PV260 - SOFTWARE QUALITY

PRINCIPLES OF TESTING. REQUIREMENTS & TEST CASES. TEST PLANS & RISK ANALYSIS

Bruno Rossi

brossi@mail.muni.cz

LAB OF SOFTWARE ARCHITECTURES
AND INFORMATION SYSTEMS

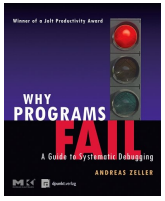
FACULTY OF INFORMATICS
MASARYK UNIVERSITY, BRNO

"Discovering the unexpected is more important than confirming the known."

George Box

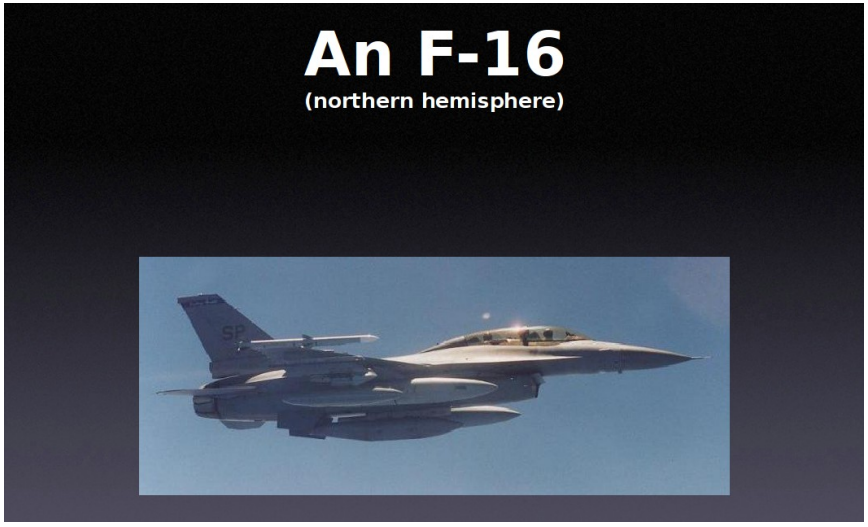
Introduction

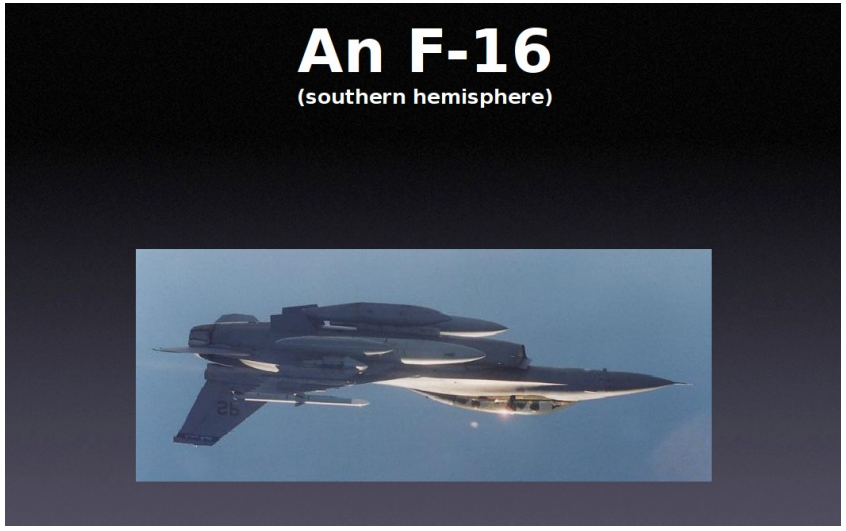
- In Eclipse and Mozilla, 30–40% of all changes are fixes (Sliverski et al., 2005)
- Fixes are 2–3 times smaller than other changes (Mockus +Votta, 2000)
- 4% of all one-line changes introduce new errors (Purushothaman + Perry, 2004)

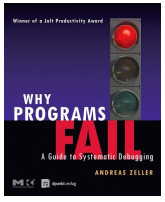


A. Zeller, Why Programs Fail, Second Edition: A Guide to Systematic Debugging, 2 edition. Amsterdam ; Boston: Morgan Kaufmann, 2009.

Motivating examples







A. Zeller, Why Programs Fail, Second Edition: A Guide to Systematic Debugging, 2 edition. Amsterdam ; Boston: Morgan Kaufmann, 2009.

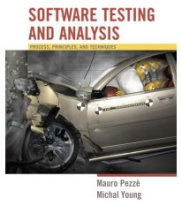
Example: a Memory Leak

Apache web server, version 2.0.48

Response to normal page request on secure (https) port

```
Static void ssl_io_filter_disable(ap_filter_t *f)
{   bio_filter_in_ctx_t *inctx = f->ctx;

    inctx->ssl = NULL;
    inctx->filter ctx->pssl = NULL;
}
```

No obvious error, but Apache leaked memory slowly (in normal use) or quickly (if exploited for a DOS attack)



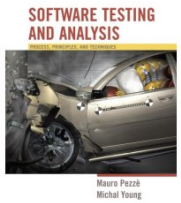
Example: a Memory Leak

Apache web server, version 2.0.48

Response to normal page request on secure (https) port

```
Static void ssl_io_filter_disable(ap_filter_t *f)
{   bio_filter_in_ctx_t *inctx = f->ctx;
    SSL_free(inctx -> ssl);
    inctx->ssl = NULL;
    inctx->filter ctx->pssl = NULL;
}
```

The missing code is for a **structure defined and created elsewhere**, accessed through an opaque pointer.



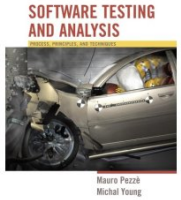
Example: a Memory Leak

Apache web server, version 2.0.48

Response to normal page request on secure (https) port

```
Static void ssl_io_filter_disable(ap_filter_t *f)
{   bio_filter_in_ctx_t *inctx = f->ctx;
    SSL_free(inctx -> ssl);
    inctx->ssl = NULL;
    inctx->filter_ctx->pssl = NULL;
}
```

Almost impossible to find with unit testing. (Inspection and some dynamic techniques could have found it)



What is Software Testing

- *"Testing is the **process** of **exercising or evaluating** a system or system component by manual or automated means to verify that it **satisfies specified requirements**."* IEEE standards definition

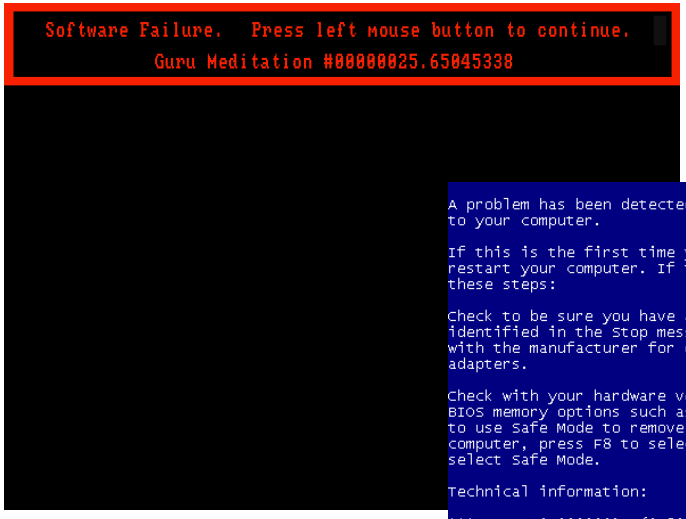
Software Testing – Important Terms

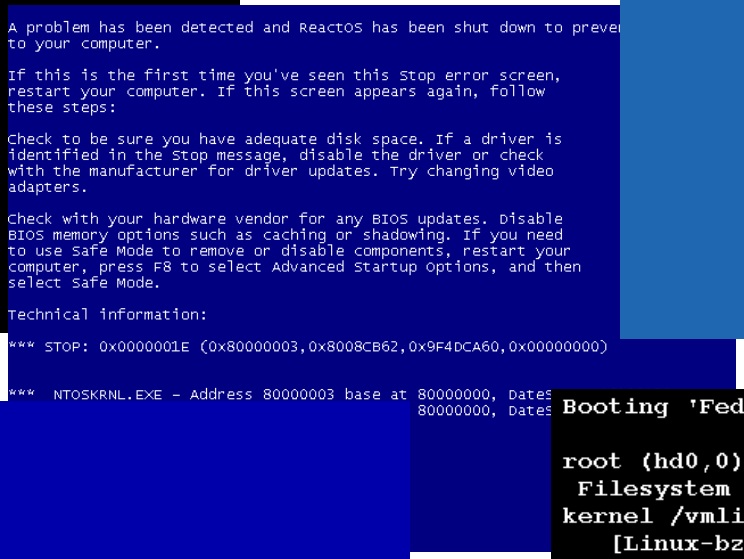
- **Defect:** *"An **imperfection** or **deficiency** in a work product where that work product **does not meet its requirements** or **specifications** and needs to be either repaired or replaced."*
- **Error**: *"A **human action** that produces an incorrect result"*
- **Failure**: *"(A) **Termination** of the ability of a product to perform a required function or its inability to perform within previously specified limits.*
  *(B) **An event** in which a system or system component does not perform a required function within specified limits.*

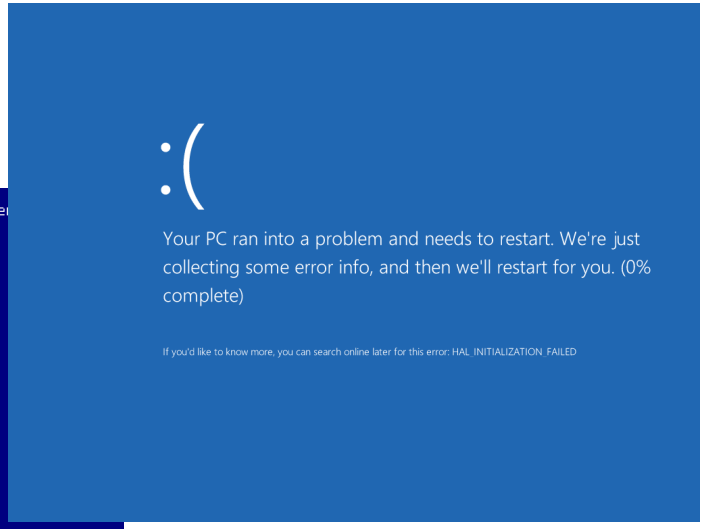
  → A failure may be produced when a fault is encountered. "
- **Fault**: *"A **manifestation** of an error in software."*
- **Problem**: *"(A) **Difficulty** or uncertainty experienced by one or more persons, resulting from an unsatisfactory encounter with a system in use.*
  *(B) A **negative situation** to overcome"*

