

Functional Testing

Software Engineering
Andreas Zeller • Saarland University

From Pressman, "Software Engineering – a practitioner's approach", Chapter 14 and Pezze + Young, "Software Testing and Analysis", Chapters 10-11

Today, we'll talk about testing – how to test software. The question is: How do we design tests? And we'll start with functional testing.

1

Exam



- 13:00–16:00
- 5 rooms
- Check your e-mail!

2

Exam Material



- One A4 page ("cheat sheet"), handwritten
- A pen
- Nothing else (no books, PDAs, etc.)
- Reference material (e.g. Z operators) provided with exam

3



7

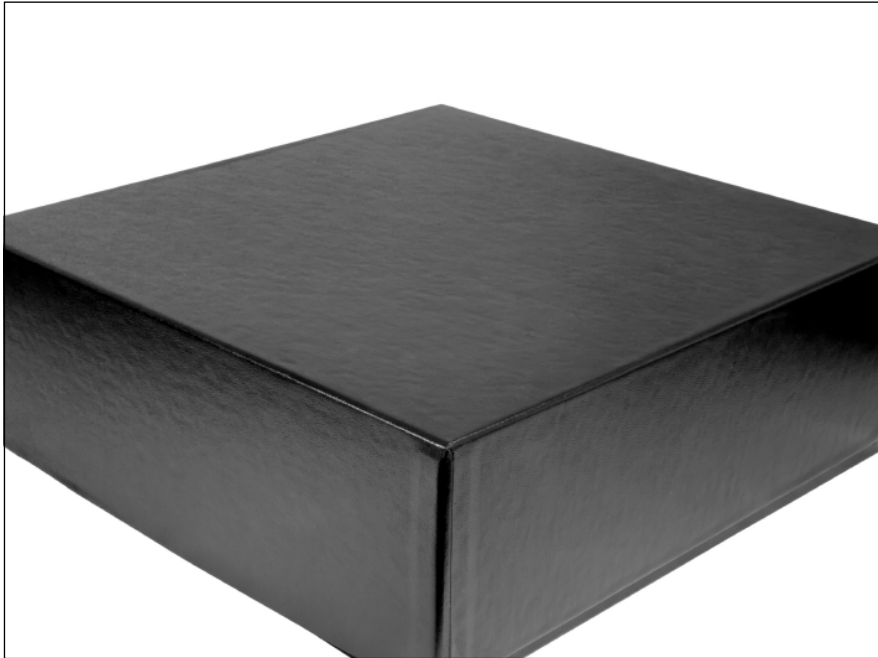
Evaluation next Thursday!



8

From Pressman, “Software Engineering – a practitioner’s approach”, Chapter 14 and Pezze + Young, “Software Testing and Analysis”, Chapters 10-11

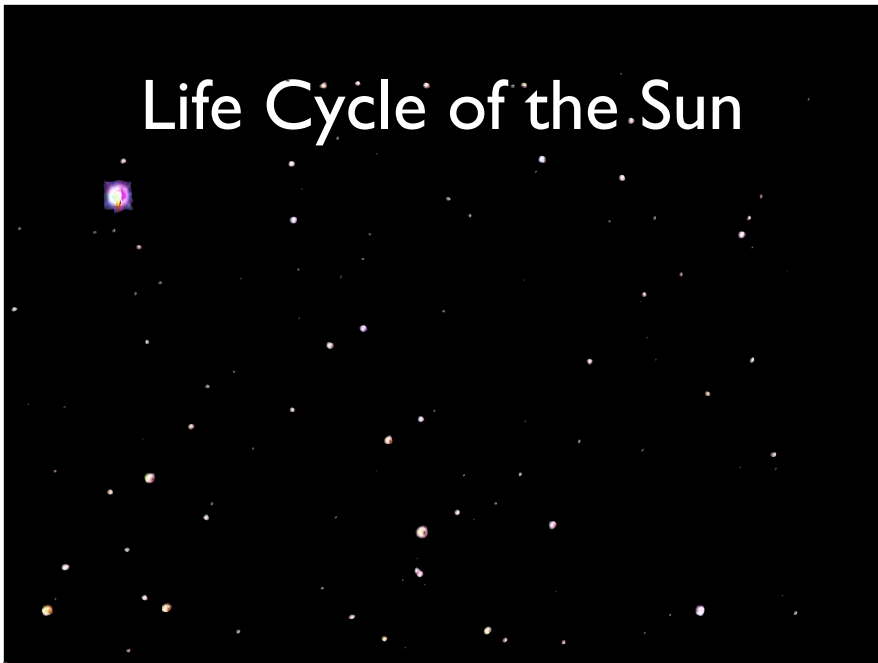
Today, we’ll talk about testing – how to test software. The question is: How do we design tests? And we’ll start with functional testing.



9

Functional testing is also called “black-box” testing, because we see the program as a black box – that is, we ignore how it is being written

