someDayInYear + daysInMonth(aMonth);
          }
          for (aMonth = 1; aMonth < laterMonth; aMonth = aMonth + 1) {
              laterDayInYear = laterDayInYear + daysInMonth(aMonth);
          }
          /* The answer */
          int daysBetween = 0;
          System.out.println("The difference in days between " +
                             someMonth + "/" + someDay + " and " +
                             laterMonth + "/" + laterDay + " is: ");
          daysBetween = laterDayInYear - someDayInYear;
          daysBetween = daysBetween + laterDay - someDay;
          System.out.println(daysBetween);

      }
}
```

# Mutation Testing

- Fault mutation is a **fault-base testing** technique that mutates the original code
- Each atomic change is called **mutant**. Each mutant injects one fault
  - It creates a test case per mutant
  - If a mutant fails any test, then it is said to be **killed**
  - All mutants that are not killed are said to remain **live at this point**

# Example of Mutation Operators

They seem naive, tell when this changes are critical

| *Mutation Operator* | *Original Code* | *Mutated Code* |
|---|---|---|
| ***Add 1*** | ***q=0*** | ***q=1*** |
| ***Replace Variable*** | ***r=x*** | ***r=y*** |
| ***Replace Operator*** | ***q=q+1*** | ***q=q-1*** |

# Test Case Adequacy

Barbara Russo

SwSE - Software and Systems Engineering research group

unibz
Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

# Test Case Adequacy

- **Ideally**, adequacy means that test cases show **correctness** of a product
- In practice, we can only approximate the problem by setting a set of adequacy criteria and we can only discuss whether a set of test cases is **inadequate against such criteria**
- **If this happens, we can extend the test cases**

# Test Case Adequacy

- **Adequacy criterion:** a predicate that is TRUE/ FALSE on a pair <Program, Test Suite>
  - Example: a test suite exercises all executable statements

# Test obligation

- **Test obligation:** a partial test specification that checks an adequacy criterion
  - Example: *Execute executable statements within loops*
- An adequacy criterion can be checked by one or more test obligations
  - Example: *Execute all executable statements*

# Test Suite

- A test suite (i.e. a set of test cases) satisfies an adequacy criterion if
  - **all its test cases succeed** and
  - for **every test obligation** of an adequacy criterion, there exists **at least a test case that satisfies it**

# Adequacy degree

- The adequacy degree is the **level of adequacy a test suite** achieves against an adequacy criterion:
- Example: **percentage of statement coverage** for a pair <*myProgram*, *myTestSuite*>
- Example: **ratio of killed total mutants** (K/M) measures the adequacy degree of a test suite in mutation testing

# Test Subsumption

- An *adequacy criterion* A subsumes an adequacy criterion B if **every test suite X that satisfies A contains some test suite Y that satisfies B** (i.e. X also satisfies B)
- Example: Branch coverage subsumes executable statement coverage
- We tend to discard adequacy criteria that are subsumed

# Test Subsumption

- Stronger adequacy criteria can potentially reveal more faults

# Structural Testing (White-box Testing)

- Test cases are derived from the structure of the code

```
1.  public static void String collapseSpaces(String argStr){
2.      char last = argSrt.charAt(0);
3.      StringBuffer argBuf = new StringBuffer();
4.      for(int cldx=0; cldx<argStr.length(), clds++){
5.          char ch = argStr.charAt(cldx);
6.          if(ch!= ' ' || last!= ' '){
7.              argBuf.append(ch)
8.              last=ch;
9.           }
10.    }
11.}
```

Various testing goals: test data structures; test loop structures; test a specific state of the execution; test for extreme/default values

- Input: Your Output: Your

| *last* | *ch* | *argBuf* | *cldx* |
|---|---|---|---|
| *Y* | *Y* | *Y* | *0* |
| *Y* | *o* | *o* | *1* |
| *o* | *u* | *u* | *2* |
| *u* | *r* | *r* | *3* |
| *r* | | | |

- Input: Your Output: Your

- Input: Your Output: Your
- Input: ‘ ’You Output: You

| *last* | *ch* | *argBuf* | *cldx* |
|---|---|---|---|
| *‘ ‘* | *‘ ‘* | *-* | *0* |
| *‘ ‘* | *Y* | *Y* | *1* |
| *Y* | *o* | *o* | *2* |
| *o* | *u* | *u* | *3* |
| *u* | | | |

- Input: Your Output: Your
- Input: ‘ ’You Output: You

- Input: Your Output: Your
- Input: ‘ ’You Output: You
- Input: I am Output: I am

| *last* | *ch* | *argBuf* | *cldx* |
|---|---|---|---|
| ***I*** | ***I*** | ***I*** | ***0*** |
| ***I*** | ***‘’*** | ***‘’*** | ***1*** |
| ***‘’*** | ***a*** | ***a*** | ***2*** |
| ***a*** | ***m*** | ***m*** | ***3*** |
| ***m*** | | | |

- Input: Your Output: Your
- Input: ‘ ’You Output: You
- Input: I am Output: I am
-

| *last* | *ch* | *argBuf* | *cldx* |
|---|---|---|---|
| *I* | *I* | *I* | *0* |
| *I* | *‘ ’* | *‘ ’* | *1* |
| *‘ ’* | *a* | *a* | *2* |
| *a* | *m* | *m* | *3* |
| *m* | | | |

# Example

# Example

- In the code example above, we can satisfy the statement coverage by a test case with any string with no spaces,

# Example

- In the code example above, we can satisfy the statement coverage by a test case with any string with no spaces,
- but if we want to satisfy the if condition (branch coverage) we need to create another test case with a string containing empty characters (to test the true and the false of the branch)

# Example

- If we want to test the for loop at different counter values, we might need to add new test case obligations
- For example, in the above code example, testing for argStr.length()=0 will include a new test case for an empty string. If it is not checked it can cause future failures

# Exercise in class


- 100% statement coverage:
- If 100% of statements are covered by tests then the method is correct

# Exercise

```
[1] int foo (int a, int b, int c, int d, float e) {
[2]    if (a == 0) {
[3]        return 0;
[4]    }
[5]    int x = 0;
[6]    if ( (a==b) II ( (c == d) && bug(a) ) ) {
[7]        x=1;
[8]    }
[9]    e = 1/x;
[10]    return e;
[11] }
```

bug(a) = TRUE if !a==0 else 0

Adequacy criterion: 100% statement coverage

Two test cases: TC(0,0,0,0,0) and TC(1,0,0,0,0)

# Exercise

- Do the selected Test Cases **catch the error**?
- Discuss **accuracy** of the 100% statement coverage technique in this case: FP? FN?
- Which **other coverage technique** subsume the statement one in this case?

# Exercise

- Do the selected Test Cases **catch the error**?
- Discuss **accuracy** of the 100% statement coverage technique in this case: FP? FN?
- Which **other coverage technique** subsume the statement one in this case?

Branch Coverage