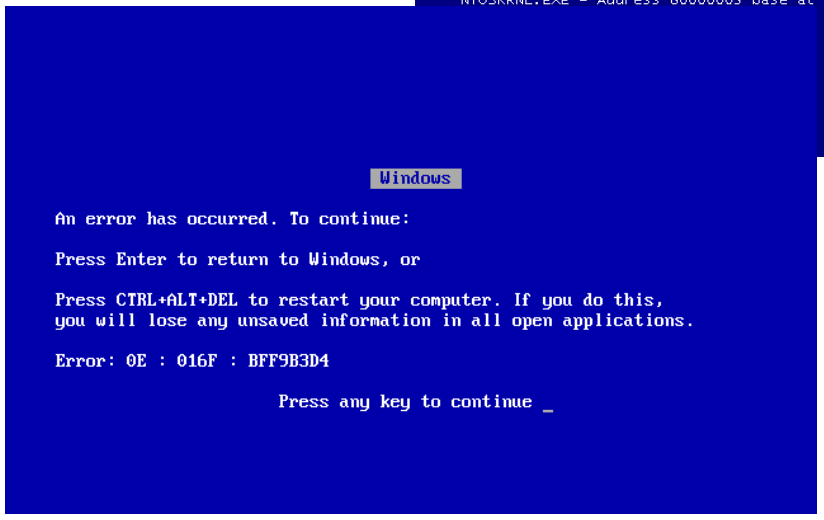
Definitions according to IEEE Std 1044-2009 "IEEE Standard Classification for Software Anomalies"

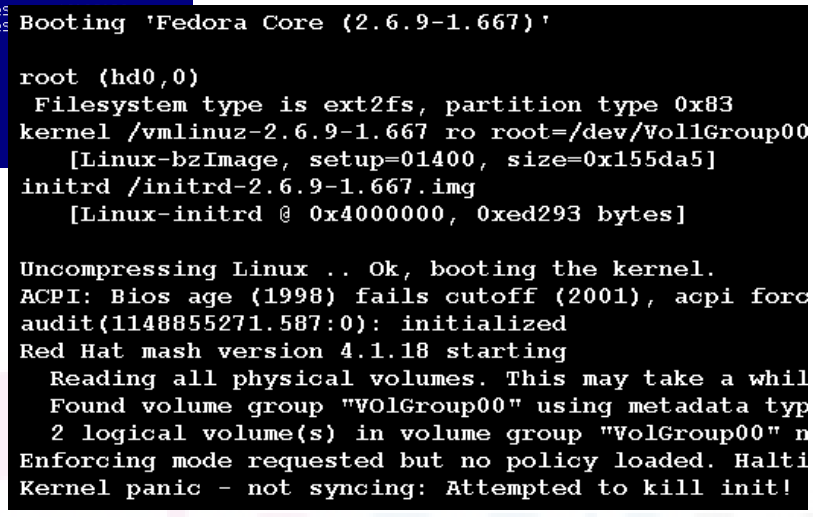
Hopefully you have not seen some of these…











…or some of these

YouTube

500 Internal Server Error

Sorry, something went wrong.

A team of highly trained monkeys has been dispatched to deal with this situation.

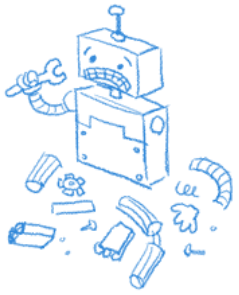
If you see them, show them this information:


```
AB38WEPIDWfs5FLs3YWvAJbHZzGGd1X3seRUSOX7Kh9K1gde_FLVY4GDBjkn
8jPuyamICiGBZExjMpiZT4j7rx-0NZ707H-cPNSEbJ0n_b7MYf692YtZtrQI
DsAGxZ38bYUMy4UyGJHtGSUG4N0BuXXX35-jWJZDtkJoj_ZNdJoOTOJSG2PC
X_mCxpP5lQi7-rZUcx83I33yavfWr2WcE4EUyS0TyqzFqzh_QJVNbc7_yxRH
8udCCKkxQVBdsBDK2qejBUTemZ31SFOWC10wUulgiE-L750WxOmGjsP2GiSp
6Z3-0IepREkPtU649pzpZ6PBIqWlBXOZ8GnoQIiAiqqOcneErAHFs0aCNi9-
tB34vRO8oFi_JtZ4AzvPEVIpgLjaAs_PwERN2NRADOPVqrtEPbUGZh-c7PdZ
```


Google

404. That's an error.

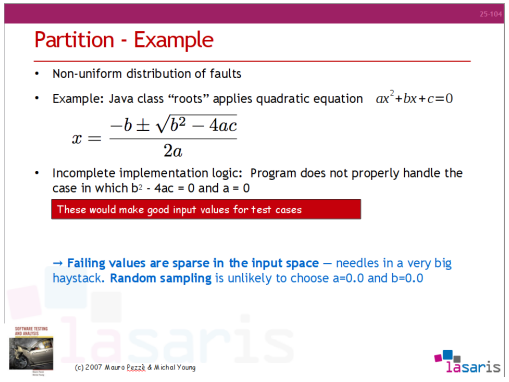
The requested URL /intl/en/options/ was not found on this server. That's all we know.



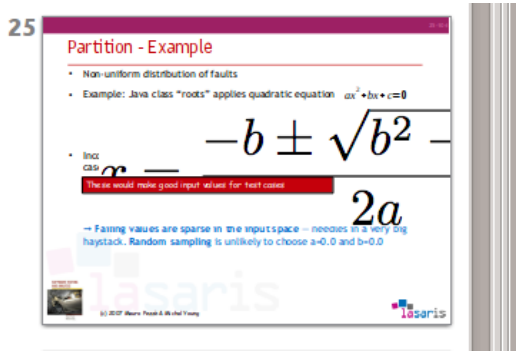
Defects are omnipresent

This is one failure I encountered when preparing a presentation some years ago on *LibreOffice 4.2.7.2*

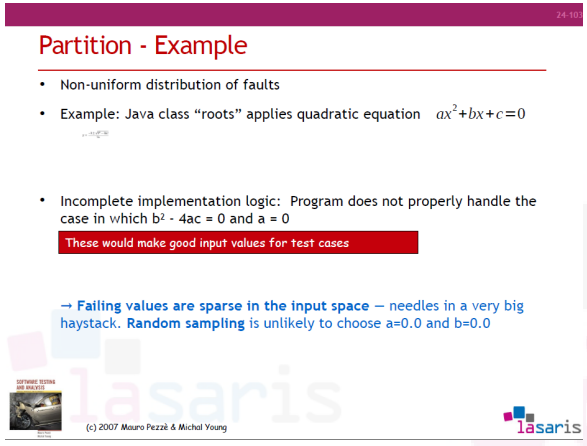
A formula in ppt that got converted into image – looks good when editing



The slides preview on the left, looks a bit strange...



When converted to pdf...



What about the term “Bug”?

- Very often a **synonymous of *“defect”*** so that ***“debugging”* is the activity related to removing defects in code**

  However:

  → **it may lead to confusion**: it is not rare the case in which *“bug”* is used in natural language to refer to different levels:

  “this line is buggy” - “this pointer being null, is a bug” - “the program crashed: it's a bug”

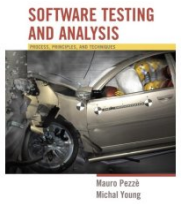
  → starting from Dijkstra, there was the search for terms that could **increase the responsibility of developers** – the term *“bug”* might give the impression of something that *magically* appears into software

Definitions according to IEEE Std 1044-2009 “IEEE Standard Classification for Software Anomalies“

Basic Principles of Software Testing

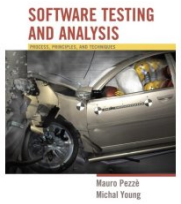
Basic Principles of Testing

- **Sensitivity:** better to fail every time than sometimes
- **Redundancy:** making intentions explicit
- **Restrictions:** making the problem easier
- **Partition:** divide and conquer
- **Visibility:** making information accessible
- **Feedback:** applying lessons from experience in process and techniques



Sensitivity: better to fail every time than sometimes

- Consistency helps:
  - a test selection criterion works better if every selected test provides the same result, i.e., **if the program fails with one of the selected tests, it fails with all of them (reliable criteria)**
  - run time deadlock analysis works better if it is machine independent, i.e., if the program deadlocks when analyzed on one machine, it deadlocks on every machine

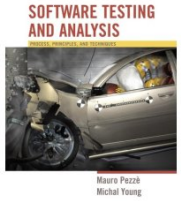


Sensitivity: better to fail every time than sometimes

- Look at the following code fragment

```
char before[] = "=Before=";
char middle[] = "Middle";
char after [] = "=After=";

int main(int argc, char *argv){

   strcpy(middle, "Muddled"); /* fault, may not fail */
   strncpy(middle, "Muddled", sizeof(middle)); /* fault, may not fail */


}
```

What's the problem?

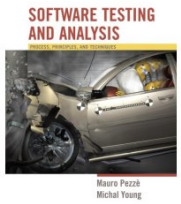


Sensitivity Example

- Let's make the following adjustment

```
char before[] = "=Before=";
char middle[] = "Middle";
char after [] = "=After=";

int main(int argc, char *argv){

   strcpy(middle, "Muddled"); /* fault, may not fail */
   strncpy(middle, "Muddled", sizeof(middle)); /* fault, may not fail */
   stringcpy(middle, "Muddled", sizeof(middle)); /* guaranteed to fail */

}

void stringcpy(char *target, const char *source, int size){
   assert(strlen(source) < size);
   strcpy(target, source);
}
```

This adds sensitivity to a non-sensitive solution



Sensitivity Example

- Let's look at the following Java code fragment. We use the ArrayList as a sort of queue and we remove one item after printing the results

```
public class TestIterator {

    public static void main(String args[]) {

        List<String> myList = new ArrayList<>();

        myList.add("PV260");
        myList.add("SW");
        myList.add("Quality");

        Iterator<String> it = myList.iterator();
        while (it.hasNext()) {
            String value = it.next();
            System.out.println(value);
            myList.remove(value);
        }
    }
}
```

Will this output
"PV260
SW
Quality" ?