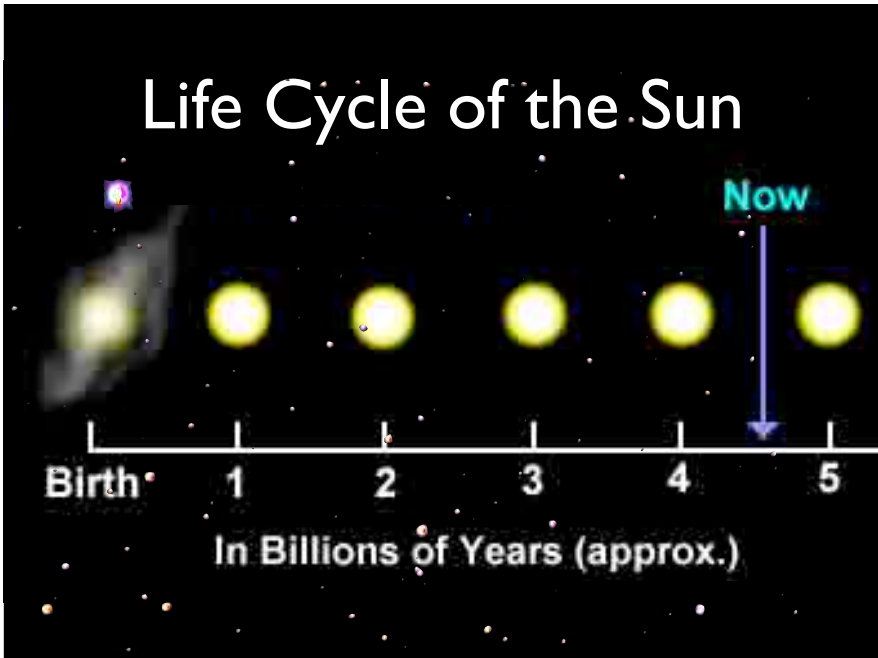
Life Cycle of the Sun



Note that in 900 million years, due to increase of the luminosity of the sun, CO₂ levels will be toxic for plants; in 1.9 billion years, surface water will have evaporated (source: Wikipedia on "Earth")

16

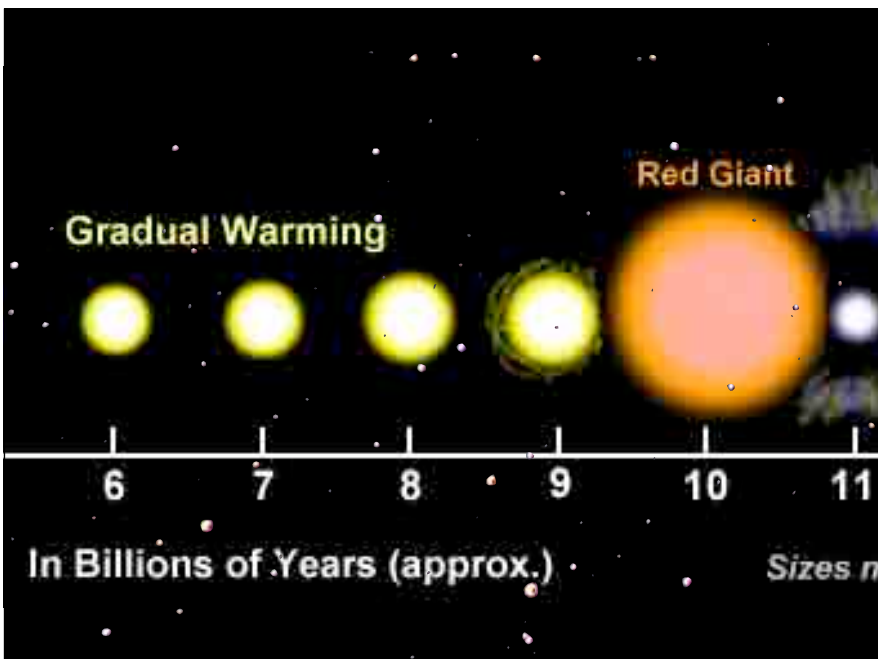
Life Cycle of the Sun



Note that in 900 million years, due to increase of the luminosity of the sun, CO₂ levels will be toxic for plants; in 1.9 billion years, surface water will have evaporated (source: Wikipedia on "Earth")

17

Gradual Warming



None of this is crucial for the computation, though.

18

Why not Random?

- Defects are not distributed uniformly
- Assume **Roots** applies quadratic equation

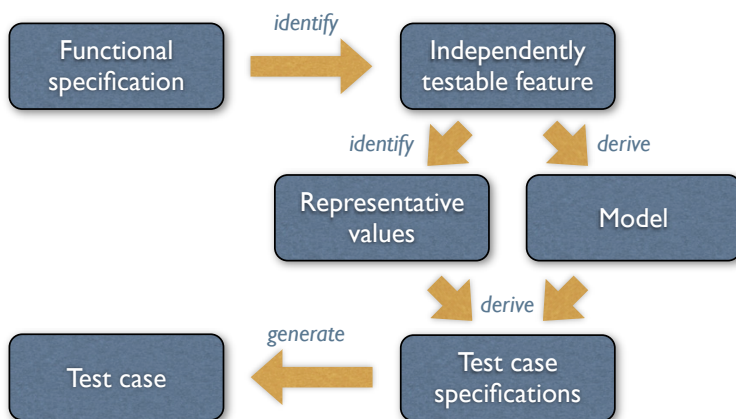
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

and fails if $b^2 - 4ac = 0$ and $a = 0$

- Random sampling is unlikely to choose $a = 0$ and $b = 0$

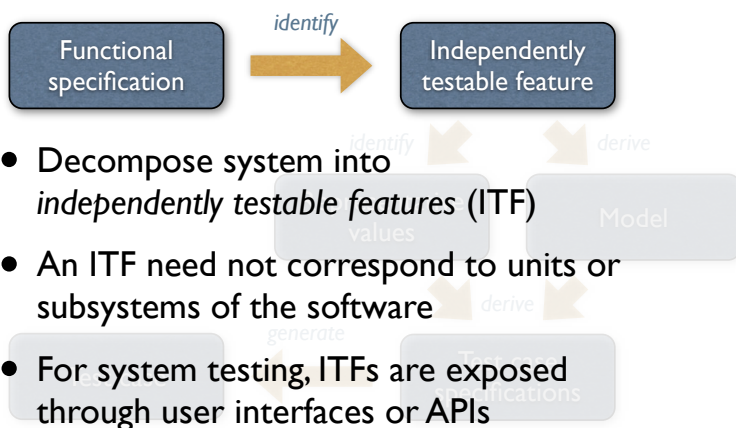
However, it is not. For one, we don't care for bias – we specifically want to search where it matters most. Second, random testing is unlikely to uncover specific defects. Therefore, we go for *functional testing*.

