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.

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, in contrast to structural or “white-box” testing, where the program is the base.
Testing Tactics

- Tests based on spec
- Test covers as much specified behavior as possible
- Tests based on code
- Test covers as much implemented behavior as possible

Why Functional?

- Program code not necessary
- Early functional test design has benefits:
  - reveals spec problems
  - assesses testability
  - gives additional explanation of spec
  - may even serve as spec, as in XP
Why Functional?

- Best for *missing logic* defects
  Common problem: Some program logic was simply forgotten
  Structural testing would not focus on code that is not there

- Applies at all granularity levels
  unit tests • integration tests • system tests • regression tests

A Challenge

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;
   }
}

- Which values for $a, b, c$ should we test?
  assuming $a, b, c$ were 32-bit integers, we’d have $(2^{32})^3 \approx 10^{28}$ legal inputs
  with 1.000.000.000.000 tests/s, we would still require 2.5 billion years

Functional testing applies at all granularity levels (in contrast to structural testing, which only applies to unit and integration testing)

Life Cycle of the Sun

2,510,588,971 years, 32 days, and 20 hours to be precise.

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”)
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”)

None of this is crucial for the computation, though.
A Challenge

```java
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;
    }

    • Which values for a, b, c should we test?
      assuming a, b, c, were 32-bit integers, we'd have (2^{32})^3 \approx 10^{28} legal inputs
      with 1.000.000.000.000 tests/s, we would still require 2.5 billion years
}
```

Random Testing

• Pick possible inputs uniformly
• Avoids designer bias
  A real problem: The test designer can make the same logical mistakes and bad assumptions as the program designer
    (especially if they are the same person)
• But treats all inputs as equally valuable

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 \[b = 0\]

One might think that picking random samples might be a good idea.

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?
Testable Features

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

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

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.)

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

- Failure (valuable test case)
- No failure

The space of possible input values (the haystack)

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 orange lines to isolate regions with likely failures

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 an 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)

Equivalence Partitioning

<table>
<thead>
<tr>
<th>Input condition</th>
<th>Equivalence classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>range</td>
<td>one valid, two invalid (larger and smaller)</td>
</tr>
<tr>
<td>specific value</td>
<td>one valid, two invalid (larger and smaller)</td>
</tr>
<tr>
<td>member of a set</td>
<td>one valid, one invalid</td>
</tr>
<tr>
<td>boolean</td>
<td>one valid, one invalid</td>
</tr>
</tbody>
</table>

How do we choose equivalence classes? The key is to examine input conditions from the spec. Each input condition induces an equivalence class – valid and invalid inputs.
Boundary Analysis

Possible test case

- Test at lower range (valid and invalid), at higher range (valid and invalid), and at center

Example: ZIP Code

- Input: 5-digit ZIP code
- Output: list of cities
- What are representative values to test?

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

How do we choose representatives from equivalence classes? A greater number of errors occurs at the boundaries of an equivalence class rather than at the “center”. Therefore, we specifically look for values that are at the boundaries — both of the input domain as well as at the output.

(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

“Special” ZIP Codes

- How about a ZIP code that reads
  12345'; DROP TABLE orders; SELECT *
  FROM zipcodes WHERE ‘zip’ = ‘
- Or a ZIP code with 65536 characters...
- This is security testing

Gutjahr’s Hypothesis

Partition testing is more effective than random testing.

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

Generally, random inputs are easier to generate, but less likely to cover parts of the specification or the code.
The main steps of a systematic approach to functional program testing (from Pezze + Young, “Software Testing and Analysis”, Chapter 10)

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

As an example, consider these steps modeling a product maintenance process… (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

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.

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.

Decision Tables

<table>
<thead>
<tr>
<th>Education account</th>
<th>T</th>
<th>T</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current purchase &gt; Threshold 1</td>
<td>-</td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Current purchase &gt; Threshold 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>Special price &lt; scheduled price</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Special price &lt; Tier 1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>F</td>
<td>T</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Special price &lt; Tier 2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>F</td>
<td>T</td>
<td>-</td>
</tr>
</tbody>
</table>

(from Pezze + Young, “Software Testing and Analysis”, Chapter 14)
Condition Coverage

- **Basic criterion**: Test every column
  "Don't care" entries (–) can take arbitrary values
- **Compound criterion**: Test every combination
  Requires $2^n$ tests for $n$ conditions and is unrealistic
- **Modified condition decision criterion (MCDC)**:
  like basic criterion, but additionally, modify each T/F value at least once
  Again, a good coverage criterion to aim for

Nicolas and I were going through the slides and found that in the Functional testing lecture, on slide 39, the Basic criterion is swapped with the Compound criterion description, at least from what we know from the Structural testing chapter from the Pezze+Young book. Are we

We modify the individual values in column 1 and 2 to generate four additional test cases – but these are already tested anyway. For instance, the modified values in column 1 are already tested in column 3.