Sensitivity Example

- Let's look at the following Java code fragment. We use the ArrayList as a sort of queue and we remove one item after printing the results

```
public class TestIterator {

    public static void main(String args[]) {

        List<String> myList = new ArrayList<>();

        myList.add("PV260");
        myList.add("SW");
        myList.add("Quality");

        Iterator<String> it = myList.iterator();
        while (it.hasNext()) {
            String value = it.next();
            System.out.println(value);
            myList.remove(value);
        }
    }
}
```

Actually, this throws java.util.ConcurrentModificationException

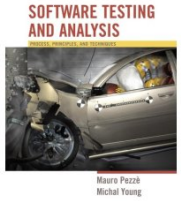
Sensitivity Example

- From Java SE documentation:

- "[...] Some Iterator implementations (including those of all the general purpose collection implementations provided by the JRE) may choose to throw this exception if this behavior is detected. **Iterators that do this are known as *fail-fast* iterators, as they fail quickly and cleanly, rather that risking arbitrary, non-deterministic behavior at an undetermined time in the future.**"
- "Note that *fail-fast* behavior cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of **unsynchronized concurrent modification**. *Fail-fast* operations throw *ConcurrentModificationException* on a best-effort basis. **Therefore, it would be wrong to write a program that depended on this exception for its correctness: *ConcurrentModificationException should be used only to detect bugs*.**"

Redundancy: making intentions explicit

- Redundant checks can increase the capabilities of catching specific faults early or more efficiently.
  - **Static type checking** is redundant with respect to dynamic type checking, but it can reveal many type mismatches earlier and more efficiently.
  - **Validation of requirement specifications** is redundant with respect to validation of the final software, but can reveal errors earlier and more efficiently.
  - **Testing and proof of properties are redundant**, but are often used together to increase confidence



Redundancy Example

- Adding redundancy by asserting that a condition must always be true for the correct execution of the program

```
void save(File *file, const char *dest){
   assert(this.isInitialized());
   ...
}
```

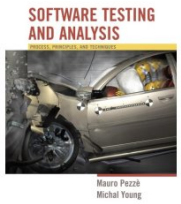
- From a language (e.g. Java) point of view, think about declarations of thrown exceptions from a method

```
public void throwException() throws FileNotFoundException{
    throw new FileNotFoundException();
}
```

Think if you could throw any exception from a method without declaration in the method signature

Restriction: making the problem easier

- Suitable restrictions can reduce hard (unsolvable) problems to simpler (solvable) problems
  - **A weaker spec may be easier to check**: it is impossible (in general) to show that pointers are used correctly, but the simple Java requirement that pointers are initialized before use is simple to enforce.
  - **A stronger spec may be easier to check**: it is impossible (in general) to show that type errors do not occur at run-time in a dynamically typed language, but statically typed languages impose stronger restrictions that are easily checkable.



Restriction Example

- Will the following compile in Java?

```
public static void questionable(){
    int k;
    for (int i=0; i<10;++i){
        if (someCondition(i)){
            k = 0;
        } else {
            k+=i;
        }
    }
}
```

Java ALWAYS enforces variable initialization before usage as the following example shows – this is a case of restriction

```
    int k;

    if (true == false){
        k+=i;
    }
```

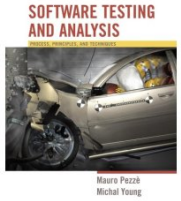
But restrictions can be applied at different levels, e.g. at the architectural level the decision of making the HTTP protocol stateless hugely simplified testing (and as such made the protocol more robust)

Partition: Divide & Conquer

- Hard testing and verification problems can be handled by **suitably partitioning the input space**:
  - both **structural** (**white box**) and **functional test** (**black box**) selection criteria identify suitable partitions of code or specifications (partitions drive the sampling of the input space)
  - **verification** techniques fold the input space according to specific characteristics, grouping homogeneous data together and determining partitions

→ Examples of **structural** (**white box**) techniques: ***unit testing***, ***integration testing***, ***performance testing***

→ Examples of **functional** (**black box**) techniques: ***system testing***, ***acceptance testing***, ***regression testing***



Partition Example

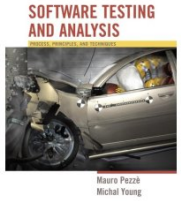
- Non-uniform distribution of faults
- Example: Java class "roots" applies quadratic equation $ax^2+bx+c=0$

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

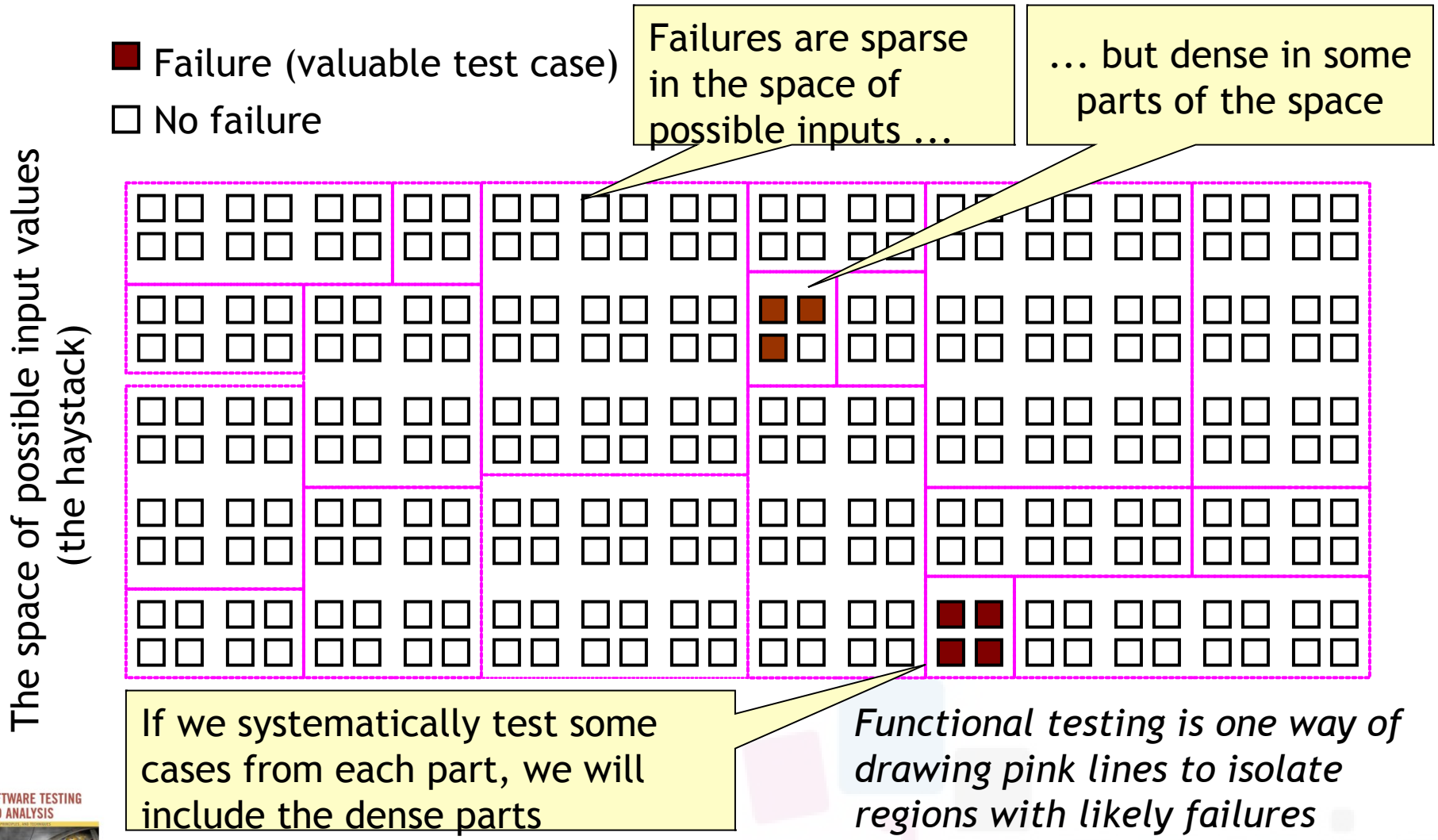
- Incomplete implementation logic: Program does not properly handle the case in which $b^2 - 4ac = 0$ and $a = 0$

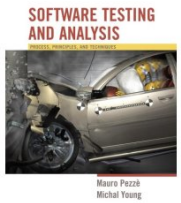
These would make good input values for test cases

→ **Failing values are sparse in the input space** — needles in a very big haystack. **Random sampling** is unlikely to choose a=0.0 and b=0.0



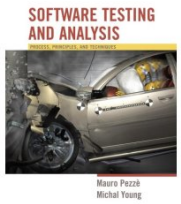
Partition Example





Visibility: Judging Status

- The ability to **measure progress** or **status against goals**
    - X visibility = ability to judge how we are doing on X, e.g., schedule visibility = “Are we ahead or behind schedule,” quality visibility = “Does quality meet our objectives?”
  - Involves setting goals that can be assessed at each stage of development
    - The biggest challenge is early assessment, e.g., assessing specifications and design with respect to product quality
- Related to **observability**
  - Example: *Choosing a simple or standard internal data format to facilitate unit testing*



Visibility Example

- The HTTP Protocol

```
GET /index.html HTTP/1.1
Host: www.google.com
```

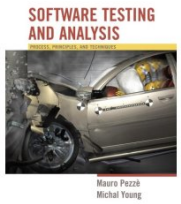
Why wasn't a more efficient binary format selected?

To note HTTP 2.0 **will** use a binary format
(from https://http2.github.io/faq):
"Binary protocols are more efficient to parse, more compact "on the wire", and most importantly, they are much less error-prone, compared to textual protocols like HTTP/1.x, because they often have a number of affordances to "help" with things like whitespace handling, capitalization, line endings, blank links and so on."
In fact, reduction of visibility is confirmed by
"It's true that HTTP/2 isn't usable through telnet, but we already have some tool support, such as a Wireshark plugin."

Feedback: tuning the development process

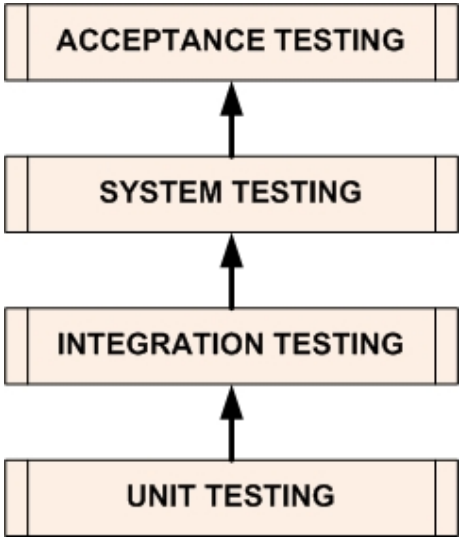
- **Learning from experience**: Each project provides information to improve the next
- **Examples**
  - Checklists are built on the basis of errors revealed in the past
  - Error taxonomies can help in building better test selection criteria
  - Design guidelines can avoid common pitfalls

Using a software reliability model fitting past project data
Looking for problematic modules based on prior knowledge



Testing Levels

Testing Levels (1/2)



A system is tested for acceptability. Aim: **evaluate the system's compliance with the business requirements and ready for delivery**.