Systematic Functional Testing



The main steps of a systematic approach to functional program testing (from Pezze + Young, "Software Testing and Analysis", Chapter 10)

Testable Features



- Decompose system into *independently testable features (ITF)*
- An ITF need not correspond to units or subsystems of the software
- For system testing, ITFs are exposed through user interfaces or APIs

Testable Features

```
class Roots {  
    // Solve  $ax^2 + bx + c = 0$   
    public roots(double a, double b, double c)  
    { ... }  
  
    // Result: values for x  
    double root_one, root_two;  
}
```



- What are the independently testable features?

Just one – roots is a unit and thus provides exactly one single testable feature.

Testable Features



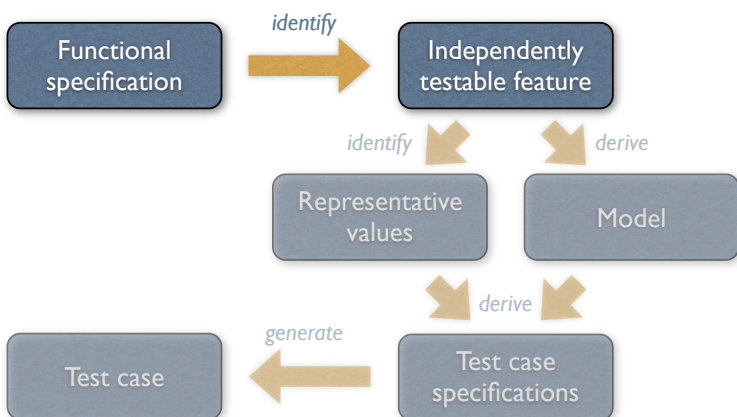
- Consider a multi-function calculator
- What are the independently testable features?



Every single function becomes an independently testable feature. Some functions (like memory access, for instance) are dependent on each other, though: to retrieve a value, you must first store it.

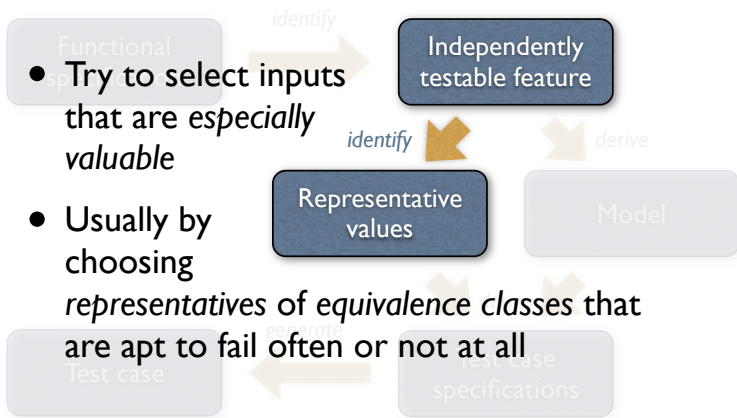
(Note how the calculator shows the #years required for the Roots calculation.)

Testable Features



The main steps of a systematic approach to functional program testing (from Pezze + Young, “Software Testing and Analysis”, Chapter 10)

Representative Values



- Try to select inputs that are especially valuable
- Usually by choosing representatives of equivalence classes that are apt to fail often or not at all

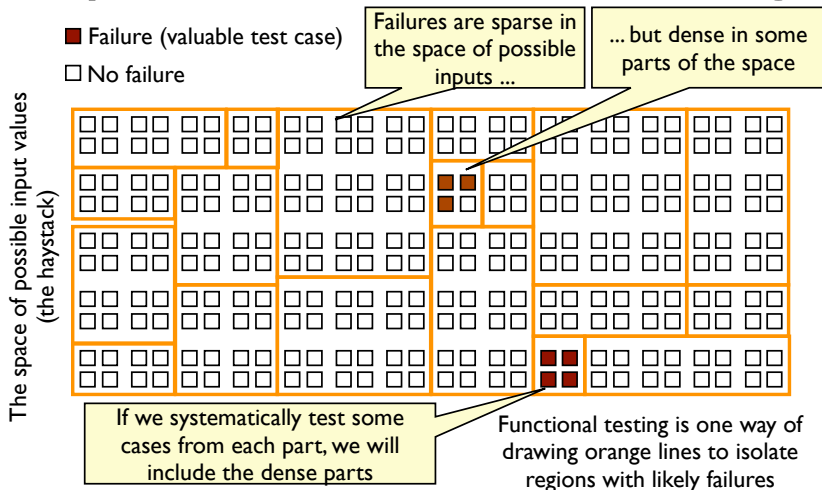
The main steps of a systematic approach to functional program testing (from Pezze + Young, "Software Testing and Analysis", Chapter 10)

Needles in a Haystack

- To find needles, look systematically
- We need to find out what makes needles special



Systematic Partition Testing



We can think of all the possible input values to a program as little boxes ... white boxes that the program processes correctly, and colored boxes on which the program fails. Our problem is that there are a lot of boxes ... a huge number, and the colored boxes are just an infinitesimal fraction of the whole set. If we reach in and pull out boxes at random, we are unlikely to find the colored ones. Systematic testing says: Let's not pull them out at random. Let's first subdivide the big bag of boxes into smaller groups (the pink lines), and do it in a way that tends to concentrate the colored boxes in a few of the groups. The number of groups needs to be much smaller than the number of boxes, so that we can systematically reach into each group to pick one or a few boxes. Functional testing is one variety of partition testing, a way of drawing the orange lines so that, when one of the boxes within a orange group is a failure, many of the other boxes in that group may also be failures. Functional testing means using the program specification to draw pink lines. (from Pezze + Young, "Software Testing and Analysis", Chapter 10)

