PV260 - SOFTWARE QUALITY

LECT 7. Requirements and Test Cases. From Unit Testing to Integration Testing

Bruno Rossi
brossi@mail.muni.cz

LAB OF SOFTWARE ARCHITECTURES
AND INFORMATION SYSTEMS

FACULTY OF INFORMATICS
MASARYK UNIVERSITY, BRNO

Outline

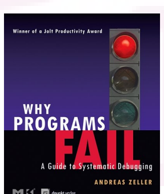
- Software Testing
  - → Introduction
  - → Basic Principles
- From Requirements to Test Cases
  - → Functional testing
  - → Translating specifications into test cases
- From Unit Testing to Integration Testing
  - → comparison with unit testing
  - → Strategies
- Specific Issues in Testing Object Oriented Software

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

George Box

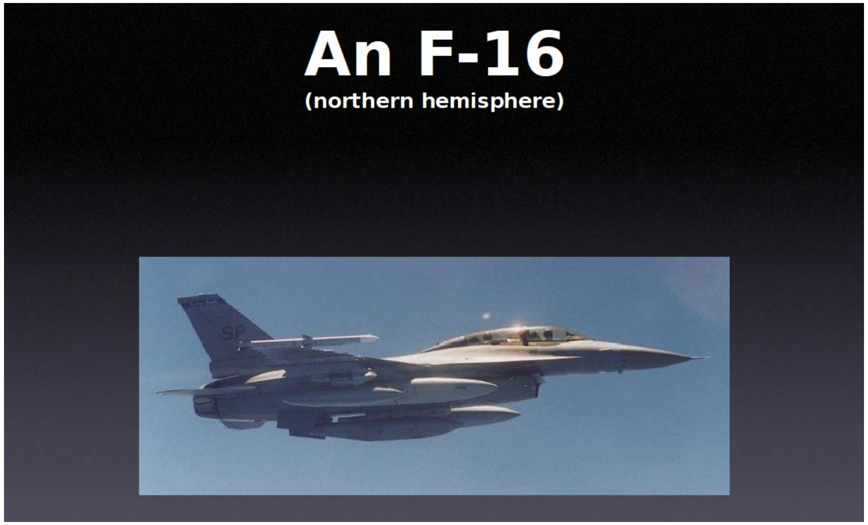
Introduction

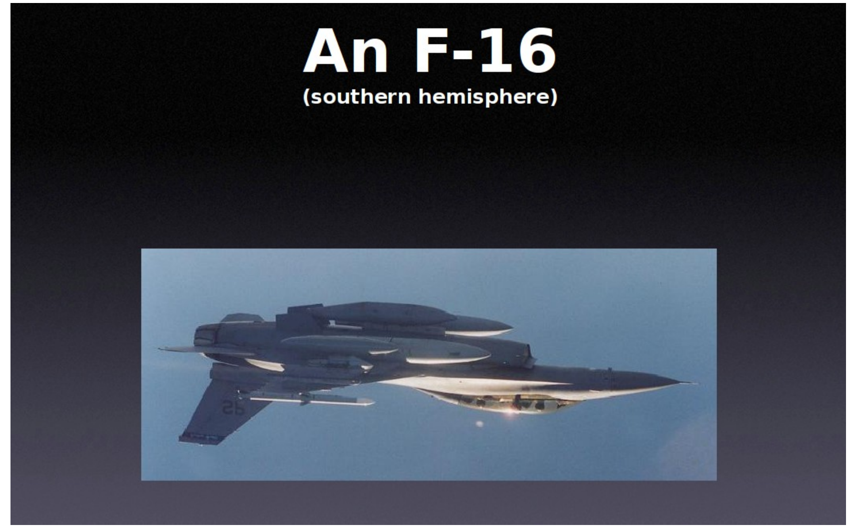
- In Eclipse and Mozilla, 30–40% of all changes are fixes (Sliverski et al., 2005)
- Fixes are 2–3 times smaller than other changes (Mockus +Votta, 2000)
- 4% of all one-line changes introduce new errors (Purushothaman + Perry, 2004)

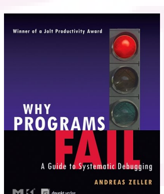


A. Zeller, Why Programs Fail, Second Edition: A Guide to Systematic Debugging, 2 edition. Amsterdam ; Boston: Morgan Kaufmann, 2009.

Motivating Examples







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

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

lasaris

What is Software Testing

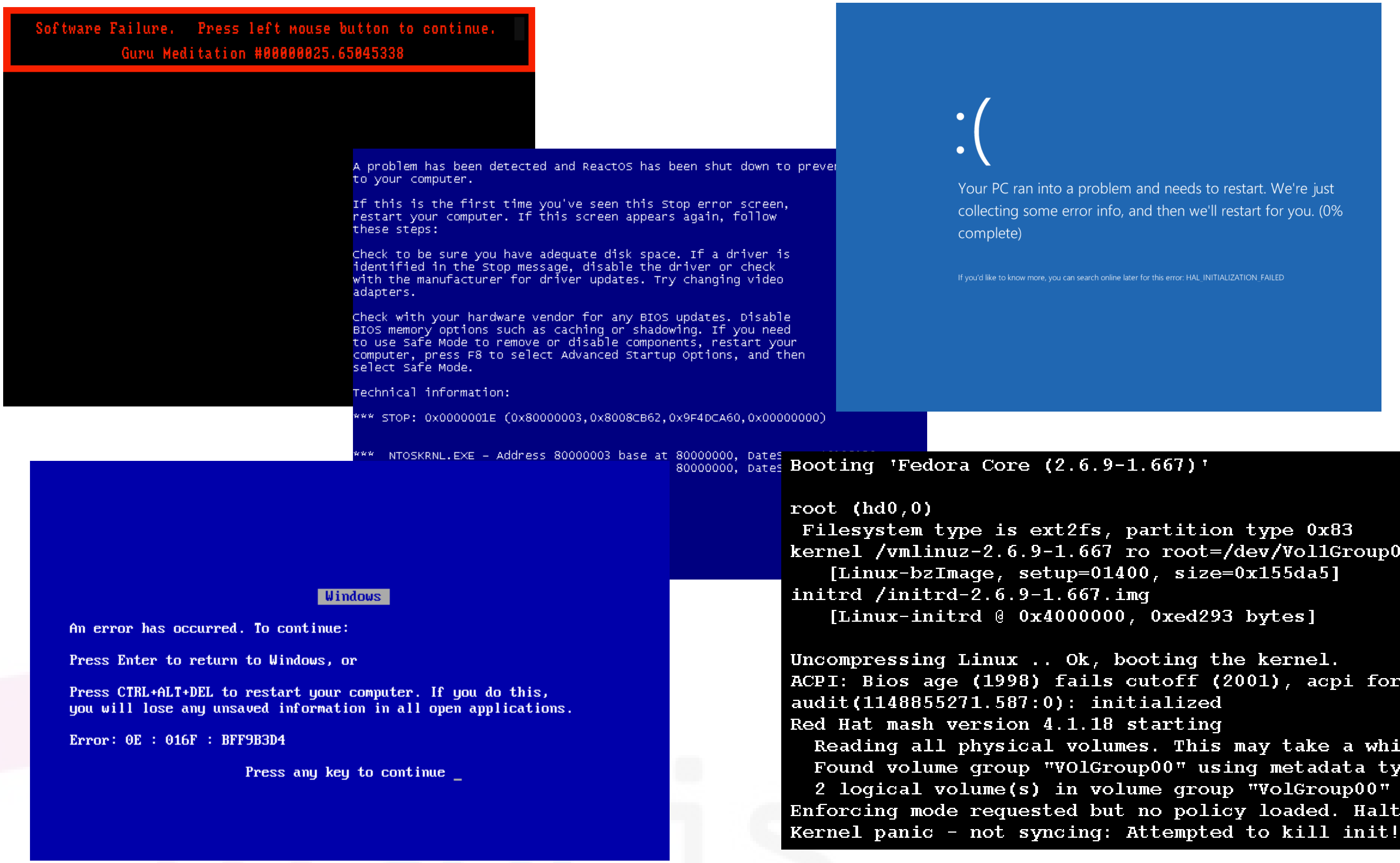
Reminder for some important terms:

- **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 haven't seen some of these



Maybe 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.

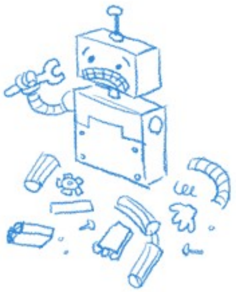
If you see them, show them this information:

AB38WEPIDWfs5FLs3YWvAJbHZzGGd1X3seRUSOX7Kh9K1gde_FLVY4GDBjkn
8jPuyamICiGBZExjMpiZT4j7rx-0NZ707H-cPNSEbJ0n_b7MYf692YtZtrQI
DsAGxZ38bYUMy4UyGJHtGSUG4N0BuXXX35-jWJZDtkJoj_ZNdJoOTOJSG2PC
X_mCxpP5lQi7-rZUcx83I33yavfWr2WcE4EUyS0TyqzFqzh_QJVNbc7_yxRH
8udCCKkxQVBdsBDK2qejBUTemZ31SFOWC10wUulgiE-L750WxOmGjsP2GiSp
6Z3-0IepREkPtU649pzpZ6PBIqWlBXOZ8GnoQIiAiqqOcneErAHFs0aCNi9-
tB34vRO8oFi_JtZ4AzvPEVTpqLiaAs_PwERN2NRADOPVqrtEPbUGZh-c7PdZ

Google

404. That's an error.

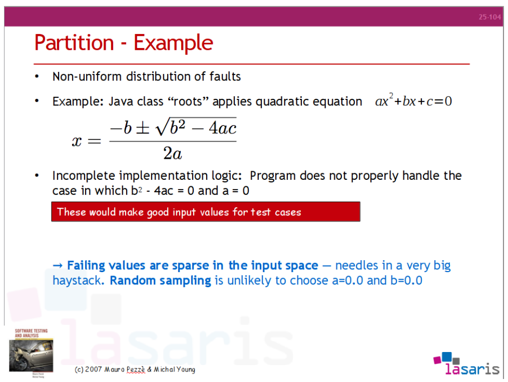
The requested URL /intl/en/options/ was not found on this server. That's all we know.