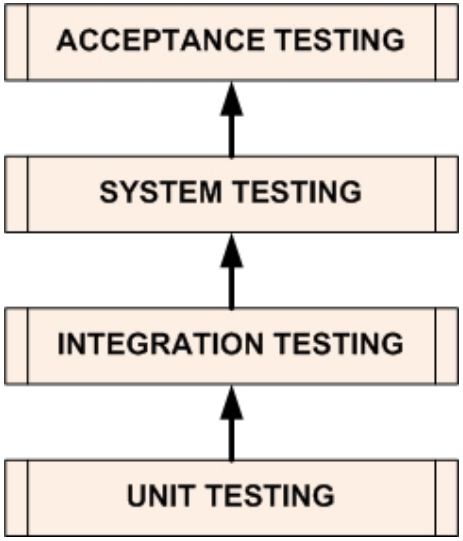
A complete, integrated system/software is tested. Aim: **evaluate the system's compliance with the specified requirements**

Individual units are combined and tested as a group. Aim: **expose faults in the interaction between integrated units**

A level of the software testing process where individual units/components of a software/system are tested – **Validate that each unit performs as designed**

http://softwaretestingfundamentals.com/software-testing-levels/

Testing Levels (2/2)

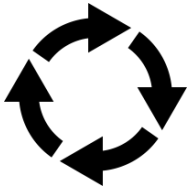


BLACK BOX TESTING

BLACK BOX TESTING

BLACK BOX / WHITE BOX TESTING

WHITE BOX TESTING



! Test Plans / Test cases are created *for each* level!

http://softwaretestingfundamentals.com/software-testing-levels/

Example Acceptance Testing Automation

Acceptance Tests Automation (1/4)

Using Fitnesse to write acceptance tests so that the customer can actually write the acceptance conditions for the software

Looking at our previous example the “root” case

$$ax^2+bx+c=0$$

That we solve by means of

$$x=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

FitNesse

Acceptance Tests Automation (2/4)

```
public class Root {
    double rootOne, rootTwo;
    int numRoots;
    public Root (double a, double b, double c){
       double q;
       double r;
       q = b*b - 4 * a *c;
       if (q >0 && a != 0){
           // if b^2 > 4ac there are two dinstict roots
           numRoots = 2;
           r = (double) Math.sqrt(q);
           rootOne = ((0-b) + r) / (2*a);
           rootTwo = ((0-b) - r) / (2*a);
       } else if (q==0){  // DEFECT HERE
           numRoots = 1;
           rootOne = (0-b)/(2*a);
           rootTwo = rootOne;
       }else {
           // equation had no roots if b^2<4ac
           numRoots = 0;
           rootOne  = -1;
           rootTwo  = -1;
       }
    }
}
```

Source code from Mauro Pezzè & Michal Young

FitNesse

Acceptance Tests Automation (3/4)

Our first attempt returns the number of solutions, but **the customer did not want only this** – so this is a mistake we would not have captured with unit tests

cz.muni.pv260.RootFixture			
a	b	c	runRoot?
1	25	2	2
3	25	3	2
4	2	4	0
16	2	12	0
1	2	1	1

The customer **also wanted the solutions to the equation**, however this opens other discussions → how should we deal with no solutions? What about imaginary numbers?

cz.muni.pv260.RootFixture					
a	b	c	runRoot?	getRootOne?	getRootTwo?
1	25	2	2	**-0.08025765162577869**<=-0.08	**-24.91974234837422**<=-24.91
3	25	3	2	**-0.12177963349613445**<=-0.12	**-8.211553699837198**<=-8.21
4	2	4	0	-1.0	-1.0
16	2	12	0	-1.0	-1.0
1	2	1	1	-1.0	-1.0

FitNesse

Acceptance Tests Automation (4/4)

Running with *a=0* reports the mistake and also opens up a discussion about the format for returning the solutions and what were the original requirements in these cases

cz.muni.pv260.RootFixture

a	b	c	runRoot?
1	25	2	2
3	25	3	2
4	2	4	0
16	2	12	0
1	2	1	1
0	0	2	0

```
java.lang.ArithmeticException: / by zero
        at cz.muni.pv260.Root.(Root.java:18)
        at cz.muni.pv260.RootFixture.runRoot(RootFixture.java:24)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcce
        at java.lang.reflect.Method.invoke(Method.java:606)
        at fit.TypeAdapter.invoke(TypeAdapter.java:108)
        at fit.TypeAdapter.get(TypeAdapter.java:97)
        at fit.Fixture$CellComparator.compareCellToResult(Fixture.java:374)
        at fit.Fixture$CellComparator.access$100(Fixture.java:360)
        at fit.Fixture.compareCellToResult(Fixture.java:302)
        at fit.Fixture.check(Fixture.java:298)
        at fit.ColumnFixture.check(ColumnFixture.java:54)
        at fit.Binding$QueryBinding.doCell(Binding.java:218)
        at fit.ColumnFixture.doCell(ColumnFixture.java:40)
        at fit.Fixture.doCells(Fixture.java:174)
        at fit.Fixture.doRow(Fixture.java:168)
        at fit.ColumnFixture.doRow(ColumnFixture.java:27)
        at fit.Fixture.doRows(Fixture.java:162)
        at fit.ColumnFixture.doRows(ColumnFixture.java:19)
        at fit.Fixture.doTable(Fixture.java:156)
        at fit.Fixture.interpretTables(Fixture.java:101)
        at fit.Fixture.doTables(Fixture.java:81)
        at fit.FitServer.process(FitServer.java:81)
        at fit.FitServer.run(FitServer.java:56)
        at fit.FitServer.main(FitServer.java:41)
```

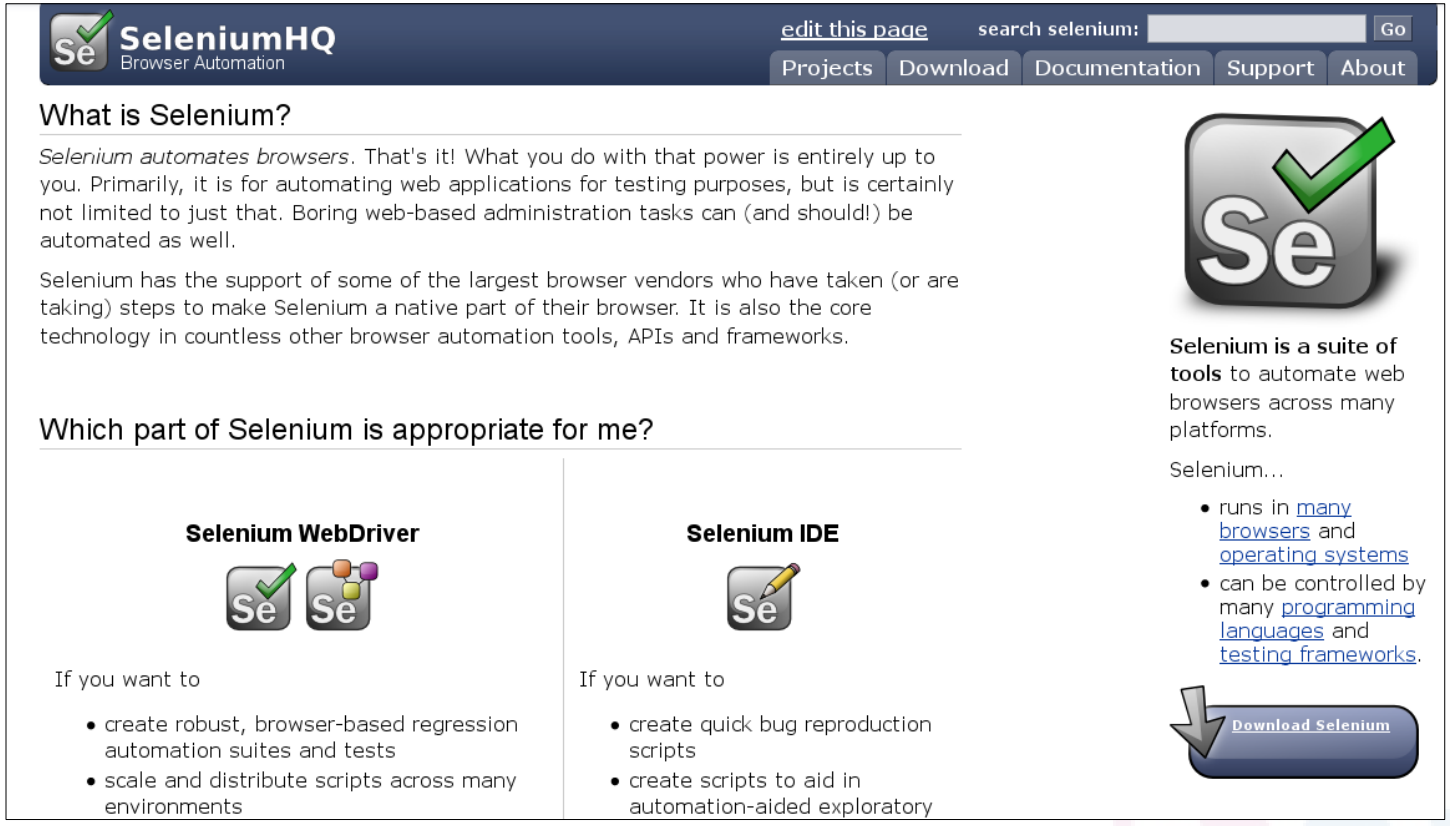
cz.muni.pv260.RootFixture

a	b	c	runRoot?	getRootOne?	getRootTwo?
1	25	2	2	**-0.08025765162577869**<=-0.08	**-24.91974234837422**<=-24.91
3	25	3	2	**-0.12177963349613445**<=-0.12	**-8.211553699837198**<=-8.21
4	2	4	0	-1.0	-1.0
16	2	12	0	-1.0	-1.0
1	2	1	1	-1.0	-1.0
0	0	2	0 *expected* 1 *actual*	-1.0 *expected* NaN *actual*	-1.0 *expected* NaN *actual*

FitNesse

Acceptance Tests Automation

Other frameworks are available for automation of acceptance testing, like Selenium (https://www.seleniumhq.org) for web-based acceptance testing (that can also be integrated with Fitnesse)



Quality of Software Tests – Mutation Testing

Estimating Software Test Suite Quality

- What if we could judge the effectiveness of a test suite in finding real faults, by measuring how well it finds **seeded fake faults**?
- How can **seeded faults** be representative of real defects?

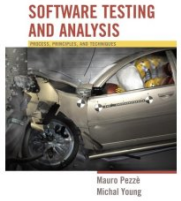
Example: I add **100 new defects** to my application

- they are exactly like real bugs in every way
- I make 100 copies of my program, each with one of my 100 new bugs

I run my test suite on the programs with seeded bugs ...

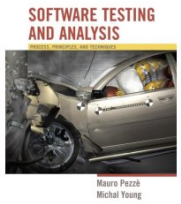
- ... and the tests reveal 20 of the bugs
- (the other 80 program copies do not fail)

→ **What can I infer about my test suite?**



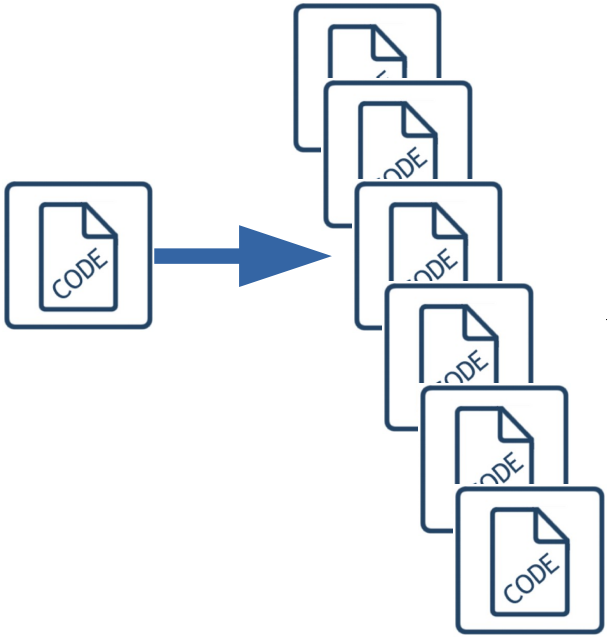
Mutation Testing Assumptions

- Competent programmer hypothesis:
  - Programs are “nearly” correct
    - Real faults are **small variations from the correct program**
    - → Mutants are reasonable models of real buggy programs
- Coupling effect hypothesis:
  - Tests that find **simple faults also find more complex faults**
    - Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults too



How Mutation Testing works (1/3)

- Create many **modified copies** of the original program called **mutants** Each mutant with a single variation from the original program.