Valid ZIP Codes

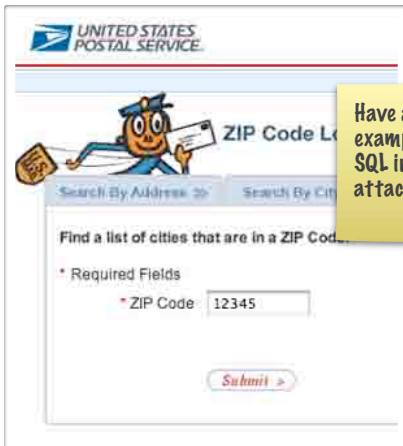


1. with 0 cities as output (0 is boundary value)
2. with 1 city as output
3. with many cities as output

34

(from Pezze + Young, "Software Testing and Analysis", Chapter 10)

Invalid ZIP Codes



4. empty input
5. 1-4 characters (4 is boundary value)
6. 6 characters (6 is boundary value)
7. very long input
8. no digits
9. non-character data

35

(from Pezze + Young, "Software Testing and Analysis", Chapter 10)

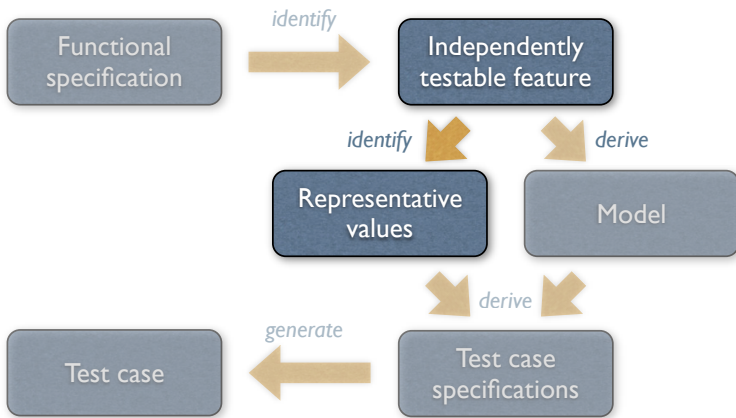
Gutjahr's Hypothesis

Partition testing is more effective than random testing.

Generally, random inputs are easier to generate, but less likely to cover parts of the specification or the code. See Gutjahr (1999) in IEEE Transactions on Software Engineering 25, 5 (1999), 661-667

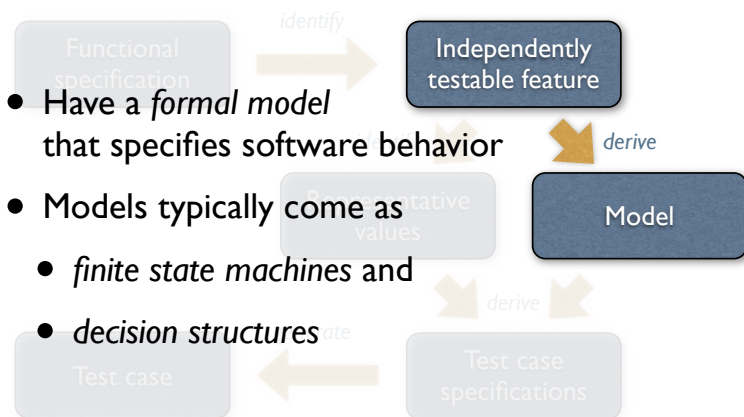
36

Representative Values



The main steps of a systematic approach to functional program testing (from Pezze + Young, "Software Testing and Analysis", Chapter 10)

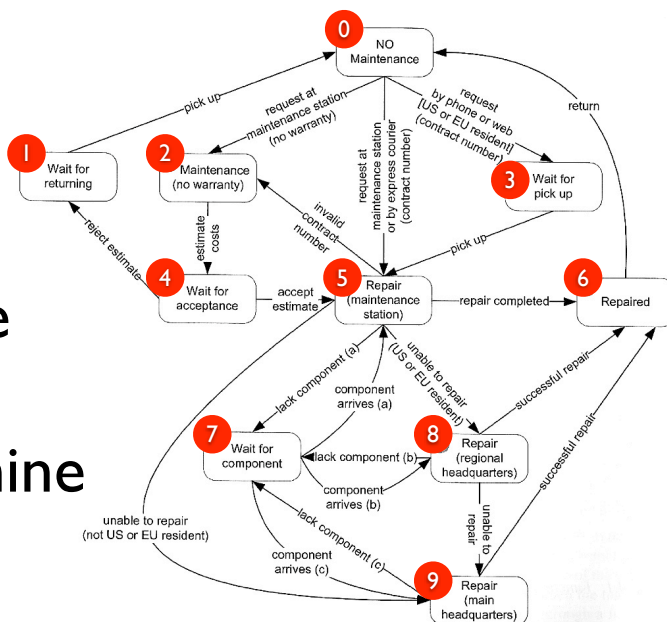
Model-Based Testing



- Have a *formal model* that specifies software behavior
- Models typically come as
 - *finite state machines* and
 - *decision structures*

The main steps of a systematic approach to functional program testing (from Pezze + Young, "Software Testing and Analysis", Chapter 10)

Finite State Machine



As an example, consider these steps modeling a product maintenance process... (from Pezze + Young, "Software Testing and Analysis", Chapter 14)

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.

