Testing Strategies

Software Engineering
Andreas Zeller • Saarland University

1

From Pressman, “Software Engineering – a practitioner’s approach”, Chapter 13 and Pezze + Young, “Software Testing and Analysis”, Chapters 1–4

2

Quality

3

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

Testing

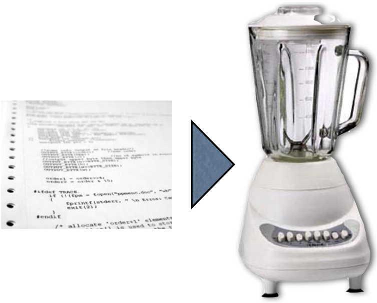
- *Testing:* a procedure intended to establish the quality, performance, or reliability of something, esp. before it is taken into widespread use.

From Oxford dictionary

4

Software Testing

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

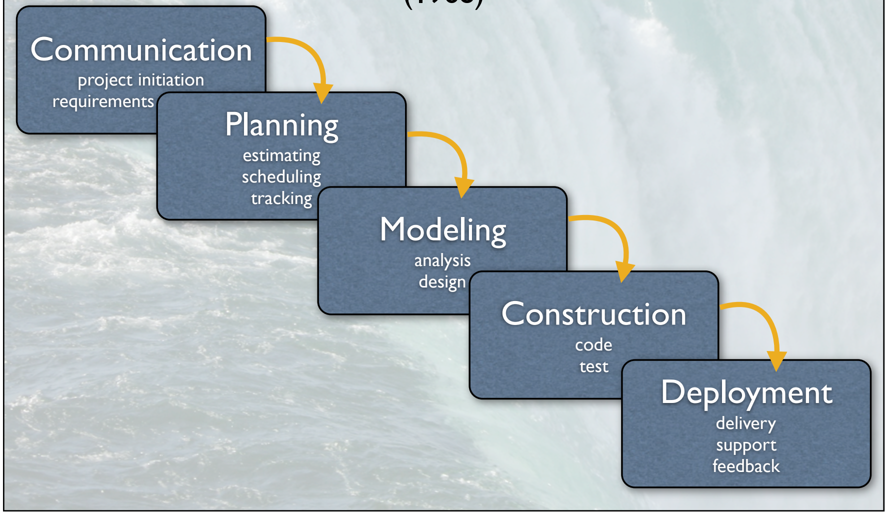


From Pressman, “Software Engineering – a practitioner’s approach”, Chapter 13

5

Waterfall Model

(1968)



Communication
project initiation
requirements

Planning
estimating
scheduling
tracking

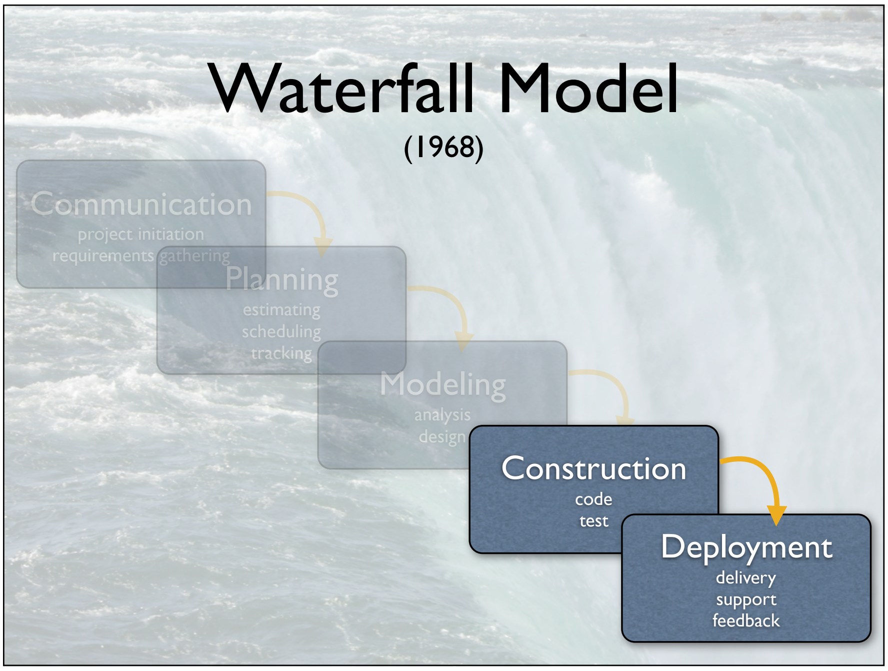
Modeling
analysis
design

Construction
code
test

Deployment
delivery
support
feedback

Let’s recall the Waterfall model.