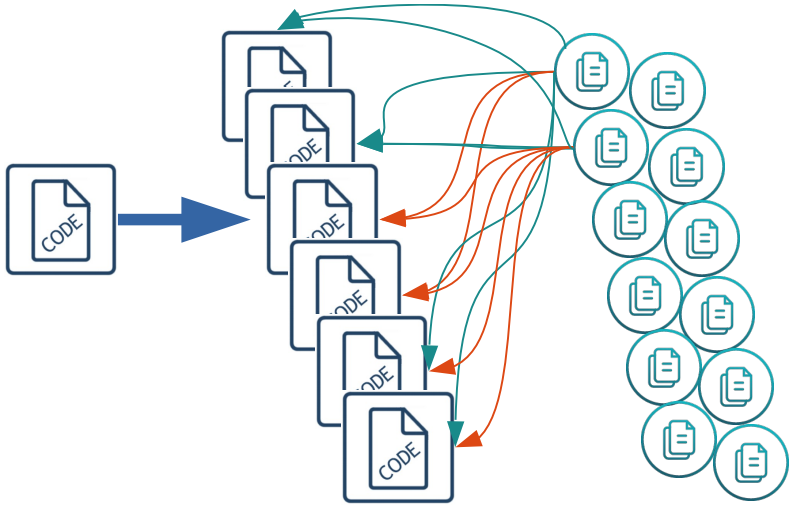
- Mutation Process: application of **mutation operators**, such as statement deletions, statement modifications (e.g. != instead of ==)

Algorithm 1: Original Code	**Algorithm 2:** Mutated Code

if *(a == b)* **then**
`// do something`
else
`// do something`
	if *(a != b)* **then**
	`// do something`
	else
	`// do something`

How Mutation Testing works (2/3)

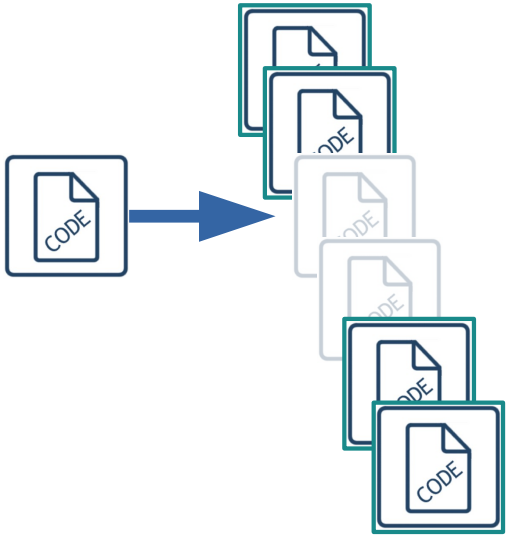
- All **mutants** are then tested by test suites to get the percentage of mutants failing the tests



- The **failure** of mutants is expected!
- If mutants **do not cause tests to fail**, they are considered **live mutants**

How Mutation Testing works (3/3)

- All **mutants** are then tested by test suites to get the percentage of mutants failing the tests



- The number of **live mutants** can be a sign of:
  - i) **tests are not sensitive enough** to catch the modified code
  - ii) there are **equivalent mutants**

e.g. original program

```
if (x==2 && y==2){
    int z = x+y;
}
```

equiv mutant

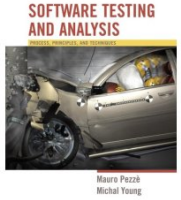
```
if (x==2 && y==2){
  int z = x*y;
}
```

Mutation Score as indication of the tests quality:

$$M_{Score} = \frac{M_{killed}}{M_{tot} - M_{eq}}$$

Mutation Operators

- Syntactic change from legal program to legal program
  - Specific to each programming language. C++ mutations don't work for Java, Java mutations don't work for Python
- Examples:
  - crp: constant for constant replacement
    - for instance: from (x < 5) to (x < 12)
    - select from constants found somewhere in program text
  - ror: relational operator replacement
    - for instance: from (x <= 5) to (x < 5)
  - vie: variable initialization elimination
    - change int x =5; to int x;



Problems of Mutation Testing

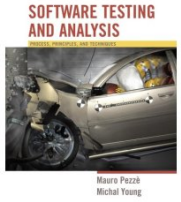
- Mutation testing has not yet widely adopted for a series of reasons, mainly:
  - **Performance** reasons
  - The **equivalent mutants** problem
  - Missing **integration tools**
  - **Benefits** might not be immediately clear

$$M_{Score} = \frac{M_{killed}}{M_{tot} - M_{eq}}$$

Equivalent mutants problem: determining **syntactically different** but **semantically equal** mutant is **undecidable**

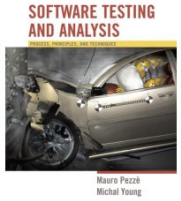
Weak Mutation

- Problem: There are lots of mutants. Running each test case to completion on every mutant is expensive
  - Number of mutants grows with the square of program size
- Approach:
  - Execute meta-mutant (with many seeded faults) together with original program
  - Mark a seeded fault as “killed” as soon as a difference in intermediate state is found
    - Without waiting for program completion
    - Restart with new mutant selection after each “kill”



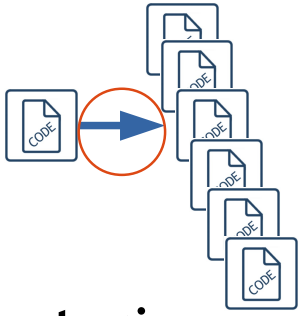
Statistical Mutation

- Problem: There are lots of mutants. Running each test case on every mutant is expensive
  - It's just too expensive to create N^2 mutants for a program of N lines (even if we don't run each test case separately to completion)
- Approach: Just create a random sample of mutants
  - May be just as good for *assessing* a test suite
    - Provided we don't design test cases to kill particular mutants

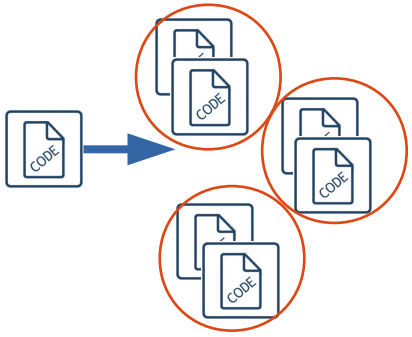


Other Optimization Approaches

- **Selective mutation**: reduce the number of active operators selecting only the most efficient operators → produce mutants not easy-to-kill



- **Second Order Strategies**: combining more than a single mutation, putting together First Order Mutants (different sub-strategies to combine them)



Sample Demo with PiTest

Real world mutation testing

PIT is a state of the art **mutation testing** system, providing **gold standard test coverage** for Java and the jvm. It's fast, scalable and integrates with modern test and build tooling.

Get Started

From Requirements to Test Cases

Test Case Definition

According to ISO/IEC/IEEE 29119:

- **Test Case Specification**: "(A) A set of **test inputs**, **execution conditions**, and **expected results** developed for a particular objective, such as to **exercise a particular program path** or to **verify compliance with a specific requirement**.
(B) A document specifying **inputs**, **predicted results**, and a **set of execution conditions** for a **test item**"

Example:

1. Open the browser
2. Go to shopping cart page (pre-conditions: user is logged-in, no items are in the shopping cart, the check-out button is not available)
3. Add item "x" → exp result: i) the page is updated with the new item, ii) the check-out button becomes available
4. Remove item "x" → exp result: i) no items are listed, ii) the check-out button is not available

Test Plan Definition

According to the International Software Testing Qualifications Board (ISTQB):

- "A **document** describing the **scope**, **approach**, **resources** and **schedule** of **intended test activities**. It identifies amongst others **test items**, the **features to be tested**, the **testing tasks**, **who will do each task**, degree of **tester independence**, the **test environment**, the **test design techniques** and **entry** and **exit criteria** to be used, and the **rationale** for their choice,and any **risks** requiring contingency planning. It is a record of the test planning process."

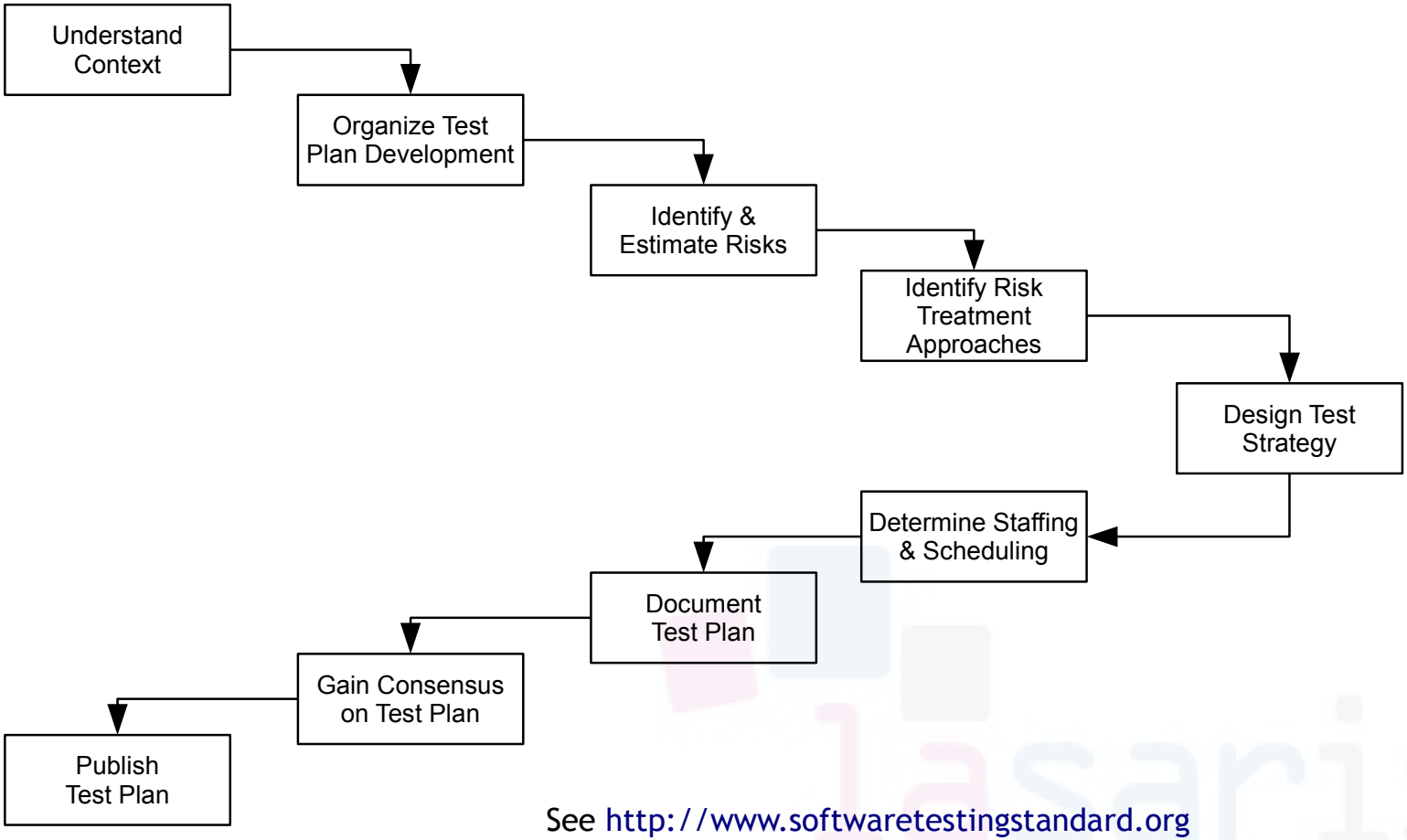
http://softwaretestingfundamentals.com/test-plan