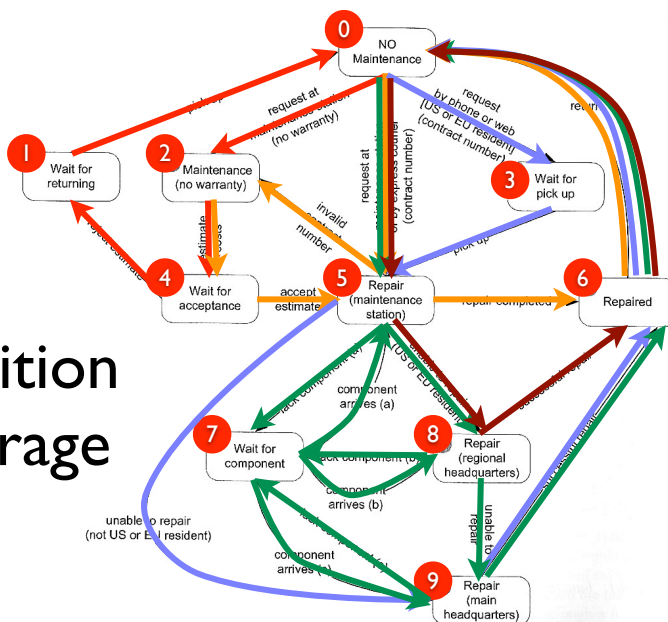
...based on these (informal) requirements

(from Pezze + Young, "Software Testing and Analysis", Chapter 14)

Coverage Criteria

- **Path coverage:** Tests cover every path
Not feasible in practice due to infinite number of paths
- **State coverage:** Every node is executed
A minimum testing criterion
- **Transition coverage:** Every edge is executed
Typically, a good coverage criterion to aim for

Transition Coverage



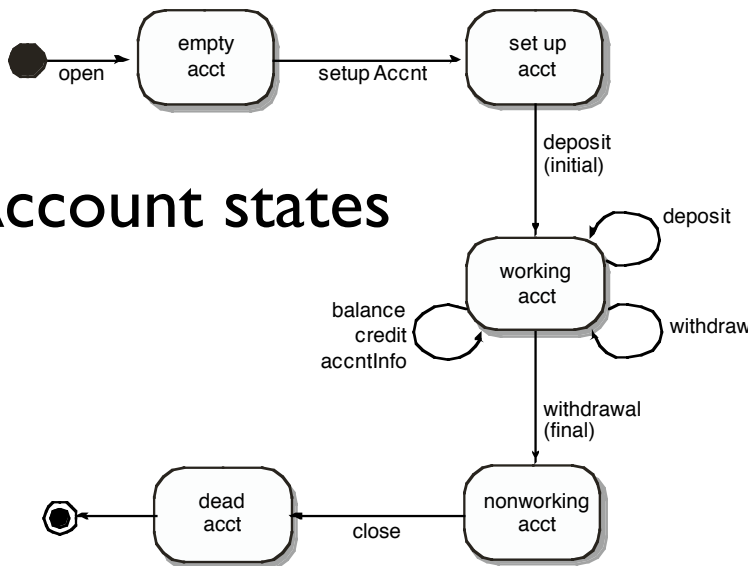
With five test cases (one color each), we can achieve transition coverage (from Pezze + Young, "Software Testing and Analysis", Chapter 14)

State-based Testing

- Protocols (e.g., network communication)
- GUIs (sequences of interactions)
- Objects (methods and states)

Finite state machines can be used to model for a large variety of behaviors – and thus serve as a base for testing.

Account states

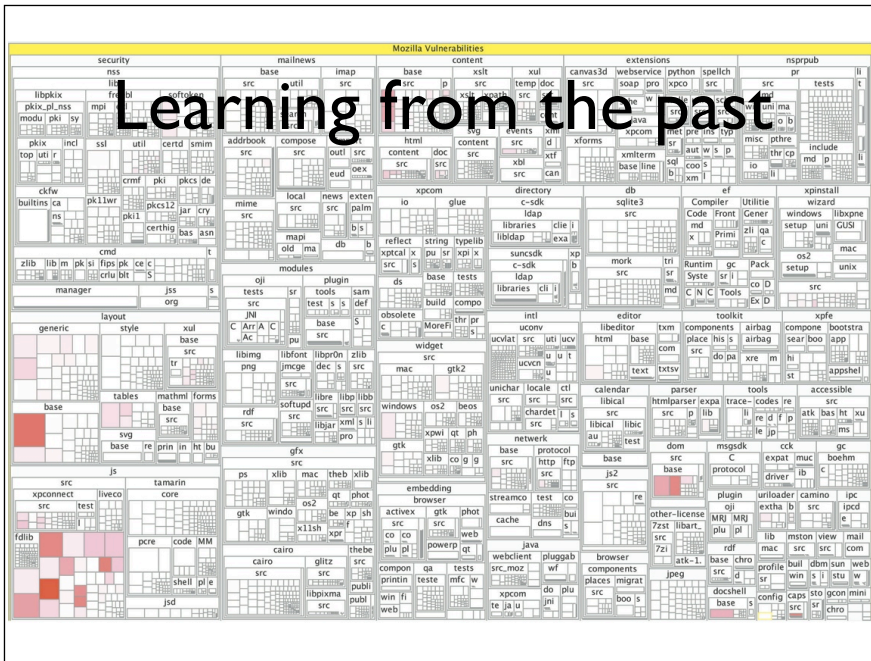


Here's an example of a finite state machine representing an Account class going through a number of states. Transition coverage means testing each Account method once. (From Pressman, "Software Engineering – a practitioner's approach", Chapter 14)