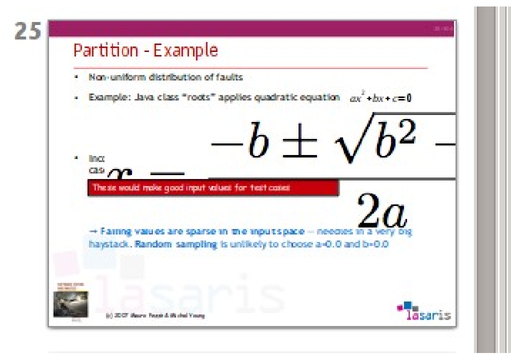
And defects are everywhere…

This is one failure I encountered when preparing this presentation on *LibreOffice 4.2.7.2*

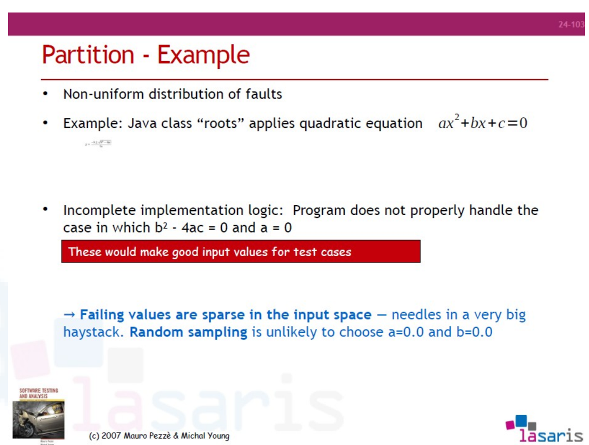
A formula in ppt that got converted into image – looks good when editing



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



When converted to pdf...



What about the term “Bug”?

Where is 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

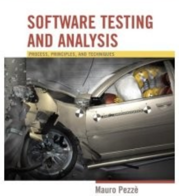
Definitions according to IEEE Std 1044-2009 “IEEE Standard Classification for Software Anomalies“

Who's to blame?

image from http://blog.smartbear.com/sqc/when-bad-software-requirements-happen-to-good-people

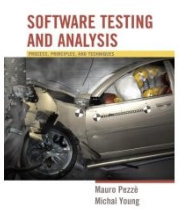
Basic Principles of Testing

- **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

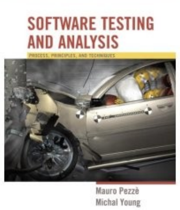


Sensitivity Example

- 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?

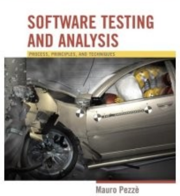


Sensitivity Example

- 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:

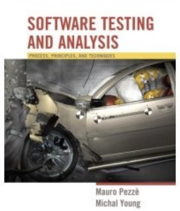
Java SE Technical Documentation

- "[...] 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());
    ...
}
```

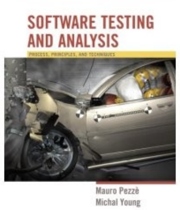
- From a language (e.g. Java) point of view, why are we obliged to declare the exception we throw from a method - isn't this redundant?

```
    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;
    }
```

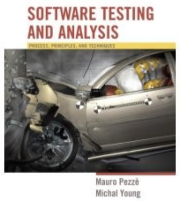
But restrictions can be applied at different levels, e.g. at the architectural level the decision of making the HTTP protocol stateless hugely simplified testing (and as such made the protocol more robust)

Partition: divide and 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

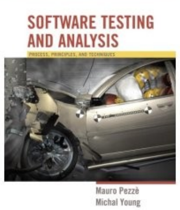
- Non-uniform distribution of faults
- Example: Java class "roots" applies quadratic equation $ax^2+bx+c=0$

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

- 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



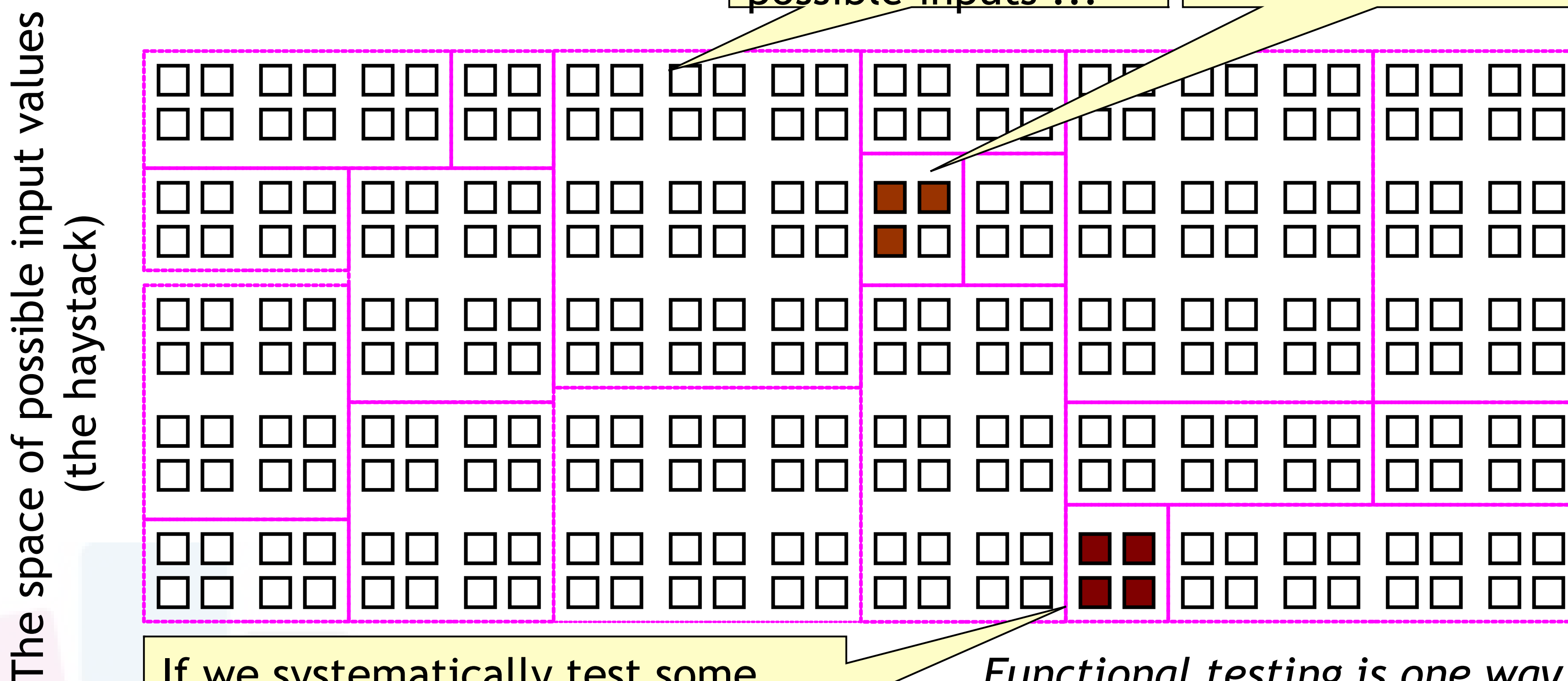
Partition - Example

■ Failure (valuable test case)

□ No failure

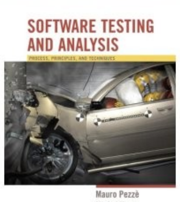
Failures are sparse in the space of possible inputs ...

... but dense in some parts of the space



If we systematically test some cases from each part, we will include the dense parts

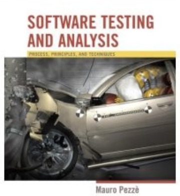
Functional testing is one way of drawing pink lines to isolate regions with likely failures



lasaris

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
```

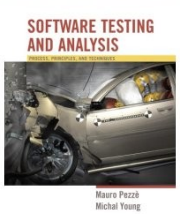
Why wasn't a more efficient binary format selected?

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



From Requirements to Test Cases

Characteristics of Requirements

According to *ISO/IEC/IEEE 29148-2011* standard:

- **Correctness**: requirements represent the client's view
- **Completeness**: all possible scenarios through the system are described, including exceptional behavior by the user
- **Consistency:** There are functional or nonfunctional requirements that contradict each other
- **Clarity:** There are no ambiguities in the requirements
- **Realism:** Requirements can be implemented and delivered
- **Traceability:** Each system function can be traced to a corresponding set of functional requirements

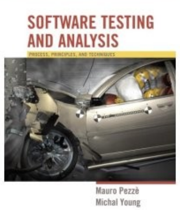
Test Cases Definition

According to *IEEE Std 829-1998:*

- **Test Case Specification:** “A document specifying **inputs**, **predicted results**, and a **set of execution conditions** for a **test item**”

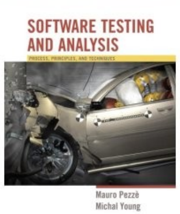
Functional Testing

- 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



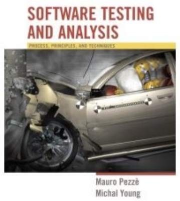
Functional testing: exploiting the specification

- 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)



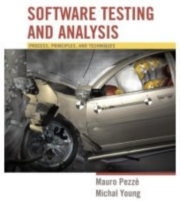
Why functional Tests?

- The base-line technique for designing test cases
  - **Timely**
    - Often useful in refining specifications and assessing testability *before* code is written
  - **Effective**
    - finds some classes of fault (e.g., missing logic) that can elude other approaches
  - **Widely applicable**
    - to any description of program behavior serving as spec
    - at any level of granularity from module to system testing.
  - **Economical**
    - typically less expensive to design and execute than structural (code-based) test cases



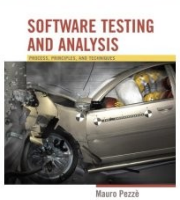
Early Functional Test Design

- 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 spec
  - 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 spec

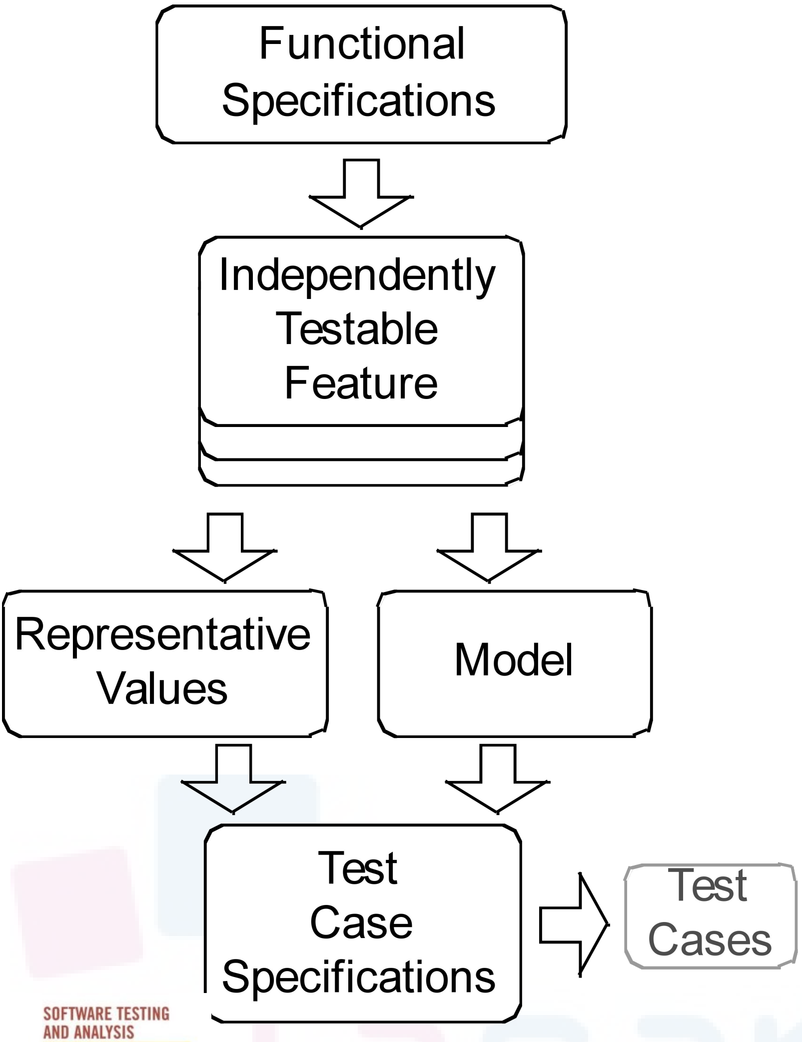


Functional vs structural test: granularity levels

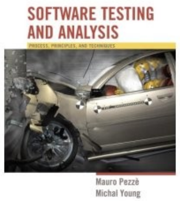
- Functional test applies at all granularity levels:
  - Unit (from module interface spec)
  - Integration (from API or subsystem spec)
  - System (from system requirements spec)
  - Regression (from system requirements + bug history)
- Structural (code-based) test design applies to relatively small 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!