Risk-based Testing

- It is *__not feasible__* to test everything in a software system
- We need some ways to prioritize which parts to test more thoroughly
  - One way is to use the so-called risk-based testing: **prioritizing test cases based on risks**
  - This is a **business-driven** decision based on the possible damage that a defect may cause

Motivation: ISO/IEC/IEEE 29119

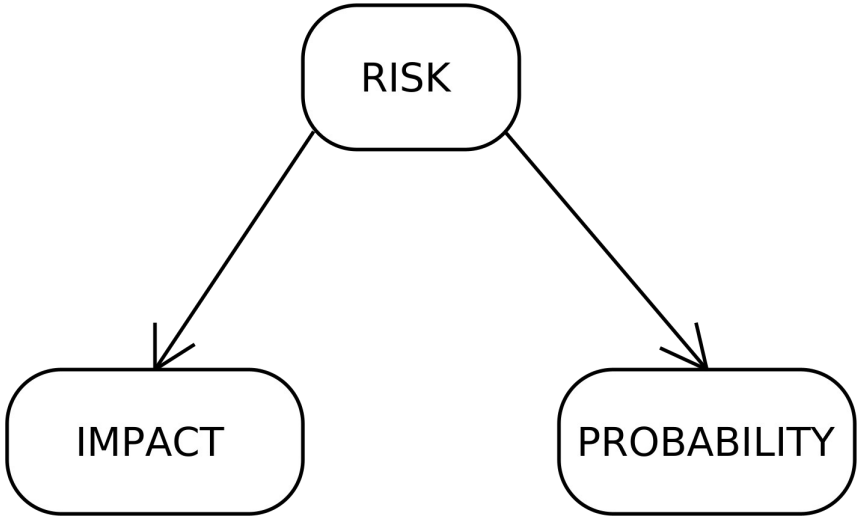
- ISO/IEC/IEEE 29119 is the recently adopted Testing Standard (there is a new version from 2022)
- Risk-based Testing is foreseen as part of the process



See http://www.softwaretestingstandard.org

What is a Risk

Risk = impact of an event * probability of the event



Risk Analysis

- Risk analysis deals with the **identification of the risks** (damage and probabilities) in the software testing process and in the prioritization of the test cases
- We usually start from a **Test Plan**:
  "A document describing the **scope**, **approach**, **resources**, and **schedule** of intended **test activities**. It identifies **test items**, the **features to be tested**, the **testing tasks**, who will do each task, and any risks requiring contingency planning" *(ISO/IEC/IEEE 29119)*

Steps for Risk Analysis (1/3)

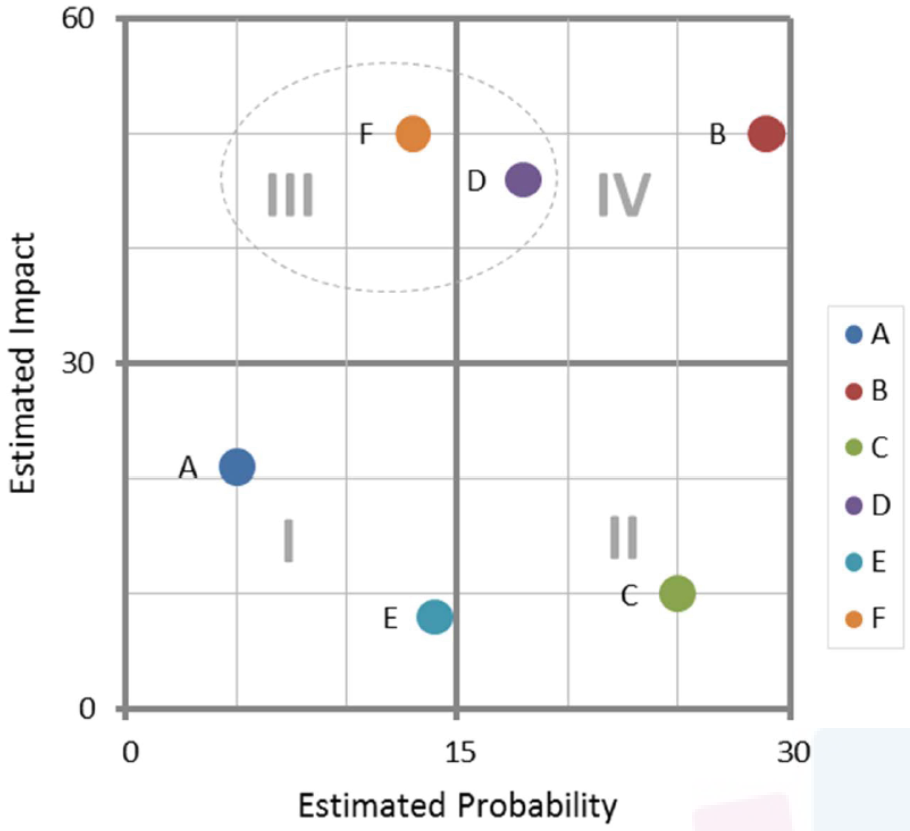
1. Define the risk items (e.g. type of failures for components)
2. Define probability of occurrence
3. Estimate impact
4. Compute Risk Values

Component	Estimated Probability	Estimated Impact	Computed Risk (= P * I)	Rank
A	5	21	105	6
B	29	50	1.450	1
C	25	10	250	4
D	18	46	828	2
E	14	8	112	5
F	13	50	650	3

M. Felderer, "Development of a Risk-Based Test Strategy and its Evaluation in Industry", PV226 Lasaris Seminar, 3rd Nov 2016.

Steps for Risk Analysis (2/3)

5. Determine Risk levels



M. Felderer, "Development of a Risk-Based Test Strategy and its Evaluation in Industry", PV226 Lasaris Seminar, 3rd Nov 2016.

Steps for Risk Analysis (3/3)

6. Definition and Refinement of Test Strategy

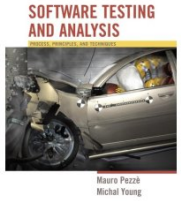
Testing techniques	Risk level I	II	III	IV	Components / Risk level	A I	B IV	C II	D III	E I	F III
Unit testing (100% branch coverage)				X			X				
Code reviews		X	X	X			X	X	X		X
Manual testing of use cases (base flow)		X						X			
Manual testing of use cases (base + alternative flows)			X	X			X		X		X
Exploratory testing	X			X		X	X			X	
Automated smoke/regression tests			X	X			X		X		X
Beta test phase at selected customers		X	X	X			X	X	X		X

M. Felderer, "Development of a Risk-Based Test Strategy and its Evaluation in Industry", PV226 Lasaris Seminar, 3rd Nov 2016.

Functional (Black Box) Testing

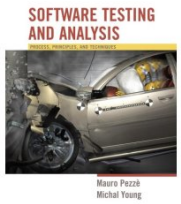
Functional Testing

- Functional testing: Deriving test cases from program specifications
  - *Functional* **refers to the source of information** used in test case design, not to what is tested
- *Also known as*:
  - specification-based testing (from specifications)
  - black-box testing (no view of the code)
- Functional specification = description of intended program behavior
  - either formal or informal



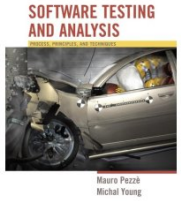
Functional testing: exploiting the specification

- Functional testing **uses the specification** (formal or informal) to partition the input space
  - E.g., specification of “roots” program suggests division between cases with zero, one, and two real roots
- Test each category, and boundaries between categories
  - No guarantees, but experience suggests failures often lie at the boundaries (as in the “roots” program)



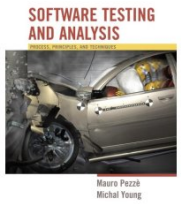
Early Functional Test Design

- Program code is **not necessary**
  - Only a description of intended behavior is needed
  - Even incomplete and informal specifications can be used
    - Although precise, complete specifications lead to better test suites
- Early functional test design has side benefits
  - Often **reveals ambiguities** and **inconsistency** in **specifications**
  - Useful for assessing testability
    - And improving test schedule and budget by improving spec
  - Useful explanation of specification
    - or in the extreme case (as in XP), test cases are the specifications

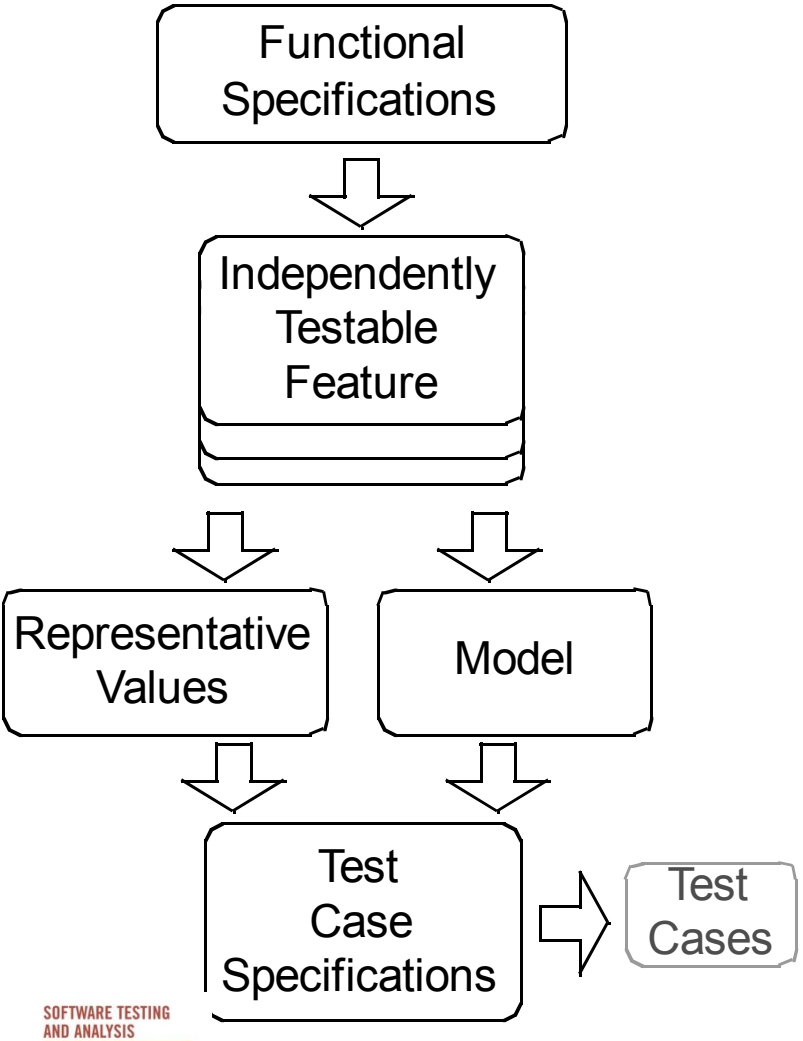


Functional vs structural test: granularity levels

- Functional test applies at all granularity levels:
  - Unit (from module interface spec)
  - Integration (from API or subsystem spec)
  - System (from system requirements spec)
  - Regression (from system requirements + bug history)
- Structural (code-based) test design applies to relatively small parts of a system:
  - Unit
  - Integration
- Functional testing is **best for *missing logic* faults**
  - A common problem: Some program logic was simply forgotten
  - Structural (code-based) testing will never focus on code that isn't there!