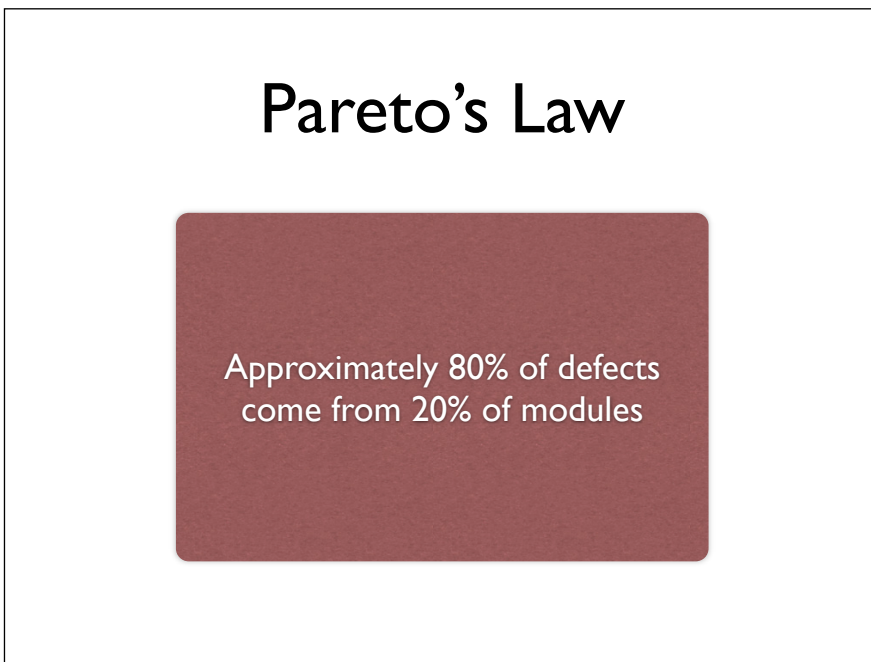
Decision Tables

	Education		Individual					
Education account	T	T	F	F	F	F	F	F
Current purchase > Threshold 1	-	-	F	F	T	T	-	-
Current purchase > Threshold 2	-	-	-	-	F	F	T	T
Special price < scheduled price	F	T	F	T	-	-	-	-
Special price < Tier 1	-	-	-	-	F	T	-	-
Special price < Tier 2	-	-	-	-	-	-	F	T
Out	Edu discount	Special price	No discount	Special price	Tier 1 discount	Special price	Tier 2 discount	Special Price

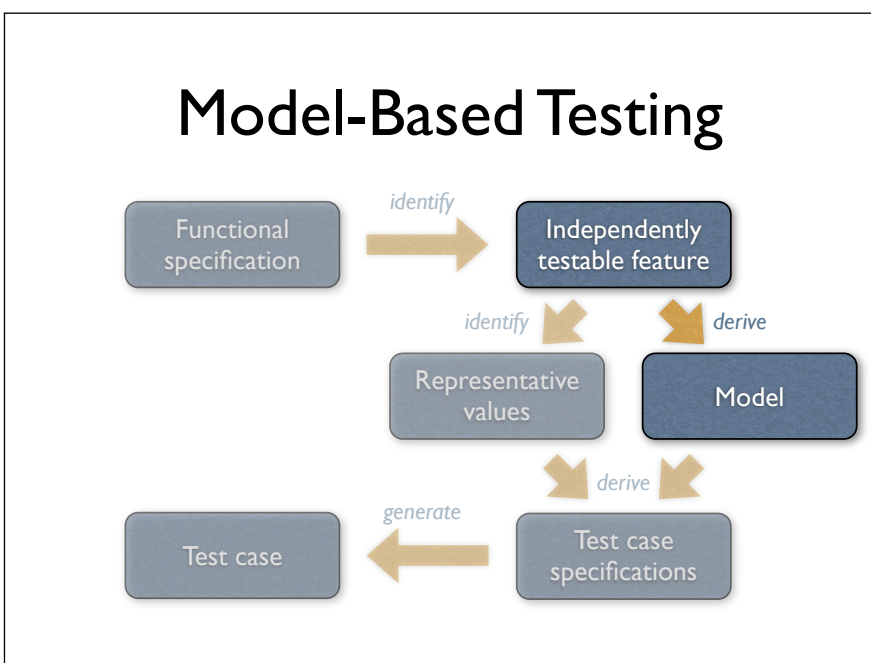
A decision table describes under which conditions a specific outcome comes to be. This decision table, for instance, determines the discount for a purchase, depending on specific thresholds for the amount purchased. (from Pezze + Young, "Software Testing and Analysis", Chapter 14)



To decide where to put most effort in testing, one can also examine the past – i.e., where did most defects occur in the past. The above picture shows the distribution of security vulnerabilities in Firefox – the redder a rectangle, the more vulnerabilities, and therefore a likely candidate for intensive testing. The group of Andreas Zeller at Saarland University researches how to mine such information automatically and how to predict future defects.



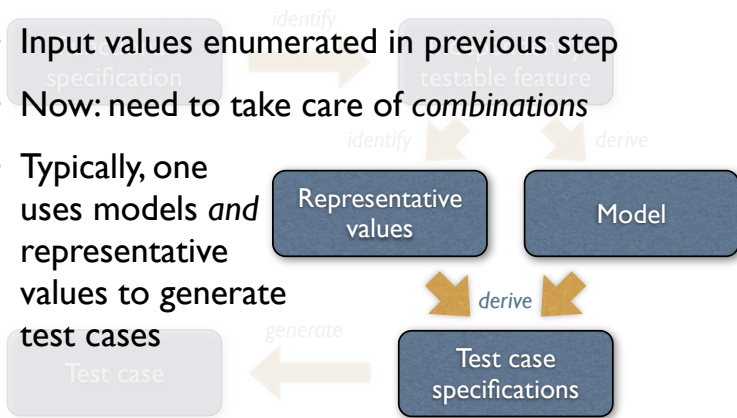
Evidence: several studies, including Zeller's own evidence :-)



The main steps of a systematic approach to functional program testing (from Pezze + Young, "Software Testing and Analysis", Chapter 10)