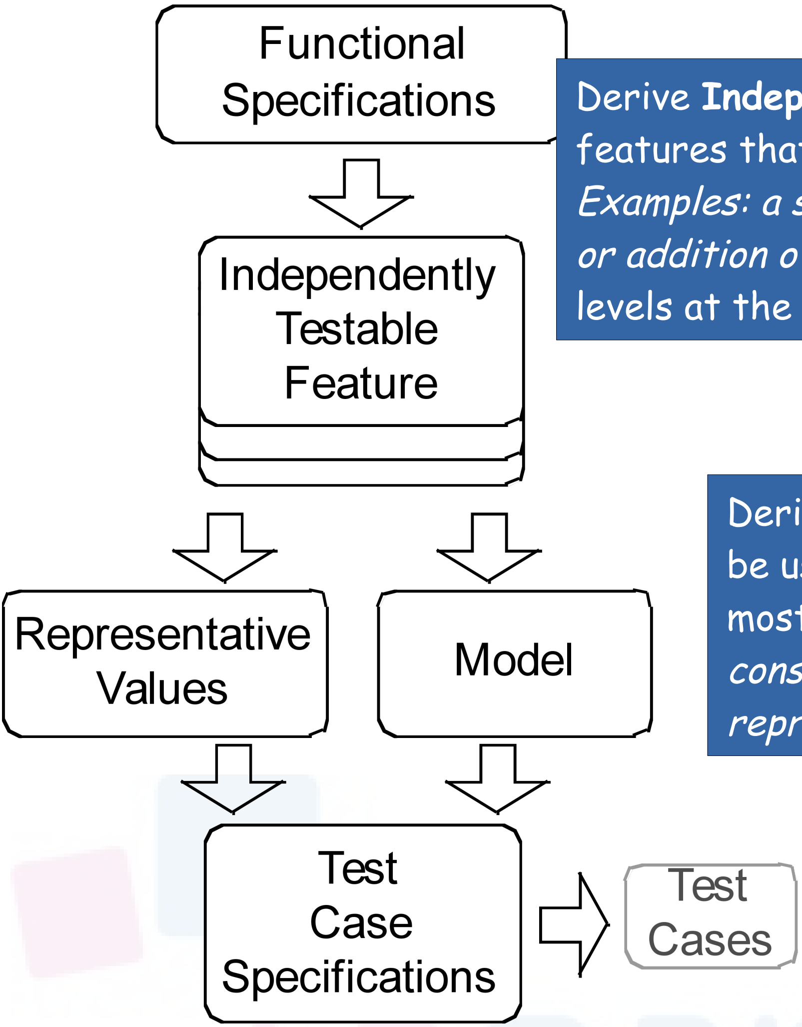
Steps: from specifications to test cases



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: example



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

- *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: Identify independently testable units

Parameter *Model*

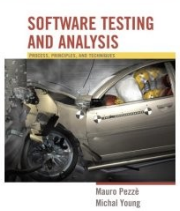
- Model number
- Number of required slots for selected model (#SMRS)
- Number of optional slots for selected model (#SMOS)

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)

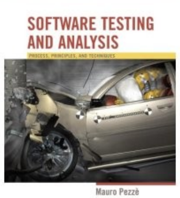


Step 2: Identify relevant values: Component (1/3)

Model number
- Malformed
- Not in database
- Valid

Number of required slots for selected model (#SMRS)
- 0
- 1
- Many

Number of optional slots for selected model (#SMOS)
- 0
- 1
- Many



Step 2: Identify relevant values: Component (2/3)

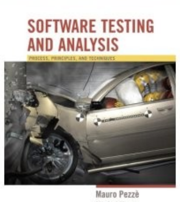
Correspondence of selection with model slots
- Omitted slots
- Extra slots
- Mismatched slots
- Complete correspondence

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 2: Identify relevant values: Component (3/3)

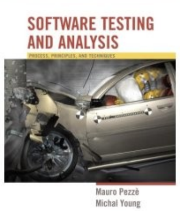
Number of models in database (#DBM)

- 0
- 1
- Many

Number of components in database (#DBC)

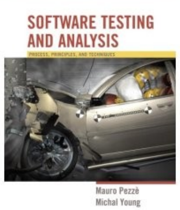
- 0
- 1
- Many

Note 0 and 1 are unusual (special) values. They might cause unanticipated behavior alone or in combination with particular values of other parameters.



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 which are impossible!
    - 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

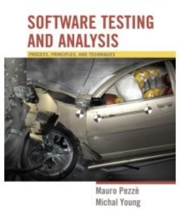
- corresponds to a erroneous values
- need be tried only once

Example

Model number: Malformed and Not in database

error value classes

- No need to test all possible combinations of errors
- One test is enough (we assume that handling an error case bypasses other program logic)



Example - Step 3: error constraint

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]

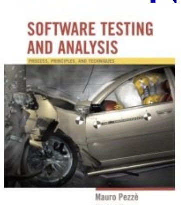
Number of models in database (#DBM)

- 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

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

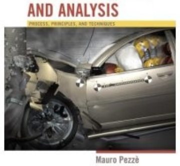
Example

combine

Number of required comp. with non empty selection = number required slots [if RSMANY]

only with

Number of required slots for selected model (#SMRS) = Many [Many]



Example - 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]

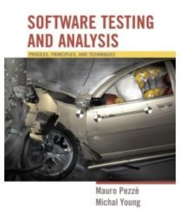
Number of required comp. with non empty selection

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]

from 2.711 to 908 test cases



Step 3 (cont): single constraints

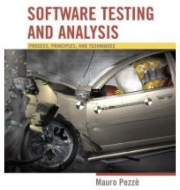
[single] indicates a value class that test designers choose to test only once to reduce the number of test cases

Example

value some default for required component selection and optional component selection may be tested only once despite not being an erroneous condition

note -

single and error have the same effect but differ in rationale. Keeping them distinct is important for documentation and regression testing



Example - 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]

from 908 to 69 test cases

Example - Summary

Parameter Model

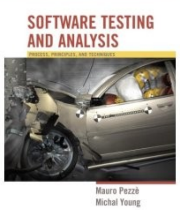
- Model number
  - Malformed [error]
  - Not in database [error]
  - Valid
- Number of required slots for selected model (#SMRS)
  - 0 [single]
  - 1 [property RSNE] [single]
  - Many [property RSNE] [property RSMANY]
- Number of optional slots for selected model (#SMOS)
  - 0 [single]
  - 1 [property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

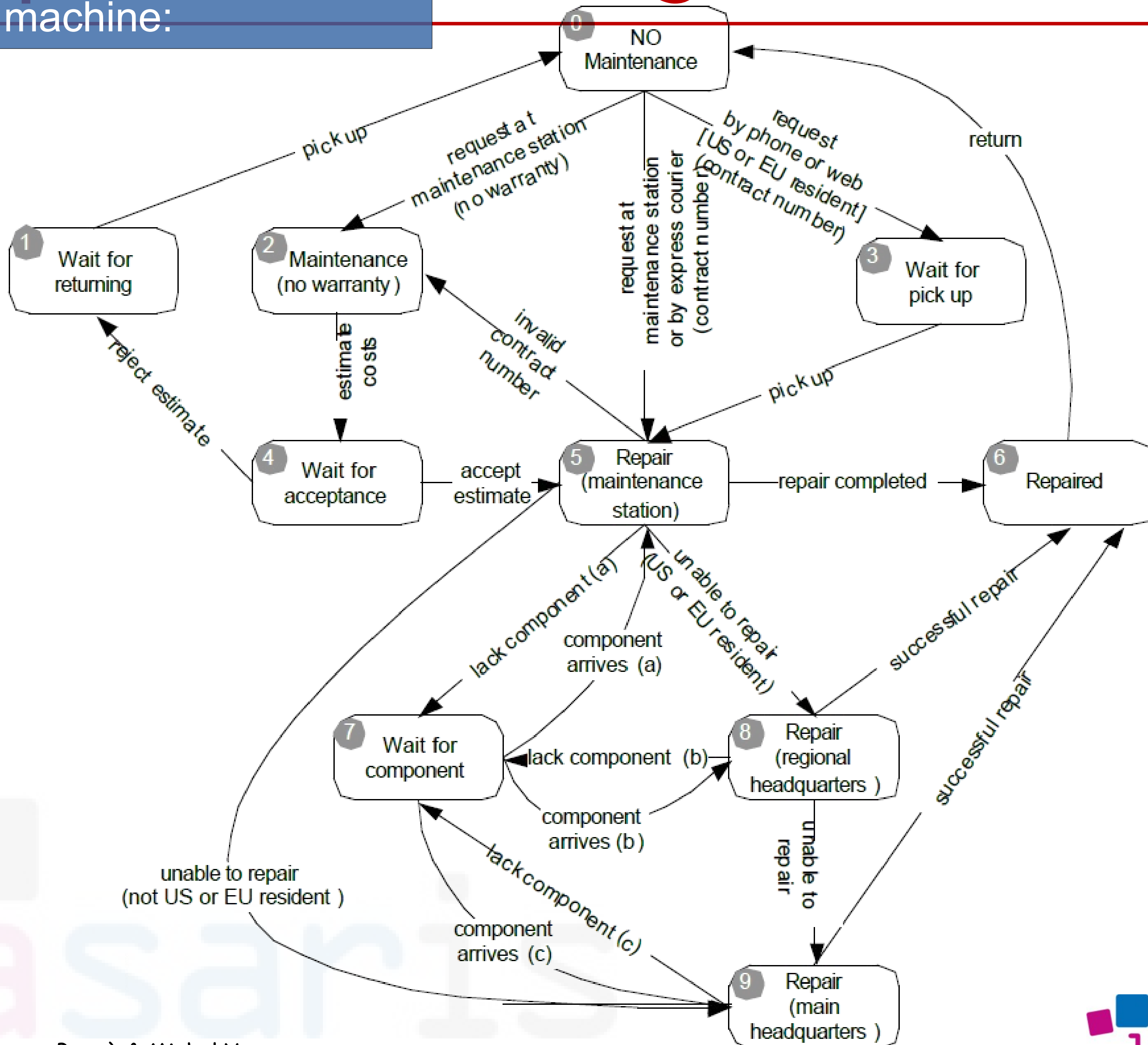
> Multiple choices in the first step ...