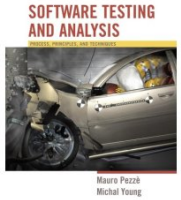


Steps: from specifications to test cases

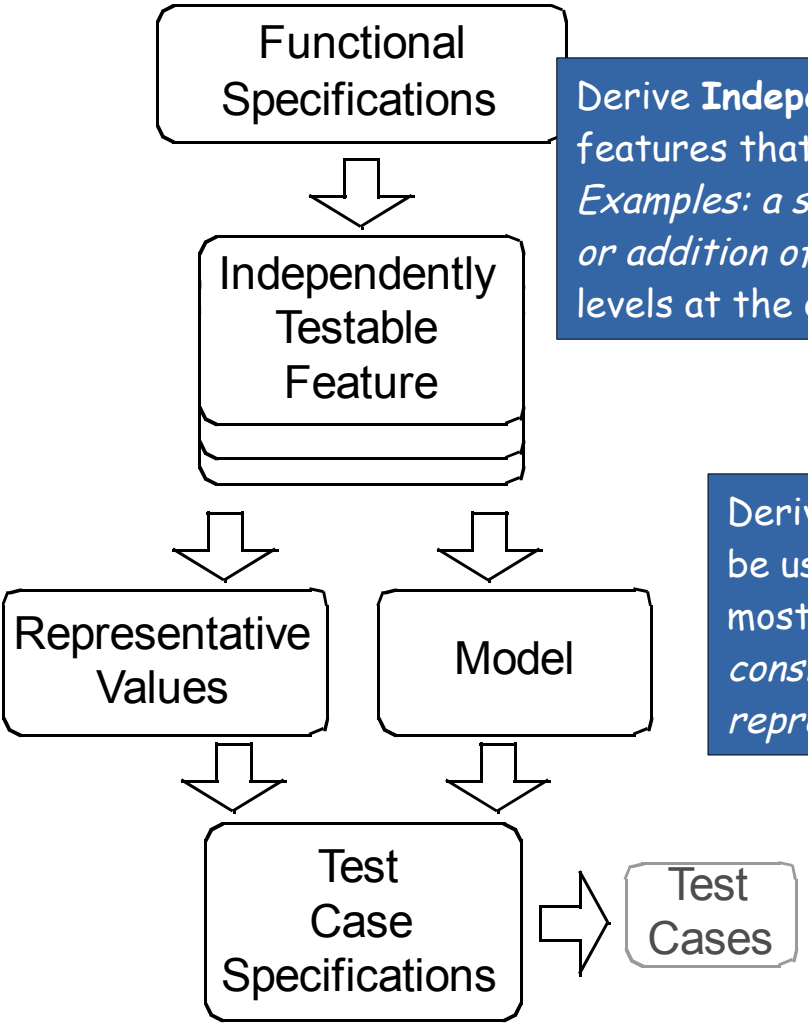


1. Decompose the specification
   - If the specification is large, break it into ***independently testable features*** to be considered in testing
2. Select representatives
   - Representative values of each input, or

     Representative behaviors of a *model*
     - Often simple input/output transformations don't describe a system. We use models in program specification, in program design, and in test design
3. Form test specifications
   - Typically: combinations of input values, or model behaviors
4. Produce and execute actual tests



Steps: from specifications to test cases



Derive **Independently Testable Features**: identify features that can be tested separately
Examples: a search functionality on a web application or addition of new users → this may map to different levels at the design and code level

NOTE: this helps also in determining if there are requirements that are not testable or need to be rewritten or clarified!

Derive **Representative values OR a model** that can be used to derive test cases. Note that this phase is mostly enumeration of values in isolation. *Example: considering empty list or a one element list as representative cases*

Generation of test case specification based on the previous step, usually based on the Cartesian product from the enumeration values (considering feasible cases). *Example: the search functionality, representative values might be 0,1, many characters and 0,1, many special characters, but the case {0,many} is clearly impossible*

Example one: using category partitioning

Using **combinatorial testing** (**category partition**) from the specifications. Sample Scenario:

"We are building a catalogue of computer components in which customers can select the different parts and assemble their PC for delivery. A model identifies a specific product and determines a set of constraints on available components. A set of (slot, component) pairs, corresponding to the required and optional slots of the model. A component might be empty for optional slots"

Step 1 - derive **Independently Testable Features**

Parameter *Model*

- Model number
- Number of required slots for selected model (#SMRS)
- Number of optional slots for selected model (#SMOS)

Parameter *Components*

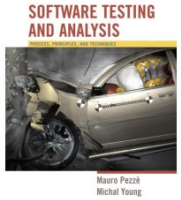
- Correspondence of selection with model slots
- Number of required components with selection ≠ empty
- Required component selection
- Number of optional components with selection ≠ empty
- Optional component selection

Environment element: *Product database*

- Number of models in database (#DBM)
- Number of components in database (#DBC)

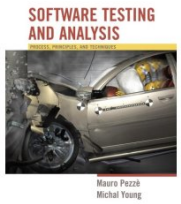
Step 2: Identify relevant values: components

- Correspondence of selection with model slots
  - Omitted slots
  - Extra slots
  - Mismatched slots
  - Complete correspondence
- Number of required components with non empty selection
  - 0
  - < number required slots
  - = number required slots
- Required component selection
  - Some defaults
  - All valid
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model
  - ≥ 1 not in database
- Number of optional components with non empty selection
  - 0
  - < #SMOS
  - = #SMOS
- Optional component selection
  - Some defaults
  - All valid
  - ≥ 1 incompatible with slots
  - ≥ 1 incompatible with another selection
  - ≥ 1 incompatible with model
  - ≥ 1 not in database



Step 3: Introduce constraints

- A combination of values for each category corresponds to a test case specification
  - in the example we have 314.928 test cases
  - most of the test cases represent “impossible” cases
    - Example: *zero slots* and *at least one incompatible slot*
- Introduce constraints to
  - rule out impossible combinations
  - reduce the size of the test suite if too large



Step 3: error constraint

[Error] indicates a value class that
- corresponds to erroneous values
- need be tried only once

Model number
Malformed [error]
Not in database [error]
Valid

Correspondence of selection with model slots
Omitted slots [error]
Extra slots [error]
Mismatched slots [error]
Complete correspondence

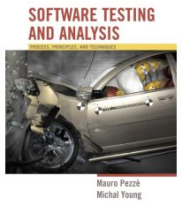
Number of required comp. with non empty selection
0 [error]
< number of required slots [error]

Required comp. selection
≥ 1 not in database [error]

Number of models in database (#DBM)
0 [error]

Number of components in database (#DBC)
0 [error]

Error constraints reduce test suite from 314.928 to 2.711 test cases



Step 3: property constraints

Number of required slots for selected model (#SMRS)

1	[property RSNE]
Many	[property RSNE] [property RSMANY]

Number of optional slots for selected model (#SMOS)

1	[property OSNE]
Many	[property OSNE] [property OSMANY]

Number of required comp. with non empty selection

0	[if RSNE] [error]
< number required slots	[if RSNE] [error]
= number required slots	[if RSMANY]

Number of optional comp. with non empty selection

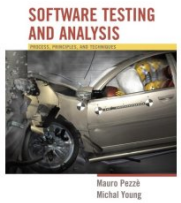
< number required slots	[if OSNE]
= number required slots	[if OSMANY]

constraint [property] [if-property]
rule out invalid combinations of values

[property] groups values of a single parameter to identify subsets of values with common properties

[if-property] bounds the choices of values for a category that can be combined with a particular value selected for a different category

from 2.711 to 908 test cases



Step 3: single constraints

Number of required slots for selected model (#SMRS)

0	[single]
1	[property RSNE] [single]

Number of optional slots for selected model (#SMOS)

0	[single]
1	[single] [property OSNE]

Required component selection

Some default	[single]

Optional component selection

Some default	[single]

Number of models in database (#DBM)

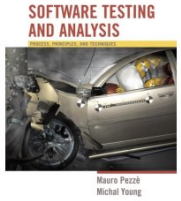
1	[single]

Number of components in database (#DBC)

1	[single]

[single] indicates a value class that test designers choose to test only once to reduce the number of test cases

from 908 to 69 test cases



Example - Summary

Parameter Model

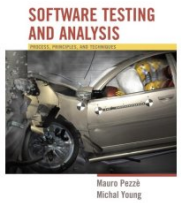
- Model number
  - Malformed [error]
  - Not in database [error]
  - Valid
- Number of required slots for selected model (#SMRS)
  - 0 [single]
  - 1 [property RSNE] [single]
  - Many [property RSNE] [property RSMANY]
- Number of optional slots for selected model (#SMOS)
  - 0 [single]
  - 1 [property OSNE] [single]
  - Many [property OSNE] [property OSMANY]

Environment Product data base

- Number of models in database (#DBM)
  - 0 [error]
  - 1 [single]
  - Many
- Number of components in database (#DBC)
  - 0 [error]
  - 1 [single]
  - Many

Parameter Component

- Correspondence of selection with model slots
  - Omitted slots [error]
  - Extra slots [error]
  - Mismatched slots [error]
  - Complete correspondence
- # of required components (selection ≠ empty)
  - 0 [if RSNE] [error]
  - < number required slots [if RSNE] [error]
  - = number required slots [if RSMANY]
- Required component selection
  - Some defaults [single]
  - All valid
    - ≥ 1 incompatible with slots
    - ≥ 1 incompatible with another selection
    - ≥ 1 incompatible with model
    - ≥ 1 not in database [error]
- # of optional components (selection ≠ empty)
  - 0
  - < #SMOS [if OSNE]
  - = #SMOS [if OSMANY]
- Optional component selection
  - Some defaults [single]
  - All valid
    - ≥ 1 incompatible with slots
    - ≥ 1 incompatible with another selection
    - ≥ 1 incompatible with model
    - ≥ 1 not in database [error]



Example Two – Deriving a Model

From an informal specification:

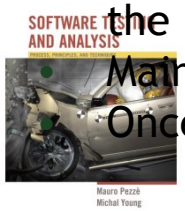
Maintenance: The Maintenance function records the history of items undergoing maintenance.