6



Waterfall Model

(1968)

Communication
project initiation
requirements gathering

Planning
estimating
scheduling
tracking

Modeling
analysis
design

Construction
code
test

Deployment
delivery
support
feedback

7

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.

We built it!

8

So, we simply assume our code is done –

Shall we deploy it?

9

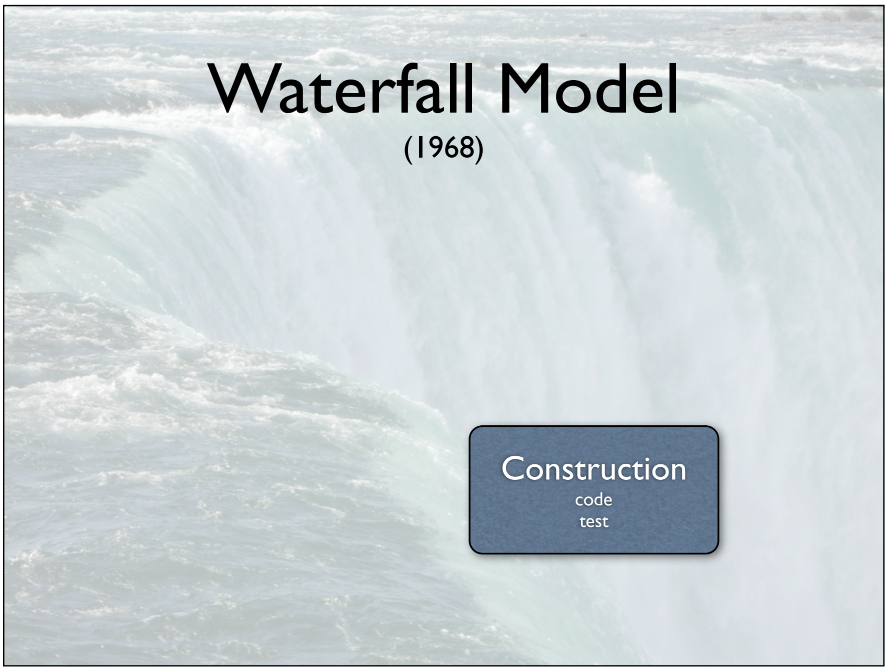
– but is it ready for release?

10

It’s not like this is the ultimate horror…

11

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



Waterfall Model

(1968)