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

> ... and so on ...

Example Two: Deriving a model

To a finite state machine:



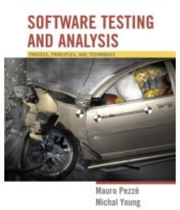
Example Two: Deriving a model

To a test suite:

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

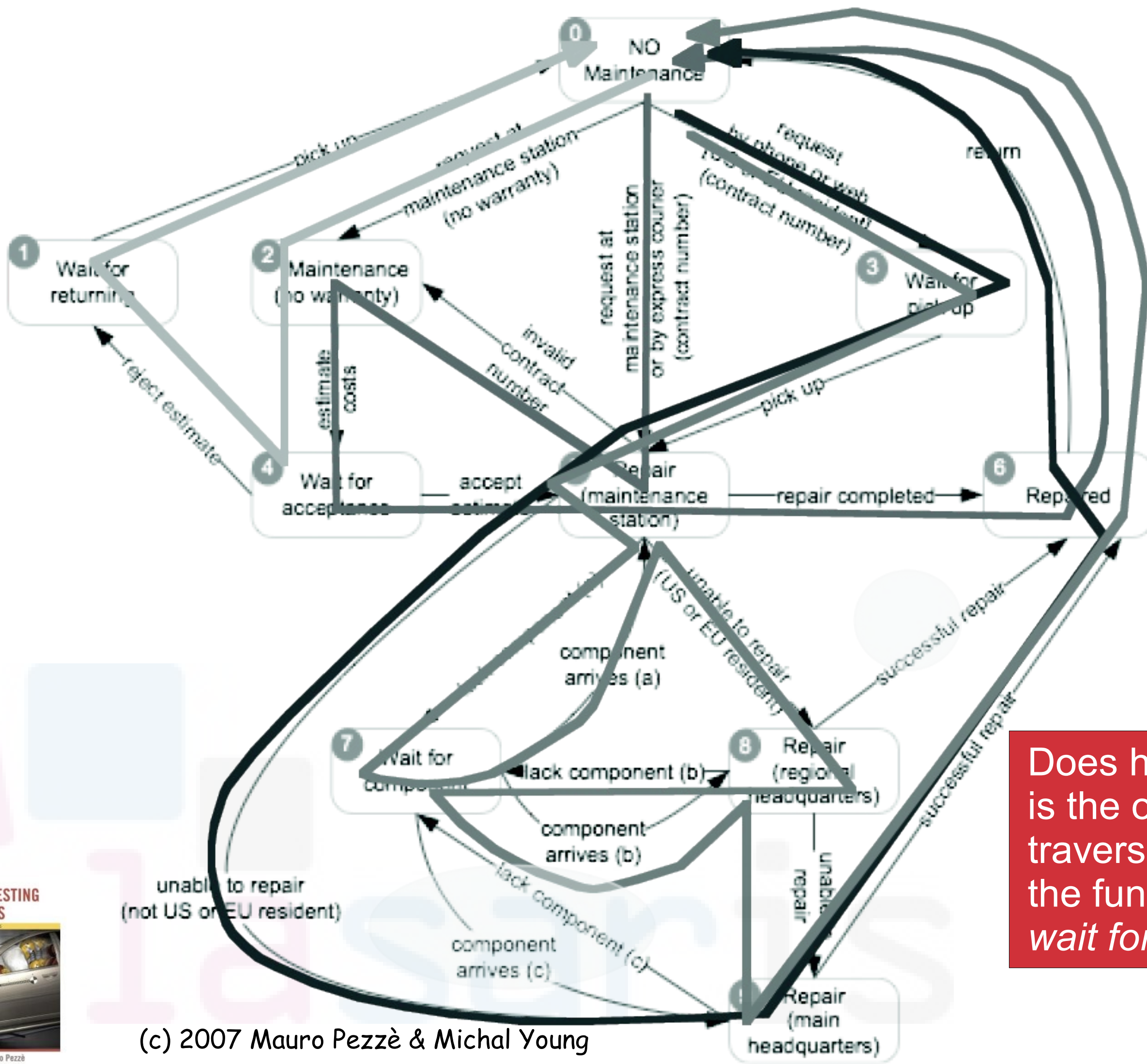
TC1	0	2	4	1	0						
TC2	0	5	2	4	5	6	0				
TC3	0	3	5	9	6	0					
TC4	0	3	5	7	5	8	7	8	9	6	0

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



Example Two: Deriving a model

Using transition coverage:

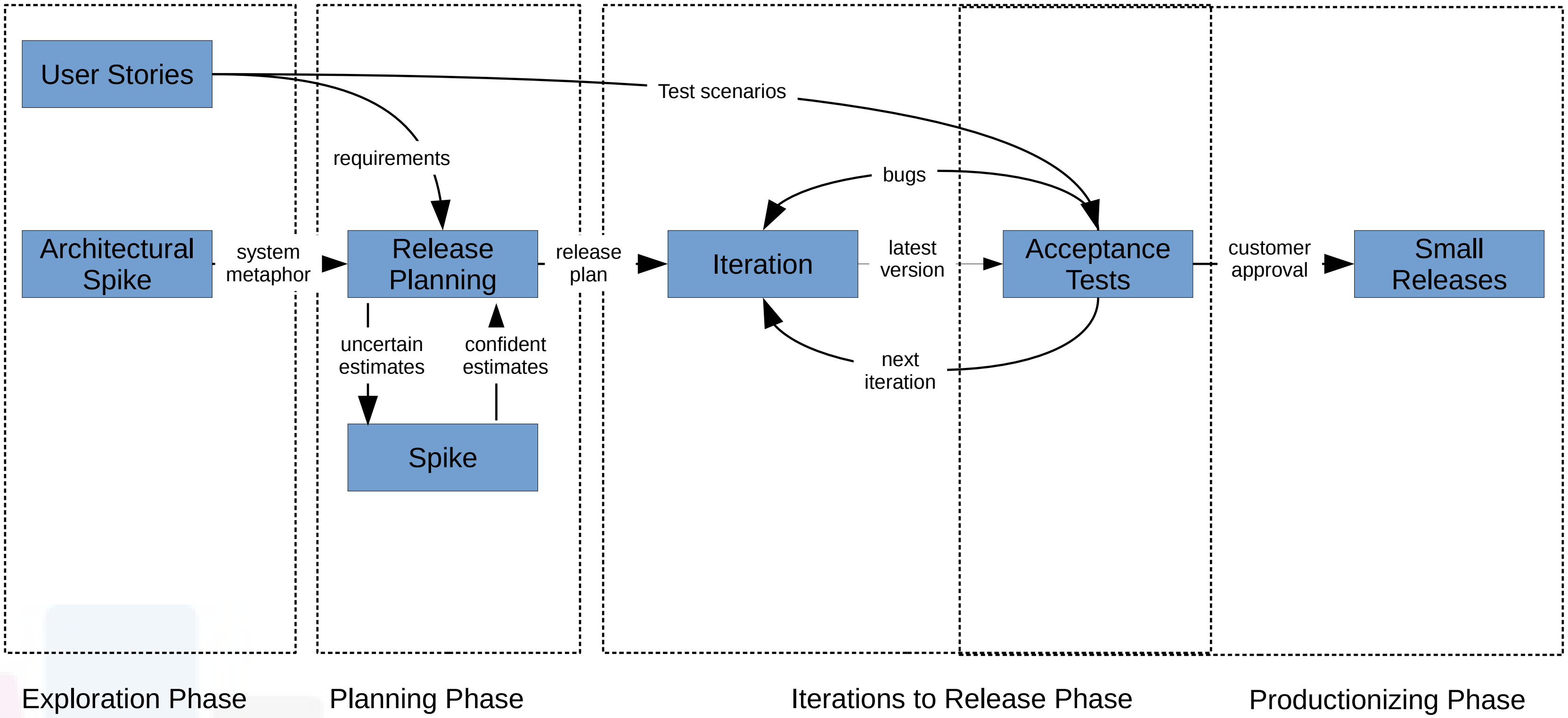


Using **transition coverage**: Every transition between states should be traversed by at least one test case

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

A complementary point of view (1/5)

eXtreme Programming (XP) process



In the Agile context, the problem of functional testing has been addressed by having **user stories** and **acceptance tests in collaboration with customers, constantly updated and runnable**

A complementary point of view (2/5)

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

lasaris

A complementary point of view (3/5)

```
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

A complementary point of view (4/5)

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

A complementary point of view (5/5)

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 (exception below)

```
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(DelegatingMethodAcces
	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*

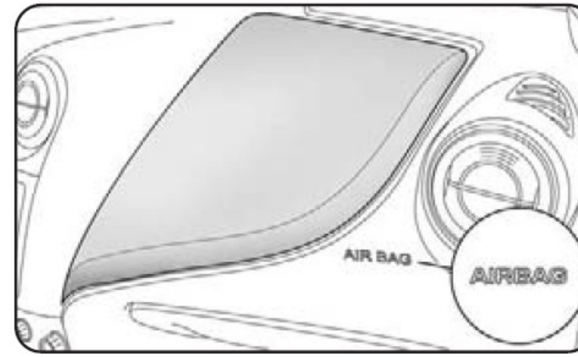
FitNesse

lasaris

From Unit Testing to Integration Testing

Motivating Example

- On cars, children should not sit on the front passenger if passenger's airbag has not been disabled



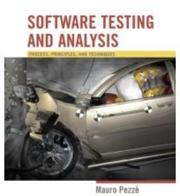
- On most cars there is lever to turn to disable it

- However: one cars' manufacturer had trouble with the following scenario
  - Airbag turned off by the user
  - Car sent for check-up → central unit replaced
  - Complete reset of the system **reactivated airbags even though lever was OFF**

- How could have this been detected by testing & which type of tests?

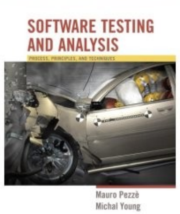
What is integration testing?

