Perspective of quality differs from one person to another. Also, it differs in the customers’ and developers’ perspectives.
Testing

- Testing: a procedure intended to establish the quality, performance, or reliability of something, esp. before it is taken into widespread use.

Software Testing

- Software testing: the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

Waterfall Model

(1968)

- Communication: project initiation, requirements.
- Planning: estimating, scheduling, tracking.
- Modeling: analysis, design.
- Construction: code, test.
- Deployment: delivery, support, feedback.

From Oxford dictionary

From Pressman, “Software Engineering – a practitioner’s approach”, Chapter 13

Let’s recall the Waterfall model.
In the second half of the course, we focus on construction and deployment — essentially, all the activities that take place after the code has been written.

So, we simply assume our code is done — but is it ready for release?
We built it!

It’s not like this is the ultimate horror…

…but still, this question causes fear, uncertainty and doubt in managers

Therefore, we focus on the “construction” stage – and more specifically, on the “test” in here.

Waterfall Model
(1968)
Waterfall Model
(1968)

Construction
code
test

Deployment
delivery
support
feedback

V&V

- Verification: Ensuring that software correctly implements a specific function
- Validation: Ensuring that software has been built according to customer requirements

These activities are summarized as V&V – verification and validation

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

• “if a user presses a request button at floor i, an available elevator must arrive at floor i soon”

Verification

• “if a user presses a request button at floor i, an available elevator must arrive at floor i within 30 seconds”

Basic Questions

• When do V&V start? When are they done?
• Which techniques should be applied?
• How do we know a product is ready?
• How can we control the quality of successive releases?
• How can we improve development?

Verification or validation depends on the spec – this one is unverifiable, but validatable (from Pezze + Young, “Software Testing and Analysis”)

this one is verifiable.
Waterfall Model (1968)

- Developers on software should do no testing at all
- Software should be “tossed over a wall” to strangers who will test it mercilessly
- Testers should get involved with the project only when testing is about to begin

Early descriptions of the waterfall model separated coding and testing into two different activities.

What do these facts have in common? They’re all wrong!

Verification and validation activities occur all over the software process (from Pezze + Young, “Software Testing and Analysis”)
V&V Activities

This is called the “V”-model of “V&V” activities (because of its shape) (from Pezze + Young, “Software Testing and Analysis”)

Unit Tests

- Uncover errors at module boundaries
- Typically written by programmer herself
- Frequently fully automatic (→ regression)

Stubs and Drivers

- A driver exercises a module’s functions
- A stub simulates not-yet-ready modules
- Frequently realized as mock objects

From Pressman, “Software Engineering – a practitioner’s approach”, Chapter 13
Integration Tests

- General idea: Constructing software while conducting tests
- Options: Big bang vs. incremental construction

Big Bang

- All components are combined in advance
- The entire program is tested as a whole
- Chaos results
- For every failure, the entire program must be taken into account

Top-Down Integration

- Top module is tested with stubs (and then used as driver)
- Stubs are replaced one at a time (“depth first”)
- As new modules are integrated, tests are re-run
- Allows for early demonstration of capability

This is called the “V”-model of “V&V” activities (because of its shape) (from Pezze + Young, “Software Testing and Analysis”)

From Pressman, “Software Engineering – a practitioner’s approach”, Chapter 13
Bottom-Up Integration

- Bottom modules implemented first and combined into clusters
- Drivers are replaced one at a time
- Removes the need for complex stubs

Sandwich Integration

- Combines bottom-up and top-down integration
- Top modules tested with stubs, bottom modules with drivers
- Combines the best of the two approaches

Evidence: pragmatic – there is no way a test can ever cover all possible paths through a program
Who Tests the Software?

Developer
- understands the system
- but will test gently
- driven by delivery

Independent Tester
- must learn about system
- will attempt to break it
- driven by quality

The Ideal Tester

The Developer

A good tester should be creative and destructive – even sadistic in places. — Gerald Weinberg, “The psychology of computer programming”

The conflict between developers and testers is usually overstated, though.

From Pressman, “Software Engineering – a practitioner’s approach”, Chapter 13
The Developers

Weinberg’s Law

A developer is unsuited to test his or her code.

Acceptance Testing

- Acceptance testing checks whether the contractual requirements are met
- Typically incremental
  (alpha test at production site, beta test at user’s site)
- Work is over when acceptance testing is done

Let’s simply say that developers should respect testers – and vice versa.

Theory: As humans want to be honest with themselves, developers are blindfolded with respect to their own mistakes.
Evidence: “seen again and again in every project” (Endres/Rombach)
From Gerald Weinberg, “The psychology of computer programming”
Special System Tests

- **Recovery testing**
  forces the software to fail in a variety of ways and verifies that recovery is properly performed

- **Security testing**
  verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration

- **Stress testing**
  executes a system in a manner that demands resources in abnormal quantity, frequency, or volume

- **Performance testing**
  test the run-time performance of software within the context of an integrated system

V&V Activities

This is called the “V”-model of “V&V” activities (because of its shape) (from Pezze + Young, “Software Testing and Analysis”)

Basic Questions

- When do V&V start? When are they done?
- Which techniques should be applied?
- How do we know a product is ready?
- How can we control the quality of successive releases?
- How can we improve development?
Why V&V is hard
(on software)

• Many different quality requirements
• Evolving (and deteriorating) structure
• Inherent non-linearity
• Uneven distribution of faults

There is a multitude of activities (dynamic ones execute the software, static ones don’t) – and we’d like them to end when the software is 100% correct. Unfortunately, none of them is perfect.