Deriving Test Case Specs

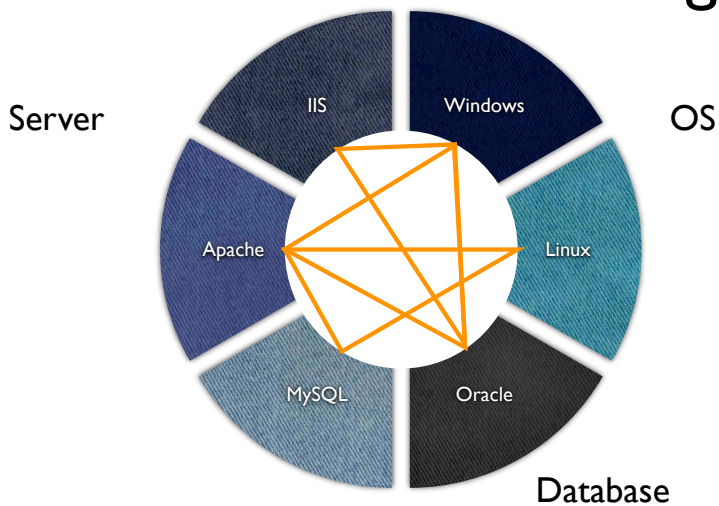
- Input values enumerated in previous step
- Now: need to take care of combinations
- Typically, one uses models and representative values to generate test cases



The main steps of a systematic approach to functional program testing (from Pezze + Young, "Software Testing and Analysis", Chapter 10)

55

Combinatorial Testing



Many domains come as a combination of individual inputs. We therefore need to cope with a combinatorial explosion.

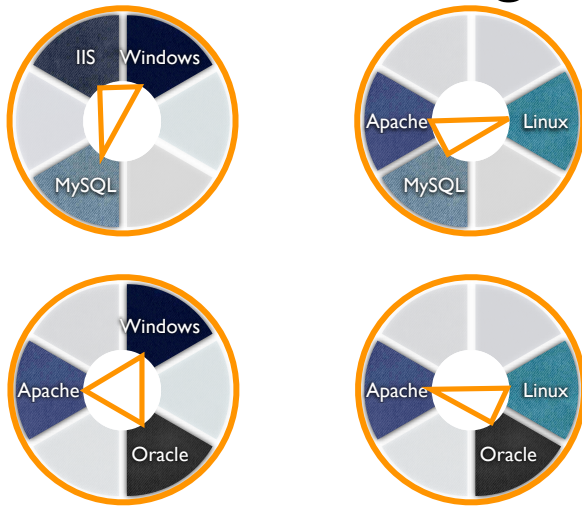
56

Combinatorial Testing

- Eliminate invalid combinations
IIS only runs on Windows, for example
- Cover *all pairs* of combinations
such as MySQL on Windows and Linux
- Combinations typically generated automatically
and – hopefully – tested automatically, too

57

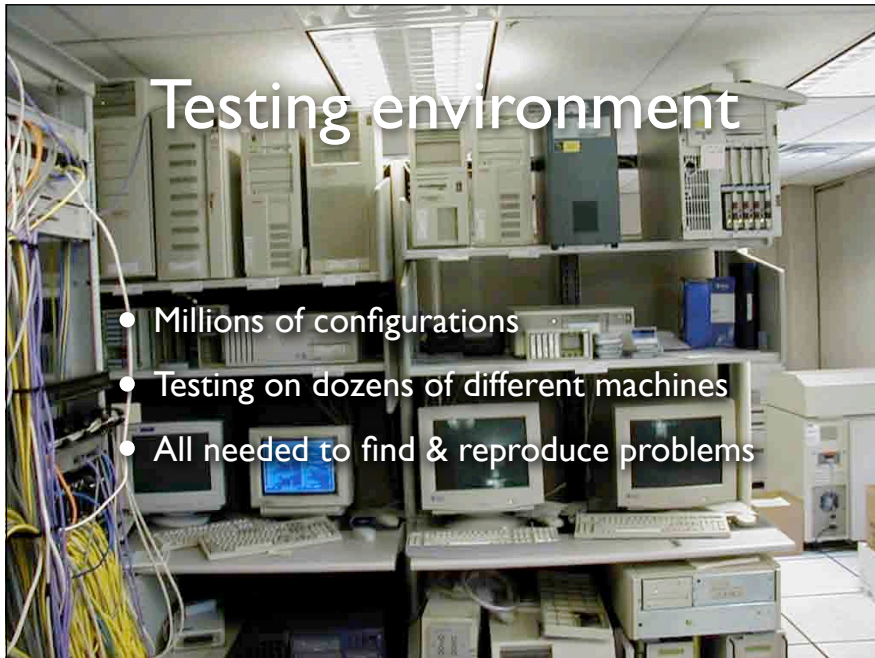
Pairwise Testing



Pairwise testing means to cover every single pair of configurations

58

Testing environment



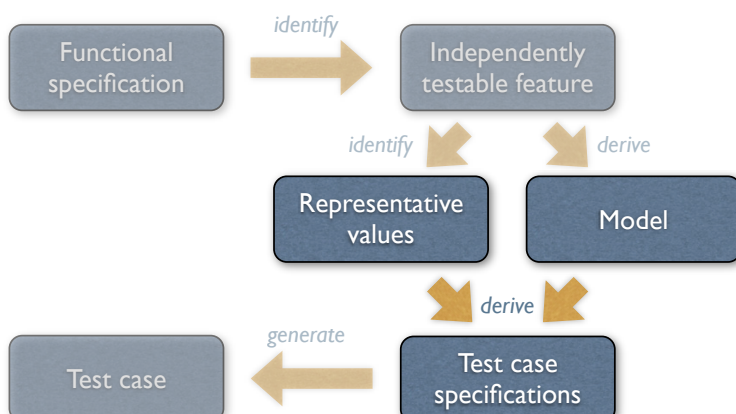
- Millions of configurations
- Testing on dozens of different machines
- All needed to find & reproduce problems

In practice, such testing needs hundreds and hundreds of PCs in every possible configuration – Microsoft, for instance, has entire buildings filled with every hardware imaginable