	Module test	Integration test	System test
Specification:	Module interface	Interface specs, module breakdown	Requirements specification
Visible structure:	Coding details	Modular structure (software architecture)	— none —
Scaffolding required:	Some	Often extensive	Some
Looking for faults in:	Modules	Interactions, compatibility	System functionality



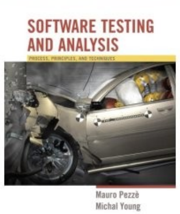
Integration versus Unit Testing

- Unit (module) testing is a necessary foundation
  - Unit level has maximum controllability and visibility
  - Integration testing can never compensate for inadequate unit testing
- Integration testing may serve as a *process check*
  - If module faults are revealed in integration testing, they **signal inadequate unit testing**
  - If integration faults **occur in interfaces** between correctly implemented modules, the errors can be traced to module breakdown and interface specifications



Integration Faults

- Inconsistent interpretation of parameters or values
  - Example: Mixed units (meters/yards) in Martian Lander
- Violations of value domains, capacity, or size limits
  - Example: Buffer overflow
- Side effects on parameters or resources
  - Example: Conflict on (unspecified) temporary file
- Omitted or misunderstood functionality
  - Example: Inconsistent interpretation of web hits
- Nonfunctional properties
  - Example: Unanticipated performance issues
- Dynamic mismatches
  - Example: Incompatible polymorphic method calls



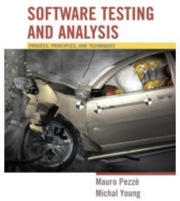
Example: A Memory Leak

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)

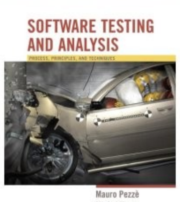


Example: A Memory Leak

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.



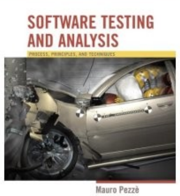
Example: A Memory Leak

Apache web server, version 2.0.48

Response to normal page request on secure (https) port

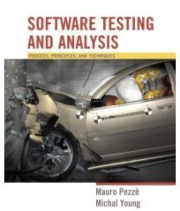
```
Static void ssl_io_filter_disable(ap_filter_t *f)
{   bio_filter_in_ctx_t *inctx = f->ctx;
    SSL_free(inctx -> ssl);
    inctx->ssl = NULL;
    inctx->filter ctx->pssl = NULL;
}
```

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



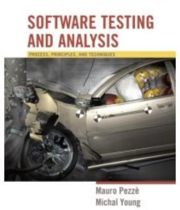
Maybe you have heard...

- Yes, I implemented ⟨module A⟩, but I didn't test it thoroughly yet. It will be tested along with ⟨module B⟩ when that's ready.



Translation...

- Yes, I implemented ⟨module A⟩, but I didn't test it thoroughly yet. It will be tested along with ⟨module B⟩ when that's ready.
- I didn't think at all about the strategy for testing. I didn't design ⟨module A⟩ for testability and I didn't think about the best order to build and test modules ⟨A⟩ and ⟨B⟩.

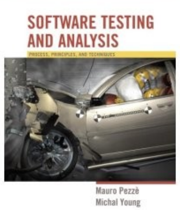


Big Bang Integration Test

An extreme and desperate approach:

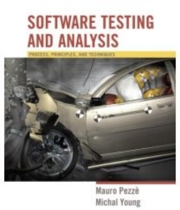
Test only after integrating all modules

+ Does not require scaffolding
    - The only excuse, and a bad one
- Minimum observability, diagnosability, efficacy, feedback
- High cost of repair
    - Recall: Cost of repairing a fault rises as a function of *time between error and repair*

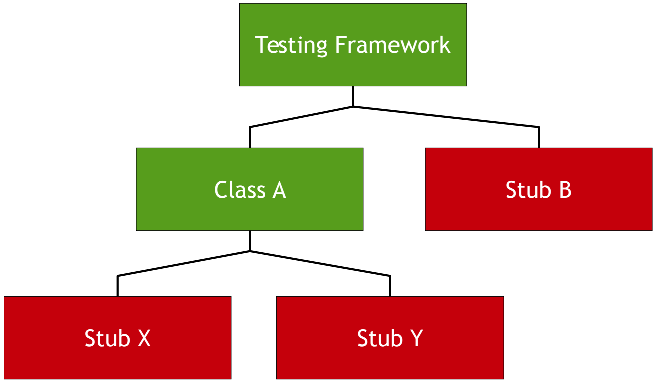


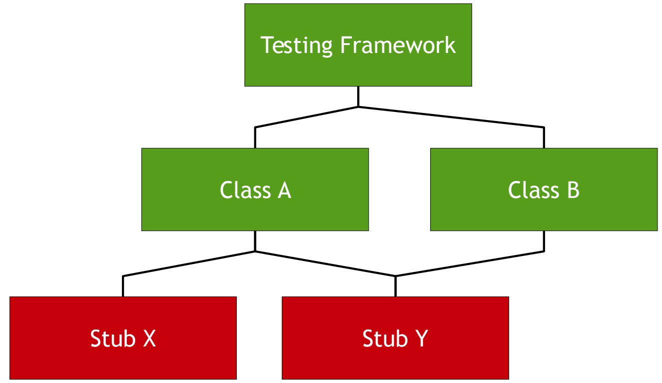
Structural and Functional Strategies

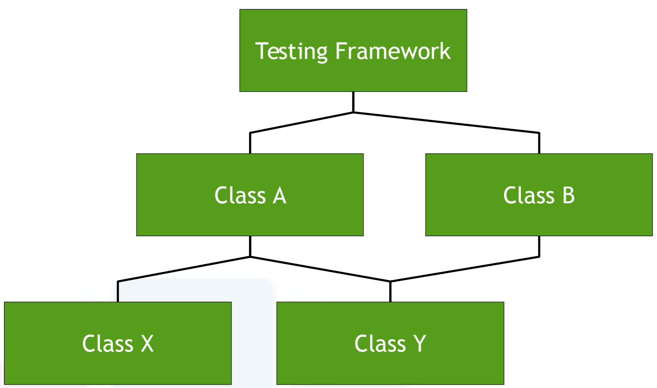
- Structural orientation:
  Modules constructed, integrated and tested based on a hierarchical project structure
  - Top-down, Bottom-up, Sandwich, Backbone
- Functional orientation:
  Modules integrated according to application characteristics or features
  - Threads, Critical module



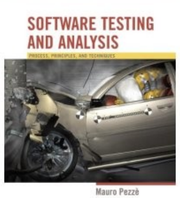
Top-Down



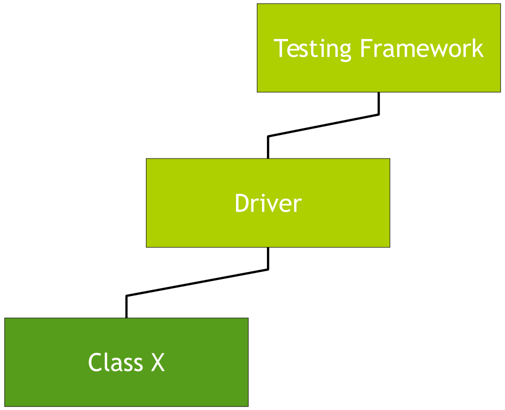


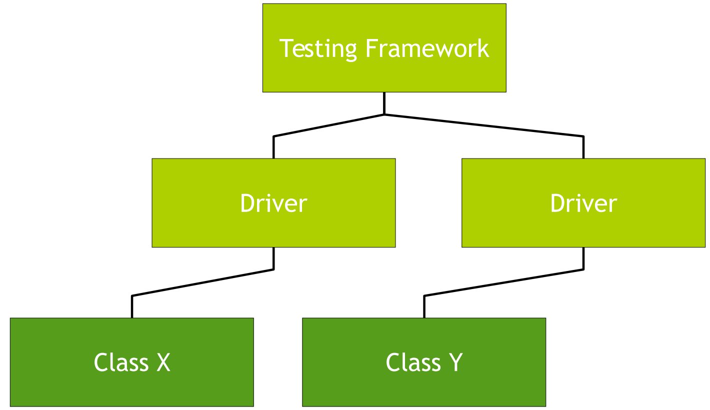


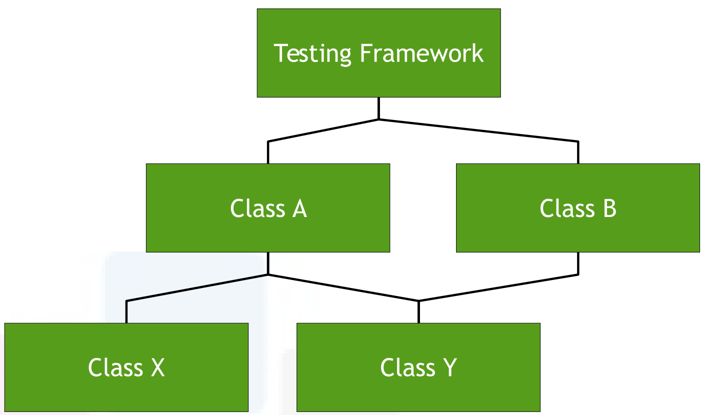
- Working from the top level (in terms of “use” or “include” relation) toward the bottom.
- No drivers required if program tested from top-level interface (e.g. GUI, CLI, web app, etc.)
- Write stubs of called or used modules at each step in construction
- As modules replace stubs, more functionality is testable
- ...until the program is complete, and all functionality can be tested



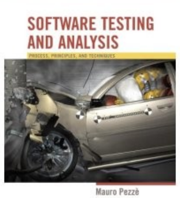
Bottom-Up .



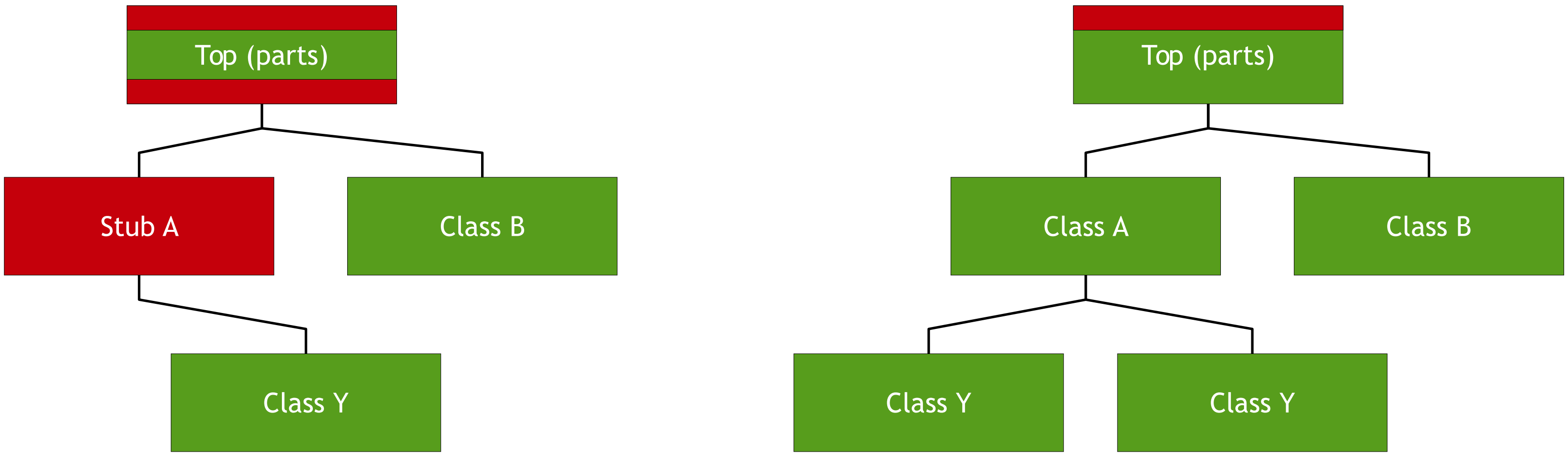




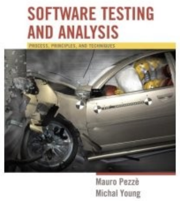
- Starting at the leaves of the "uses" hierarchy, we never need stubs
- ... but we must construct drivers for each module (as in unit testing) ...
- ... an intermediate module replaces a driver, and needs its own driver
- so we may have several working subsystems that are eventually integrated into a single system.