If an elevator can safely carry a load of 1000 kg, it can also safely carry any smaller load;
If a procedure correctly sorts a set of 256 elements, it may fail on a set of 255 or 53 or 12 elements, as well as on 257 or 1023.
(from Pezze + Young, “Software Testing and Analysis”)
The Curse of Testing

Dijkstra’s Law

Testing can show the presence but not the absence of errors

Static Analysis

We cannot tell whether this condition ever holds (halting problem)

The halting problem prevents us from matching lock(S)/unlock(S) — so our technique may be overly pessimistic. (from Pezze + Young, “Software Testing and Analysis”)
Pessimistic Inaccuracy

static void questionable() {
    int k;
    for (int i = 0; i < 10; i++)
        if (someCondition(i))
            k = 0;
        else
            k += 1;
    System.out.println(k);
}

• Is k being used uninitialized in this method?

You can’t always get what you want

• Correctness properties are undecidable
  the halting problem can be embedded in almost every property of interest

Simplified Properties

original problem                     simplified property

if ( .... ) {
    ...
    lock(S);
}  
    ...
if ( .... ) {
    ...
    unlock(S);
}  

Java prescribes a more restrictive, but statically checkable construct.

Static checking for match is necessarily inaccurate

synchronized(S) {
    ...
    ...
}  

An alternative is to go for a higher abstraction level
(from Pezze + Young, “Software Testing and Analysis”)

The Java compiler cannot tell whether someCondition() ever holds, so it
refuses the program (pessimistically) – even if someCondition(i) always returns true.
(from Pezze + Young, “Software Testing and Analysis”)
Simplified Properties

Static Verification

A proof can cover all runs – but only at a higher abstraction level

In some way, fear, uncertainty and doubt will thus prevail…
Hetzel-Myers Law

A combination of different V&V methods outperforms any single method alone.

Evidence: Various studies showed that different methods have strength in different application areas — in our picture, they would cover different parts of the program, different abstractions.

Trade-Offs

- We can be inaccurate (optimistic or pessimistic)...
- or we can simplify properties...
- but not all!

Evidence: Various studies showed that different methods have strength in different application areas — in our picture, they would cover different parts of the program, different abstractions.

and we have a wide range of techniques at our disposal (from Pezze + Young, Software Testing and Analysis).

But we can of course attempt to cover as many runs — and abstractions — as possible!
Basic Questions

• When do V&V start? When are they done?
• Which techniques should be applied?
• How do we know a product is ready?
• How can we control the quality of successive releases?
• How can we improve development?

Readiness in Practice

Let the customer test it :)
Readiness in Practice

Relative to a theoretically sound and experimentally validated statistical model, we have done sufficient testing to say with 95% confidence that the probability of 1,000 CPU hours of failure-free operation is \( \geq 0.995 \).

Basic Questions

- When do V&V start? When are they done?
- Which techniques should be applied?
- How do we know a product is ready?
- How can we control the quality of successive releases?
- How can we improve development?

Regression Tests

The idea is to have automated tests (here: JUnit) that run all day.
Basic Questions

• When do V&V start? When are they done?
• Which techniques should be applied?
• How do we know a product is ready?
• How can we control the quality of successive releases?
• How can we improve development?

Collecting Data

Pareto’s Law

Approximately 80% of defects come from 20% of modules

To improve development, one needs to capture data from projects and aggregate it to improve development. (The data shown here shows the occurrence of vulnerabilities in Mozilla Firefox.)

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

61

62

63
Basic Questions

• When do V&V start? When are they done?
• Which techniques should be applied?
• How do we know a product is ready?
• How can we control the quality of successive releases?
• How can we improve development?

Strategic Issues

• Specify requirements in a quantifiable manner
• State testing objectives explicitly
• Understand the users of the software and develop a profile for each user category
• Develop a testing plan that emphasizes “rapid cycle testing”

Strategic Issues

• Build “robust” software that is designed to test itself
• Use effective formal technical reviews as a filter prior to testing
• Conduct formal technical reviews to assess the test strategy and test cases themselves
• Develop a continuous improvement approach for the testing process
Design for Testing

- **OO design principles** also improve testing
  Encapsulation leads to good unit tests
- **Provide diagnostic methods**
  Primarily used for debugging, but may also be useful as regular methods
- **Assertions** are great helpers for testing
  Test cases may be derived automatically

---

Summary

- **Validation and Verification**
  - Validation includes verifying user feedback
  - Verification includes testing, requirements, and inputs

- **V&V Activities**
  - Various activities such as design, test, integration, and evaluation

---

Trade-Offs

- We can be accurate
  - (optimistic or pessimistic)
  - or we can simpify
  - properties...
  - but not all