Source: <http://www.ci.newton.ma.us/MIS/Network.htm>

59

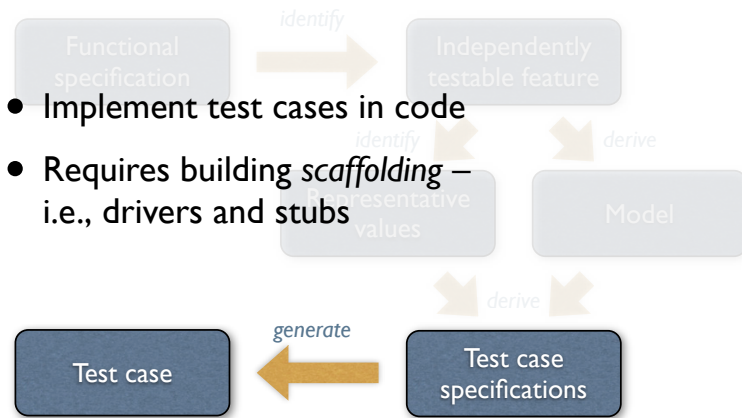
Deriving Test Case Specs



The main steps of a systematic approach to functional program testing (from Pezze + Young, "Software Testing and Analysis", Chapter 10)

60

Deriving Test Cases



- Implement test cases in code
- Requires building *scaffolding* – i.e., drivers and stubs

The main steps of a systematic approach to functional program testing (from Pezze + Young, “Software Testing and Analysis”, Chapter 10)

Unit Tests

- Directly access units (= classes, modules, components...) at their programming interfaces
- Encapsulate a set of tests as a single syntactical unit
- Available for all programming languages (JUNIT for Java, CPPUNIT for C++, etc.)

Here’s an example for automated unit tests – the well-known JUnit

Running a Test

A test case...

1. sets up an environment for the test
2. tests the unit
3. tears down the environment again.

The environment provides the stubs such that a JUnit test case can work. The JUnit test case is the driver.

Testing a URL Class

<http://www.askigor.org/status.php?id=sample>



As an example, consider parsing a URL

64

```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class URLTest extends TestCase {
    private URL askigor_url;

    // Create new test
    public URLTest(String name) { super(name); }

    // Assign a name to this test case
    public String toString() { return getName(); }

    // Setup environment
    protected void setUp() {
        askigor_url = new URL("http://www.askigor.org/" +
            "status.php?id=sample"); }

    // Release environment
    protected void tearDown() { askigor_url = null;}
```

The `setUp()` and `tearDown()` functions set up the environment...

65

```
// Test for protocol (http, ftp, etc.)
public void testProtocol() {
    assertEquals(askigor_url.getProtocol(), "http");
}

// Test for host
public void testHost() {
    int noPort = -1;
    assertEquals(askigor_url.getHost(), "www.askigor.org");
    assertEquals(askigor_url.getPort(), noPort);
}

// Test for path
public void testPath() {
    assertEquals(askigor_url.getPath(), "/status.php");
}

// Test for query part
public void testQuery() {
    assertEquals(askigor_url.getQuery(), "id=sample");
}
```

This functional test can be used as a *specification!*

...while the `test*()` methods perform the actual tests.

66

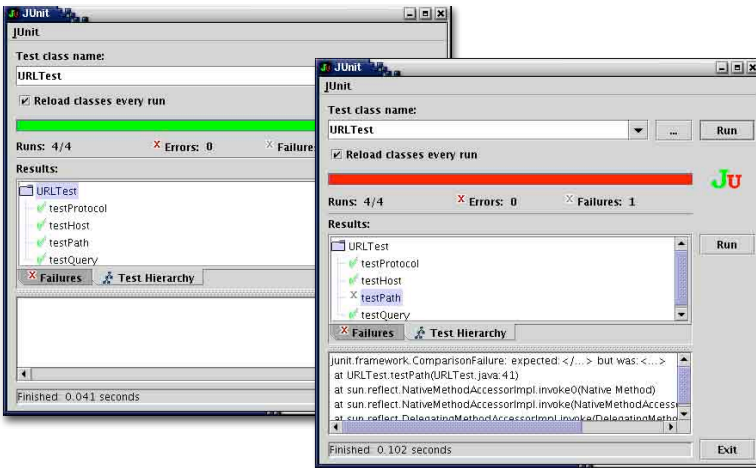
```

// Set up a suite of tests
public static Test suite() {
    TestSuite suite = new TestSuite(URLTest.class);
    return suite;
}

// Main method: Invokes GUI
public static void main(String args[]) {
    String[] testCaseName =
        { URLTest.class.getName() };
    // junit.textui.TestRunner.main(testCaseName);
    junit.swingui.TestRunner.main(testCaseName);
    // junit.awtui.TestRunner.main(testCaseName);
}
}

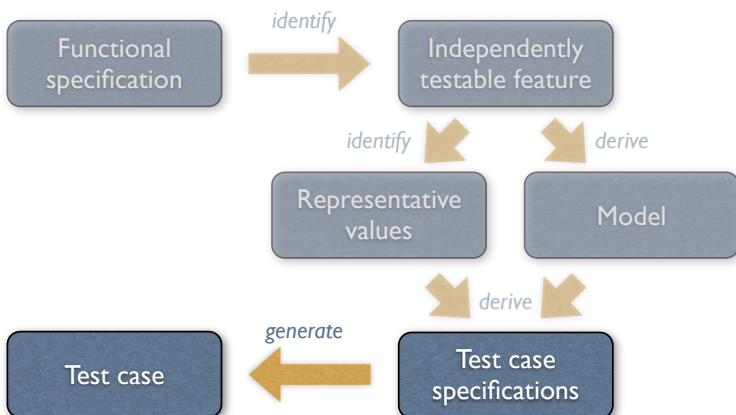
```

JUnit



JUnit comes with a GUI – and is frequently integrated in programming environments

Deriving Test Cases



The main steps of a systematic approach to functional program testing (from Pezze + Young, "Software Testing and Analysis", Chapter 10)