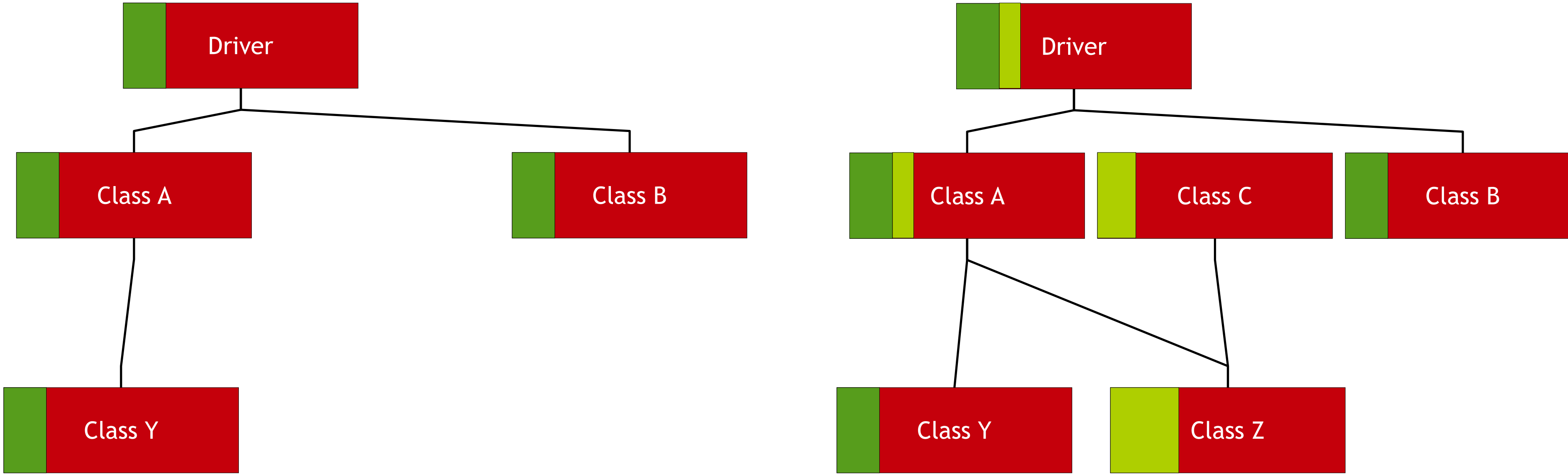
Sandwich



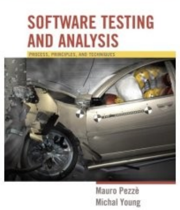
- Working from the extremes (top and bottom) toward center, we may use fewer drivers and stubs
- Sandwich integration is flexible and adaptable, but complex to plan



Thread

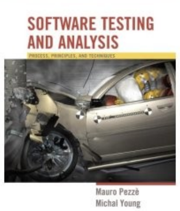


- A "thread" is a portion of several modules that together provide a user-visible program feature.
- Integrating one thread, then another, etc., we maximize visibility for the user
- As in sandwich integration testing, we can minimize stubs and drivers, but the integration plan may be complex



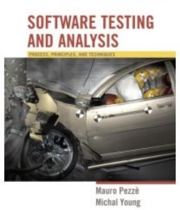
Critical Modules

- Strategy: Start with riskiest modules
  - Risk assessment is necessary first step
  - May include technical risks (is X feasible?), process risks (is schedule for X realistic?), other risks
- May resemble thread or sandwich process in tactics for flexible build order
  - E.g., constructing parts of one module to test functionality in another
- Key point is risk-oriented process
  - Integration testing as a risk-reduction activity, designed to deliver any bad news as early as possible



Choosing a Strategy

- Functional strategies require more planning
  - Structural strategies (bottom up, top down, sandwich) are **simpler**
  - But thread and critical modules testing provide **better process visibility**, especially in complex systems
- Possible to combine
  - Top-down, bottom-up, or sandwich are reasonable for relatively **small components and subsystems**
  - Combinations of thread and critical modules integration testing are **often preferred for larger subsystems**

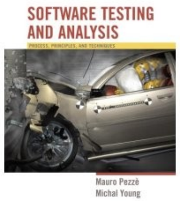


Specific Issues in Testing Object Oriented Software

OO definitions of unit and integration testing

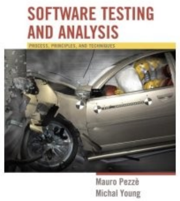
- **Procedural software**
  - **unit** = single program, function, or procedure
    more often: a unit of work that may correspond to one or more intertwined functions or programs
- **Object oriented software**
  - **unit** = class or (small) cluster of strongly related classes
    (e.g., sets of Java classes that correspond to exceptions)
  - unit testing = **intra-class testing**
  - integration testing = **inter-class testing** (cluster of classes)

→ dealing with single methods separately is usually too expensive (complex scaffolding), ***so methods are usually tested in the context of the class they belong to***



"Unit" in Unit Testing

- The Unit in Unit Testing is usually a class, however, there are specific issues that need to be taken into account when considering OO:
  - State dependent behavior
  - Encapsulation
  - Inheritance
  - Polymorphism and dynamic binding
  - Abstract and generic classes
  - Exception handling



“Isolated” calls: the combinatorial explosion problem

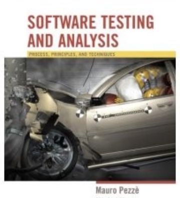
```
 abstract class Credit {
...
   abstract boolean validateCredit( Account a, int amt, CreditCard c);
...
}
```

- EduCredit
- BizCredit
- IndividualCredit

- USAccount
- UKAccount
- EUAccount
- JPAccount
- OtherAccount

- VISACard
- AmExpCard
- StoreCard

The combinatorial problem: **3 x 5 x 3 = 45** possible combinations of dynamic bindings (just for this one method!)

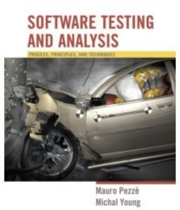


The combinatorial approach

Identify a set of combinations that cover all pairwise combinations of dynamic bindings

*Same motivation as **pairwise specification-based testing***
*the idea is that **instead of considering all combinations** we just have **pair-wise combinations** and add the third option later so we have 15 test cases instead of 45…*
The assumption is that very often failures are given by just combination of factors

Account	Credit	creditCard
USAccount	EduCredit	VISACard
USAccount	BizCredit	AmExpCard
USAccount	individualCredit	ChipmunkCard
UKAccount	EduCredit	AmExpCard
UKAccount	BizCredit	VISACard
UKAccount	individualCredit	ChipmunkCard
EUAccount	EduCredit	ChipmunkCard
EUAccount	BizCredit	AmExpCard
EUAccount	individualCredit	VISACard
JPAccount	EduCredit	VISACard
JPAccount	BizCredit	ChipmunkCard
JPAccount	individualCredit	AmExpCard
OtherAccount	EduCredit	ChipmunkCard
OtherAccount	BizCredit	VISACard
OtherAccount	individualCredit	AmExpCard



Combined calls: undesired effects

```
public abstract class Account { ...
    public int getYTDPurchased() {
  if (ytdPurchasedValid) { return ytdPurchased; }
  int totalPurchased = 0;
  for (Enumeration e = subsidiaries.elements() ;
  e.hasMoreElements(); )
      {   Account subsidiary = (Account) e.nextElement();
  totalPurchased += subsidiary.getYTDPurchased();
      }
  for (Enumeration e = customers.elements();
  e.hasMoreElements(); )
      {   Customer aCust = (Customer) e.nextElement();
  totalPurchased += aCust.getYearlyPurchase();
      }
  ytdPurchased = totalPurchased;
  ytdPurchasedValid = true;
  return totalPurchased;
    }  …  }
```

Problem:
different implementations of methods getYDTPurchased refer to different currencies.

A Data Flow Approach

```
public abstract class Account {
...
   public int getYTDPurchased() {
            if (ytdPurchasedValid) { return ytdPurchased; }
            int totalPurchased = 0;
            for (Enumeration e = subsidiaries.elements() ; e.hasMoreElements(); )
              {
                       Account subsidiary = (Account) e.nextElement();
                       totalPurchased += subsidiary.getYTDPurchased();
              }
            for (Enumeration e = customers.elements(); e.hasMoreElements(); )
              {
                       Customer aCust = (Customer) e.nextElement();
                       totalPurchased += aCust.getYearlyPurchase();
              }
            ytdPurchased = totalPurchased;
            ytdPurchasedValid = true;
            return totalPurchased;
   }
...
}
```

totalPurchased defined

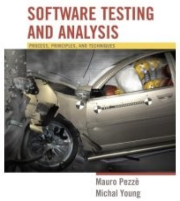
step 1: identify polymorphic calls, binding sets, defs and uses

totalPurchased used and defined

totalPurchased used and defined

totalPurchased used

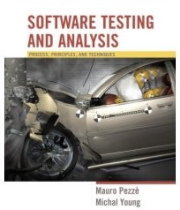
totalPurchased used



Def-Use (dataflow) testing of polymorphic calls

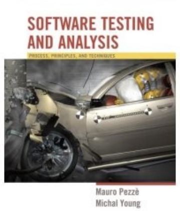
- Derive a test case for each possible polymorphic <def,use> pair
  - Each binding must be considered individually
  - Pairwise combinatorial selection may help in reducing the set of test cases

- *Example*: Dynamic binding of currency
  - We need test cases that bind the different calls to different methods *in the same run*
  - We can reveal faults due to the use of different currencies in different methods



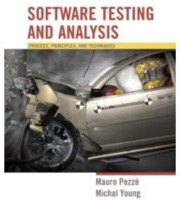
Inheritance

- When testing a subclass ...
  - We would like to re-test only what has not been thoroughly tested in the parent class
    - for example, no need to test hashCode and getClass methods inherited from class Object in Java
  - But we should test any method whose behavior may have changed
    - even accidentally!