- If the product is covered by warranty or maintenance contract, maintenance can be requested either by calling the maintenance toll free number, or through the web site, or by bringing the item to a designated maintenance station.
- If the maintenance is requested by phone or web site and the customer is a US or EU resident, the item is picked up at the customer site, otherwise, the customer shall ship the item with an express courier.
- If the maintenance contract number provided by the customer is not valid, the item follows the procedure for items not covered by warranty.
- If the product is not covered by warranty or maintenance contract, maintenance can be requested only by bringing the item to a maintenance station. The maintenance station informs the customer of the estimated costs for repair. Maintenance starts only when the customer accepts the estimate.
- If the customer does not accept the estimate, the product is returned to the customer.
- Small problems can be repaired directly at the maintenance station. If the maintenance station cannot solve the problem, the product is sent to the maintenance regional headquarters (if in US or EU) or to the maintenance main headquarters (otherwise).
- If the maintenance regional headquarters cannot solve the problem, the product is sent to the maintenance main headquarters.
- Maintenance is suspended if some components are not available.
- Once repaired, the product is returned to the customer.

Multiple choices in the first step ...

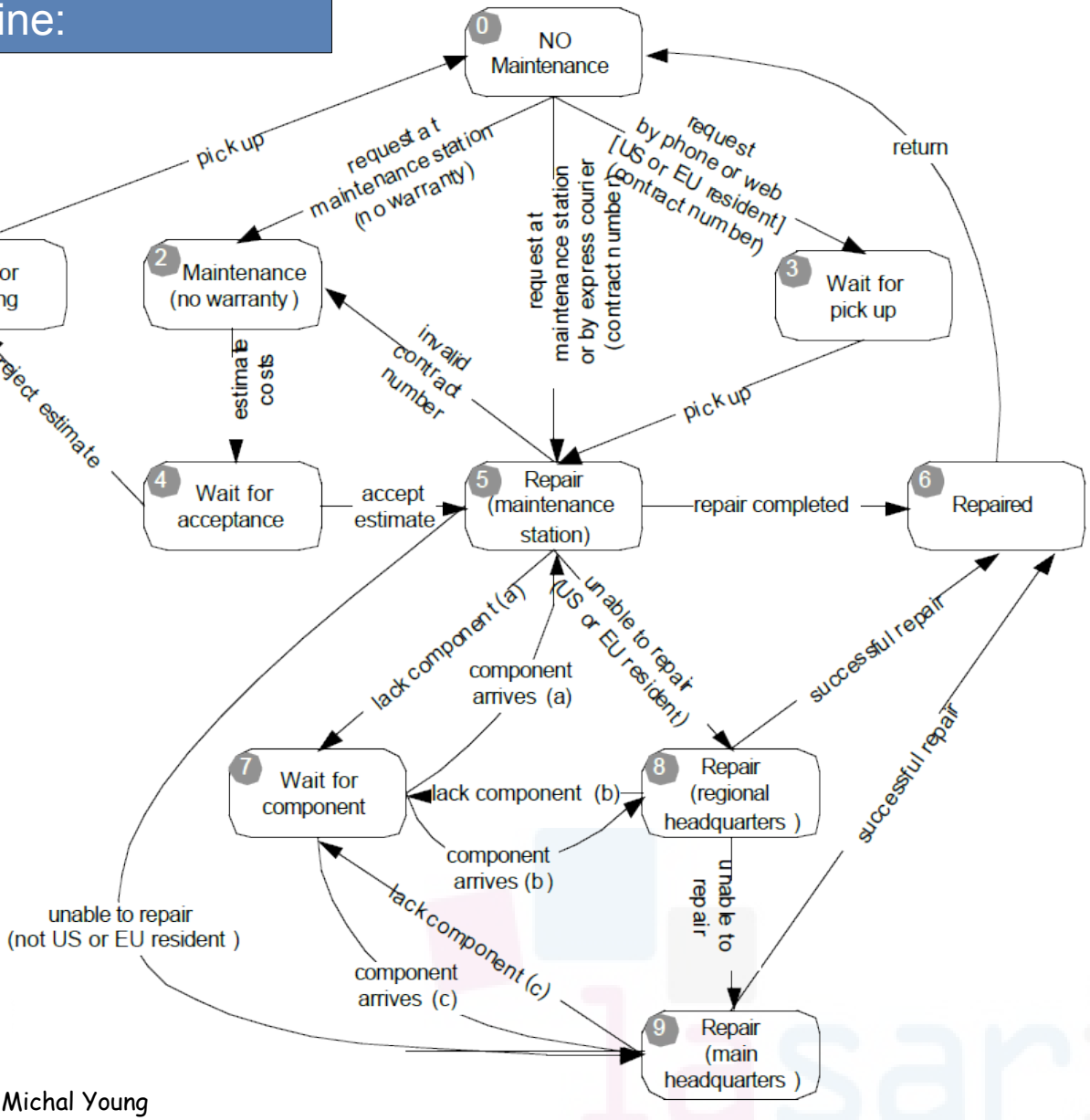
... determine the possibilities for the next step ...

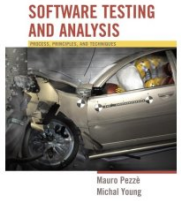
... and so on ...



Example Two – Deriving a Model

To a finite state machine:





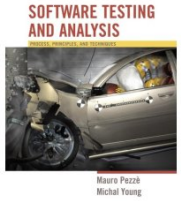
Example Two – Deriving a Model

To a test suite:

TC1	0	2	4	1	0					
TC2	0	5	2	4	5	6	0			
TC3	0	3	5	9	6	0				
TC4	0	3	5	7	5	8	7	8	9	6	0

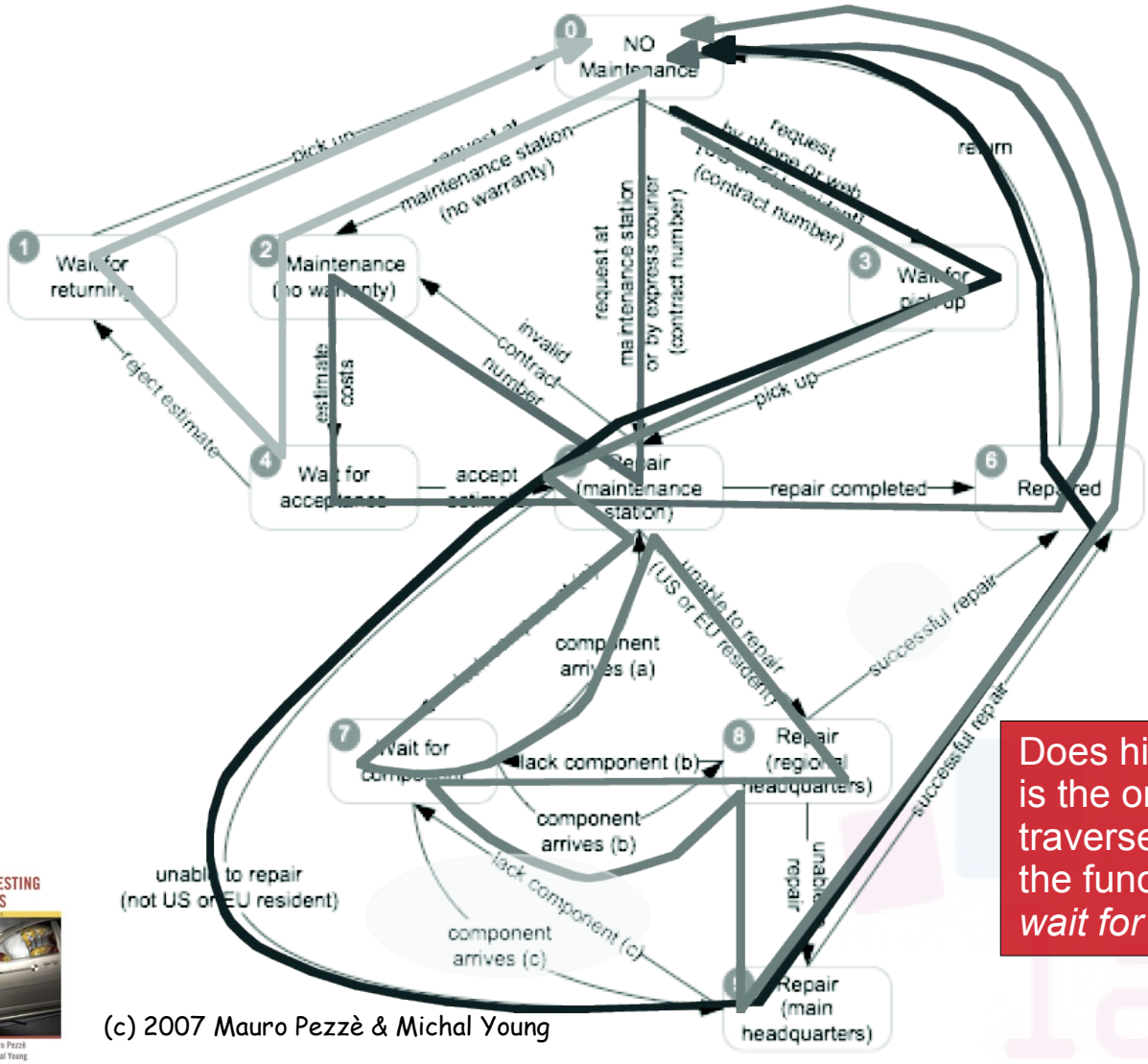
Meaning: From state 0 to state 2 to state 4 to state 1 to state 0

Is this a thorough test suite?
How can we judge?



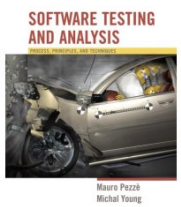
Example Two – Deriving a Model

Using transition coverage:



Using **transition coverage**: Every transition between states should be traversed by at least one test case

Does history matter? That is the order in which we traverse a node influences the functionality? (e.g. see *wait for completion*)



References

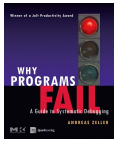
Most of the source code examples, class diagrams, etc... from [2] if not differently stated

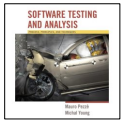
[1] A. Zeller, *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*, 2 edition. Amsterdam ; Boston: Morgan Kaufmann, 2009.

[2] M. Pezzè and M. Young, *Software Testing And Analysis: Process, Principles And Techniques*. Hoboken, N.J.: John Wiley & Sons Inc, 2007.

[3] Michel Felderer, “Development of a Risk-Based Test Strategy and its Evaluation in Industry”, PV226 Lasaris Seminar, 3rd Nov 2016. ADD LINK!

[4] ISO/IEC/IEEE 29119 Software Testing Standard,
http://www.softwaretestingstandard.org
https://www.iso.org/standard/45142.html





Acceptance Testing example using Fitnesse (www.fitnesse.org)

Mutation Testing example using PiTest (www.pitest.org)