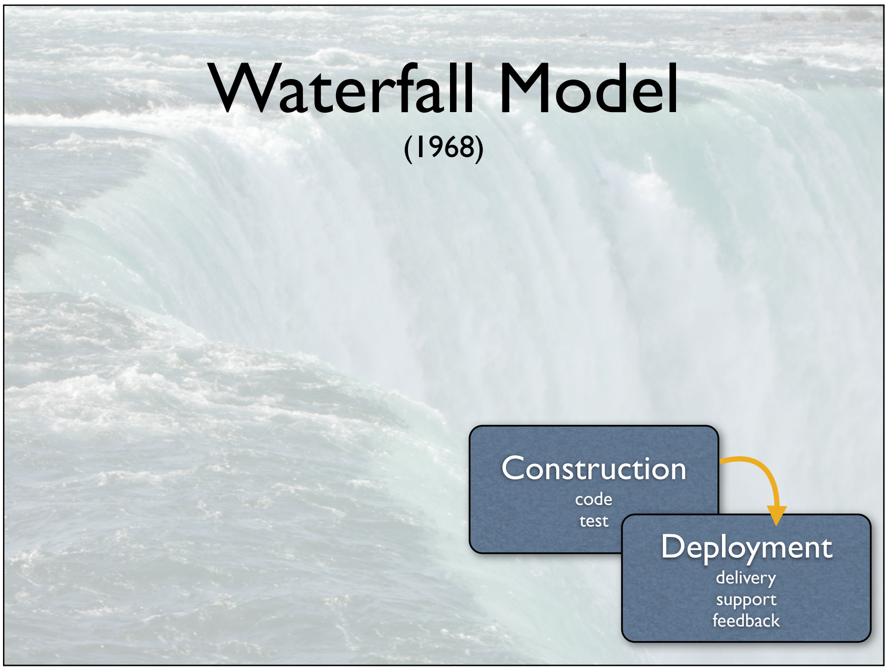
Construction

code

test

12

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



Waterfall Model

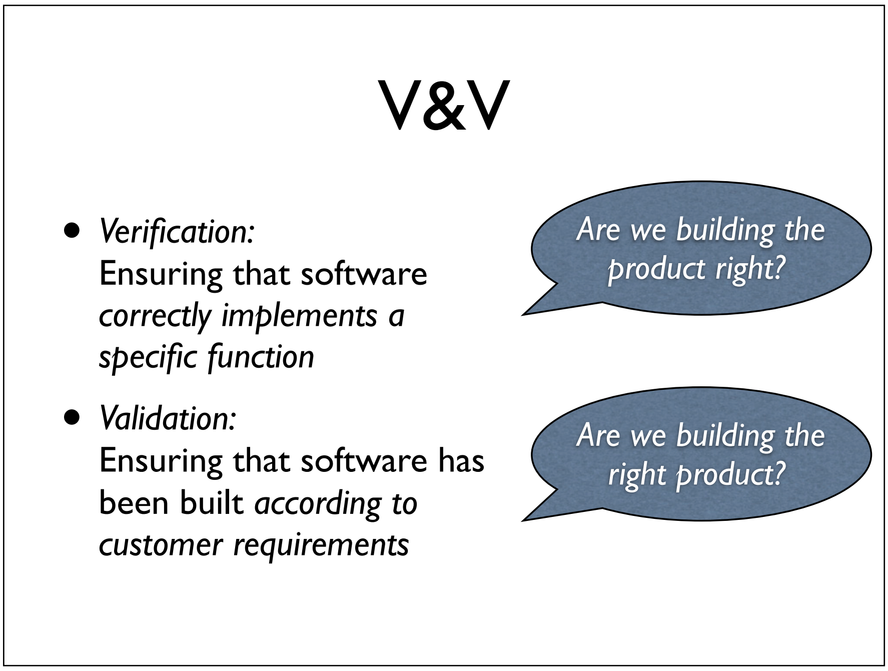
(1968)

Construction
code
test

Deployment
delivery
support
feedback

13

and the question is: how to make your code ready for deployment.



V&V

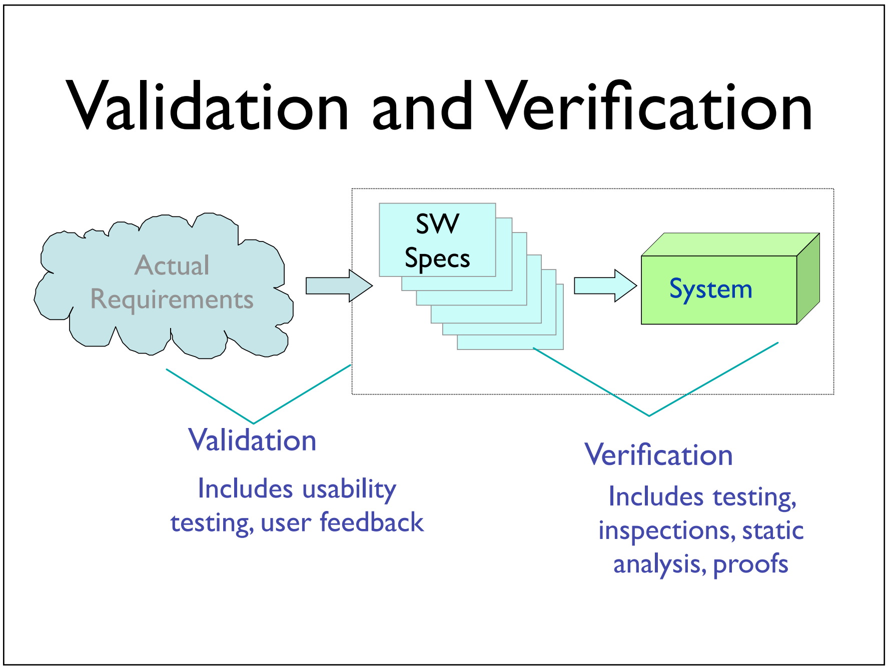
- *Verification:*
  Ensuring that software *correctly implements a specific function*
- *Validation:*
  Ensuring that software has been built *according to customer requirements*

Are we building the product right?