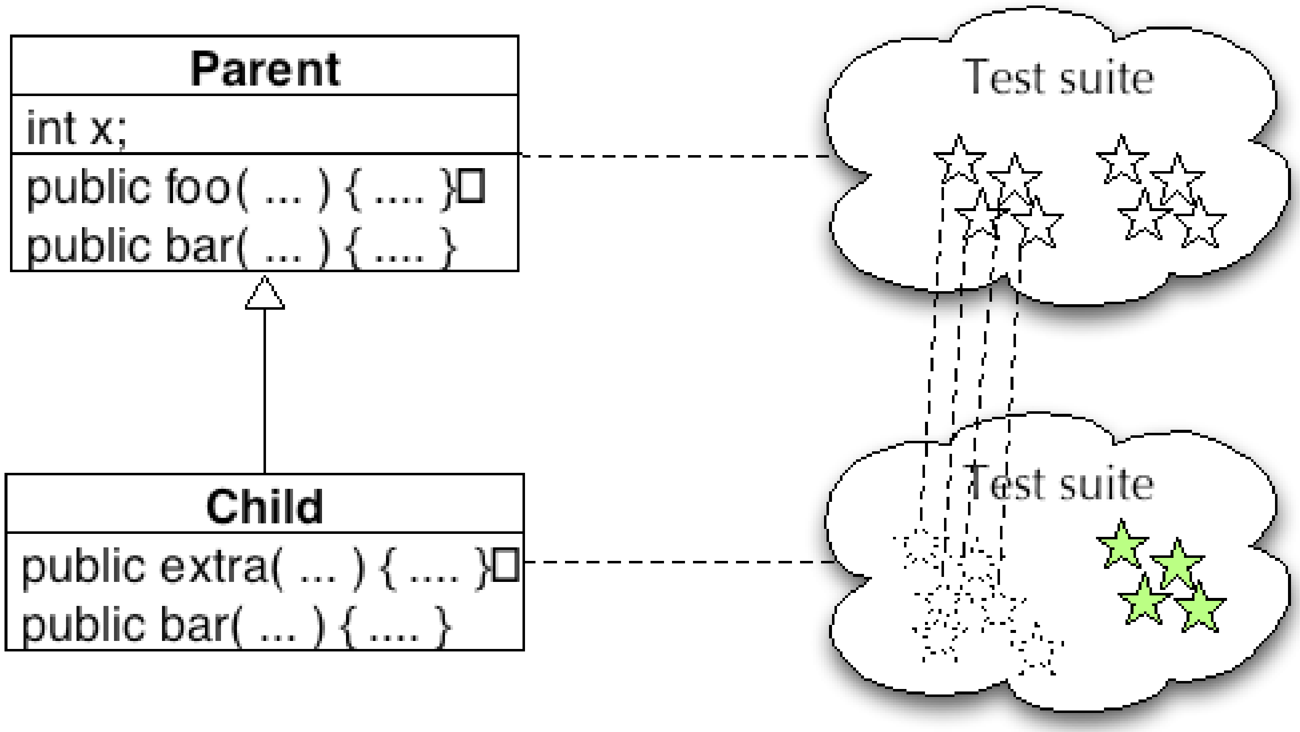


Reusing Tests with the Testing History Approach

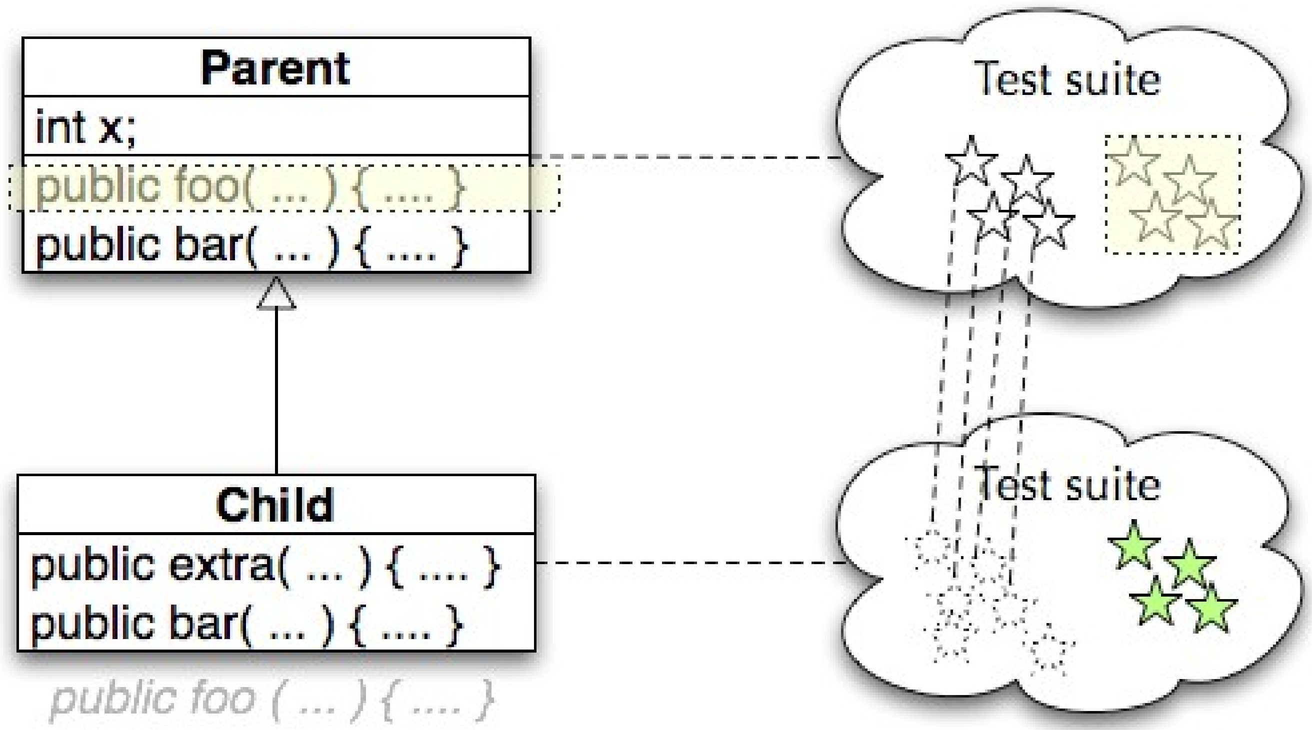
- Track test suites and test executions
  - determine which new tests are needed
  - determine which old tests must be re-executed
- New and changed behavior ...
  - new methods must be tested
  - redefined methods must be tested, but we can partially reuse test suites defined for the ancestor
  - other inherited methods do not have to be retested