This also applies to changing the other values, so adding additional test cases is not necessary in this case.

<table>
<thead>
<tr>
<th>MCDC Criterion</th>
<th>Education</th>
<th>Individual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Education account</td>
<td>F T F F F F F F</td>
<td>F T F F F F F F</td>
</tr>
<tr>
<td>Current purchase &gt; Threshold 1</td>
<td>– – F F T T – –</td>
<td>– – F F T T – –</td>
</tr>
<tr>
<td>Current purchase &gt; Threshold 2</td>
<td>– – – – F F T T</td>
<td>– – – – F F T T</td>
</tr>
<tr>
<td>Special price &lt; scheduled price</td>
<td>F T F T – – – –</td>
<td>F T F T – – – –</td>
</tr>
<tr>
<td>Special price &lt; Tier 1</td>
<td>– – – – – – T –</td>
<td>– – – – – – T –</td>
</tr>
<tr>
<td>Special price &lt; Tier 2</td>
<td>– – – – – – – F</td>
<td>– – – – – – – F</td>
</tr>
<tr>
<td>Out</td>
<td>Edu discount</td>
<td>Special price</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MCDC Criterion</th>
<th>Education</th>
<th>Individual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Education account</td>
<td>T T F F F F F F</td>
<td>T T F F F F F F</td>
</tr>
<tr>
<td>Current purchase &gt; Threshold 1</td>
<td>– – F F T T – –</td>
<td>– – F F T T – –</td>
</tr>
<tr>
<td>Current purchase &gt; Threshold 2</td>
<td>– – – – F F T T</td>
<td>– – – – F F T T</td>
</tr>
<tr>
<td>Special price &lt; scheduled price</td>
<td>T T F T – – – –</td>
<td>T T F T – – – –</td>
</tr>
<tr>
<td>Special price &lt; Tier 1</td>
<td>– – – – – – T –</td>
<td>– – – – – – T –</td>
</tr>
<tr>
<td>Special price &lt; Tier 2</td>
<td>– – – – – – – F</td>
<td>– – – – – – – F</td>
</tr>
<tr>
<td>Out</td>
<td>Edu discount</td>
<td>Special price</td>
</tr>
</tbody>
</table>
### MCDC Criterion

<table>
<thead>
<tr>
<th></th>
<th>Education</th>
<th>Individual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Education account</td>
<td>T</td>
<td>F F F F F F F F</td>
</tr>
<tr>
<td>Current purchase &gt;</td>
<td>☒</td>
<td>Threshold 1</td>
</tr>
<tr>
<td>Threshold 1</td>
<td>—</td>
<td>— F F T T — —</td>
</tr>
<tr>
<td>Current purchase &gt;</td>
<td>— — — — —</td>
<td>Threshold 2</td>
</tr>
<tr>
<td>Threshold 2</td>
<td>— — — — —</td>
<td>— — — — — — —</td>
</tr>
<tr>
<td>Special price &lt;</td>
<td>F T F T</td>
<td>scheduled price</td>
</tr>
<tr>
<td>Tier 1</td>
<td>— — — — —</td>
<td>F T — — — — —</td>
</tr>
<tr>
<td>Special price &lt;</td>
<td>— — — — —</td>
<td>Tier 1 discount</td>
</tr>
<tr>
<td>Tier 2</td>
<td>— — — — —</td>
<td>Special price</td>
</tr>
<tr>
<td>Out</td>
<td>— — — — —</td>
<td>— — — — — — —</td>
</tr>
</tbody>
</table>

Established by a number of studies done by E. Weyuker at AT&T. “Any explicit relationship between coverage and error detection would mean that we have a fixed distribution of errors over all statements and paths, which is clearly not the case.”

### Weyuker’s Hypothesis

The adequacy of a coverage criterion can only be intuitively defined.
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

![Diagram showing the flow of deriving test case specifications](image)

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

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

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
Pairwise testing means to cover every single pair of configurations.

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

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

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

Test case specifications → Test case

Model → Derived feature

Representative values → Derived feature

Functional specification → Derived feature

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

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

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