Testing history

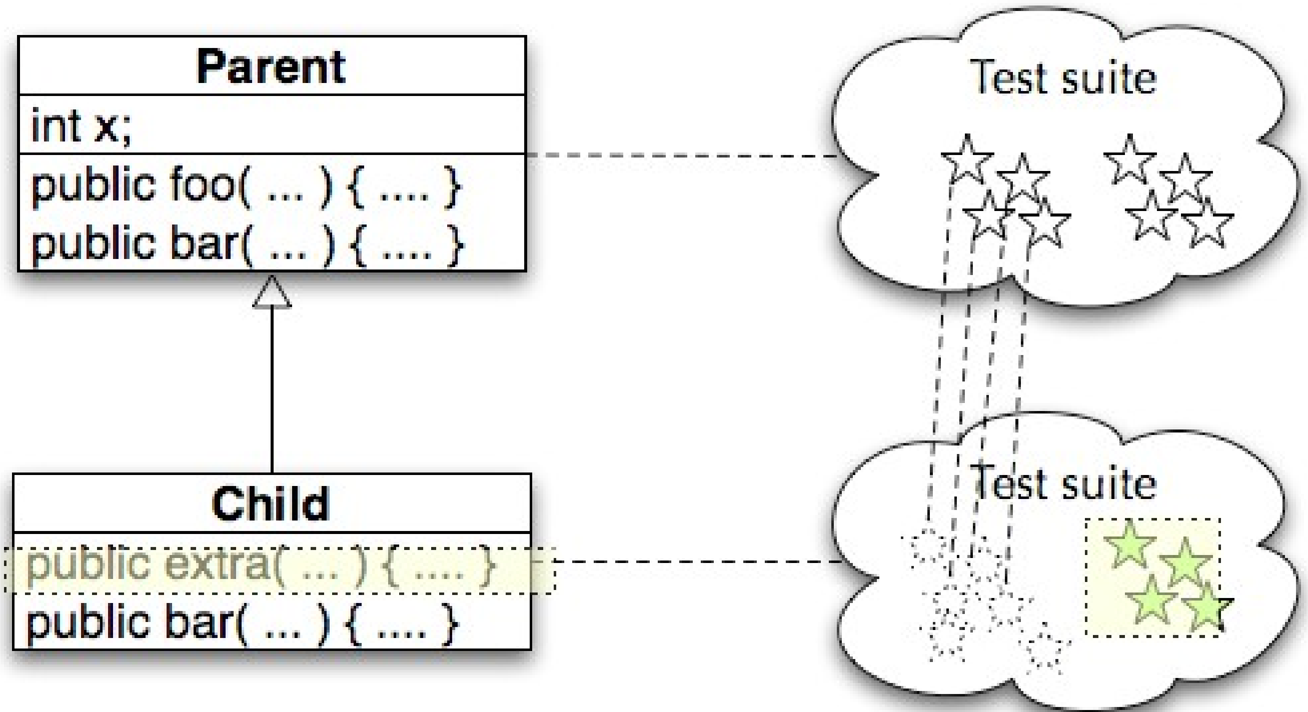


Inherited, unchanged



Inherited, unchanged ("recursive"):
No need to re-test

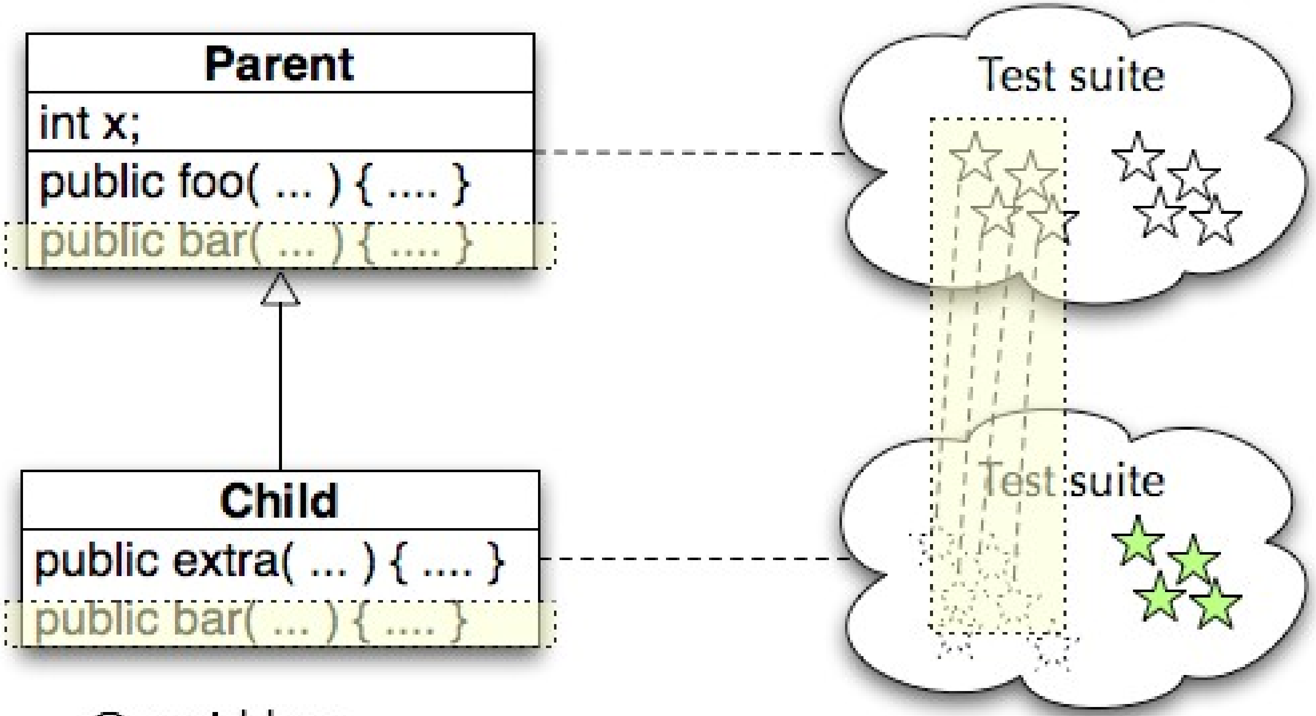
Newly introduced methods



New:

Design and execute new test cases

Overridden methods

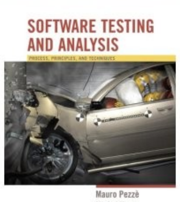


Overridden:

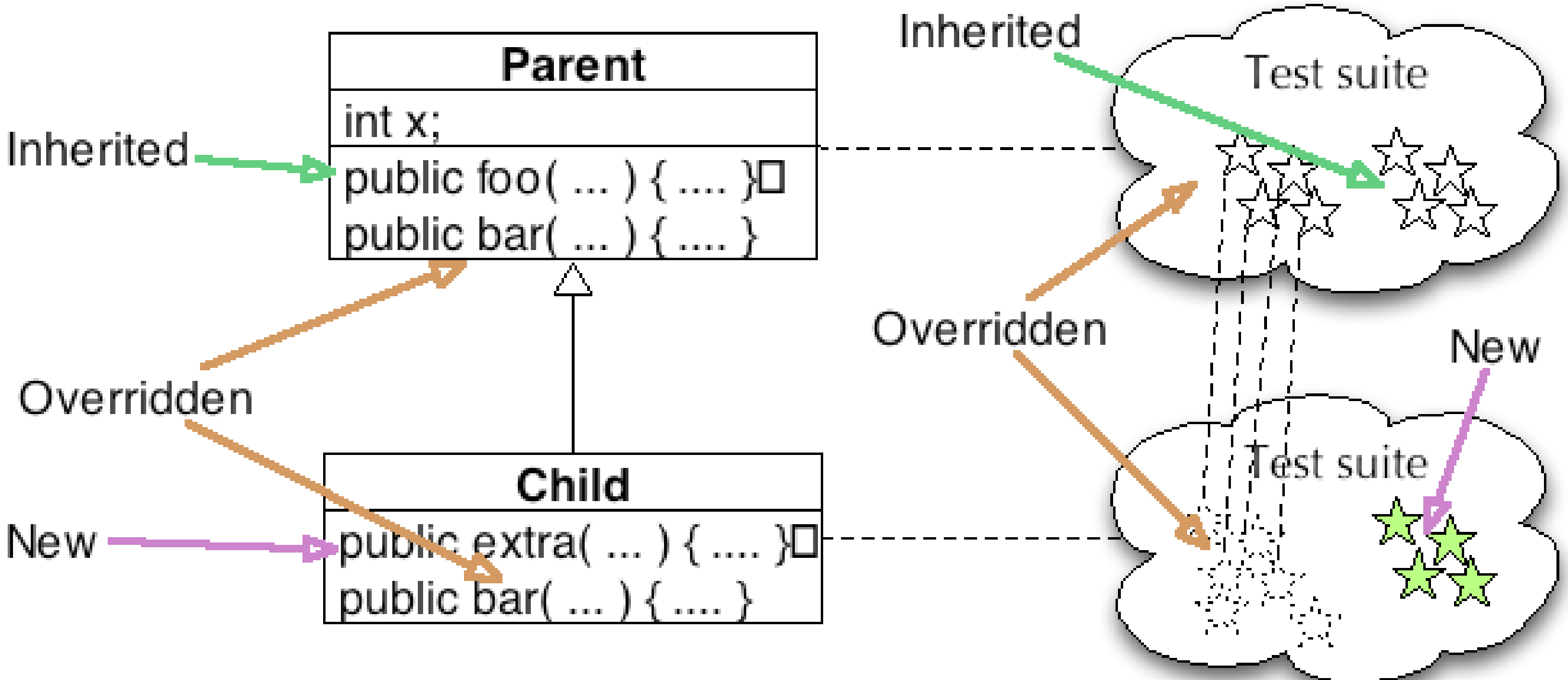
Re-execute test cases from parent, add new test cases as needed

Testing history – some details

- Abstract methods (and classes)
  - Design test cases when abstract method is introduced (even if it can’t be executed yet)
- Behavior changes
  - Should we consider a method “redefined” if another new or redefined method changes its behavior?
    - The standard “testing history” approach does not do this
    - It might be reasonable combination of data flow (structural) OO testing with the (functional) testing history approach

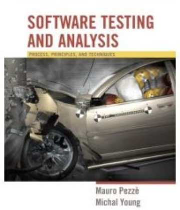


Testing History - Summary



Does Testing History help?

- Executing test cases should (usually) be cheap
  - It may be simpler to re-execute the full test suite of the parent class
  - ... but still add to it for the same reasons
- But sometimes execution is not cheap ...
  - Example: Control of physical devices
  - Or very large test suites
    - Ex: Some Microsoft product test suites require more than one night (so daily build cannot be fully tested)
  - Then some use of testing history is profitable



Testing Generic Classes

A generic class

```
class PriorityQueue<Elem Implements Comparable> {...}
```

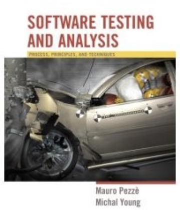
is designed to be instantiated with many different parameter types

```
PriorityQueue<Customers>
PriorityQueue<Tasks>
```

A generic class is typically designed to behave consistently some set of permitted parameter types.

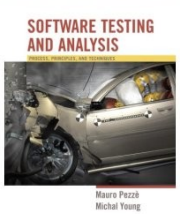
Testing can be broken into two parts

- Showing that some instantiation is correct
- showing that all permitted instantiations behave consistently



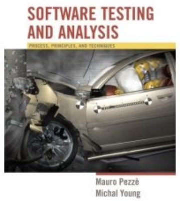
Show that some instantiation is correct

- Design tests as if the parameter were copied textually into the body of the generic class.
  - We need source code for both the generic class and the parameter class



Identify (possible) interactions

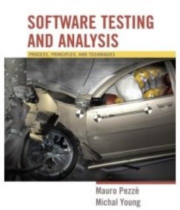
- Identify potential interactions between generic and its parameters
  - Identify potential interactions by inspection or analysis, not testing
  - Look for: method calls on parameter object, access to parameter fields, possible indirect dependence
  - Easy case is no interactions at all (e.g., a simple container class)
- Where interactions are possible, they will need to be tested



Example Interaction

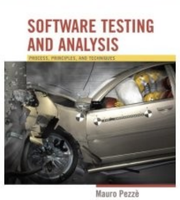
```
class PriorityQueue
   <Elem implements Comparable> {...}
```

- Priority queue uses the “Comparable” interface of Elem to make method calls on the generic parameter
- We need to establish that it does so consistently
  - So that if priority queue works for one kind of Comparable element, we can have some confidence it does so for others



Testing variation in instantiation

- We can’t test every possible instantiation
  - Just as we can’t test every possible program input
- ... but there is a contract (a specification) between the generic class and its parameters
  - Example: “implements Comparable” is a specification of possible instantiations
  - Other contracts may be written only as comments
- Functional (specification-based) testing techniques are appropriate
  - Identify and then systematically test properties implied by the specification



lasaris

Example: Testing variation in instantiation

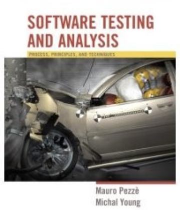
Most but not all classes that implement Comparable also satisfy the rule

```
(x.compareTo(y) == 0) == (x.equals(y))
```

(from java.lang.Comparable)

So test cases for PriorityQueue should include

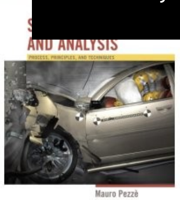
- instantiations with classes that do obey this rule:
  `class String`
- instantiations that violate the rule:
  `class BigDecimal` with values `4.0` and `4.00`



Exception handling

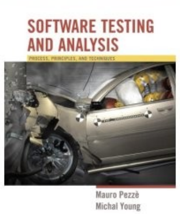
```
void addCustomer(Customer theCust) {
   customers.add(theCust);
     }
     public static Account
   newAccount(...)
   throws InvalidRegionException
     {
   Account thisAccount = null;
   String regionAbbrev = Regions.regionOfCountry(
          mailAddress.getCountry());
   if (regionAbbrev == Regions.US) {
       thisAccount = new USAccount();
   } else if (regionAbbrev == Regions.UK) {
       ....
   } else if (regionAbbrev == Regions.Invalid) {
       throw new InvalidRegionException(mailAddress.getCountry());
   }
...
     }
```

exceptions create implicit control flows and may be handled by different handlers



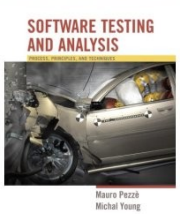
Testing Exception Handling

- Impractical to treat exceptions like normal flow
  - too many flows: every array subscript reference, every memory allocation, every cast, ...
  - multiplied by matching them to every handler that could appear immediately above them on the call stack.
  - many actually impossible
- So we separate testing exceptions
  - and ignore program error exceptions (test to prevent them, not to handle them)
- What we do test: Each exception handler, and each explicit throw or re-throw of an exception



Testing program exception handlers

- Local exception handlers
  - test the exception handler (consider a subset of points bound to the handler)
- Non-local exception handlers
  - Difficult to determine all pairings of <points, handlers>
  - So enforce (and test for) a design rule:
    if a method propagates an exception, the method call should have *no other effect*

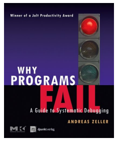


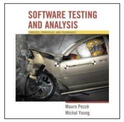
References

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.





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

lasaris