Are we building the right product?

14

These activities are summarized as V&V – verification and validation
See Pressman, ch. 13: “Testing Strategies”



Validation and Verification

Actual Requirements

SW Specs

System

Validation

Includes usability testing, user feedback

Verification

Includes testing, inspections, static analysis, proofs

15

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

16

Verification

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

this one is verifiable.

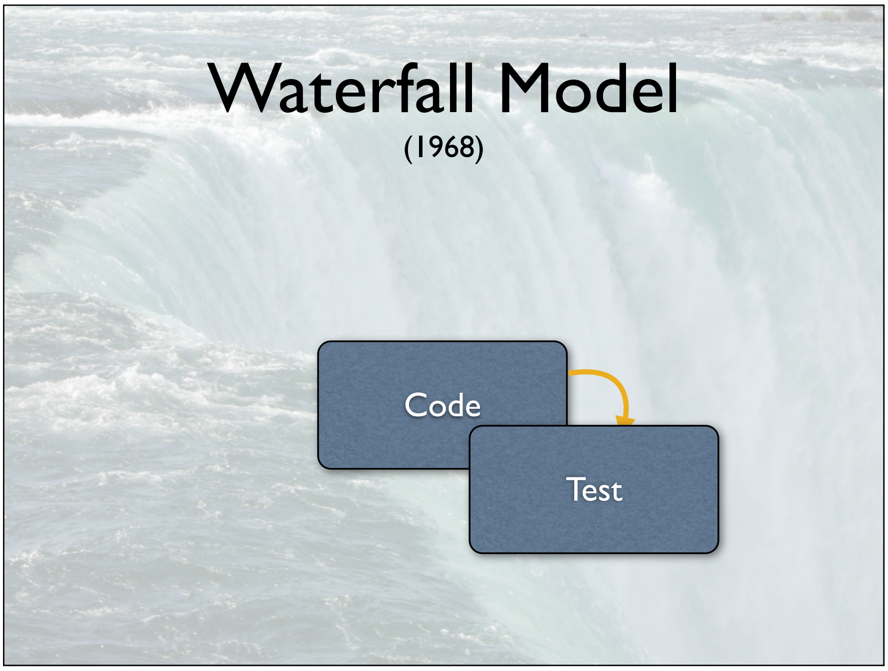
17

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?

When do V&V start? When are they done?

18

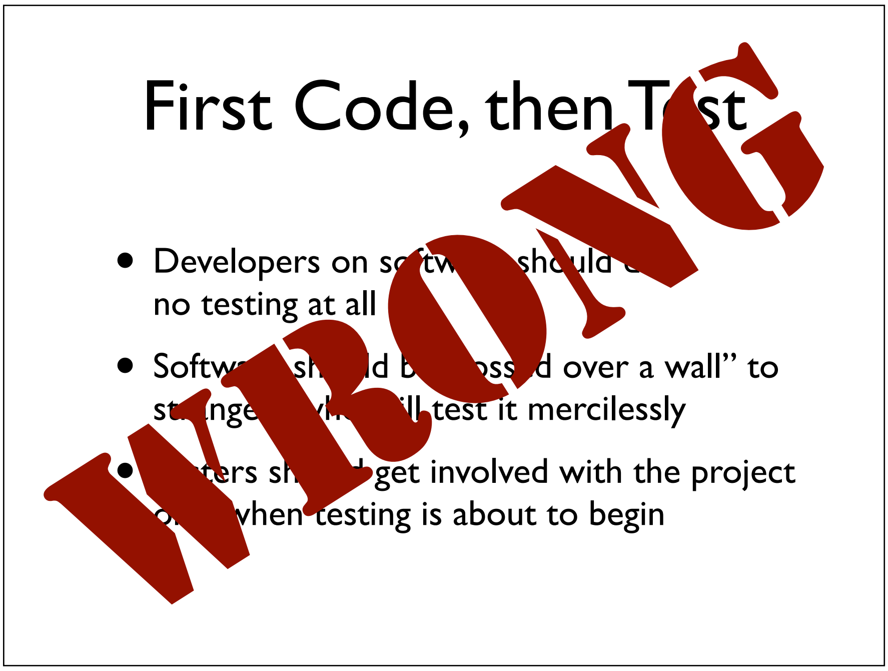


Waterfall Model

(1968)

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

19

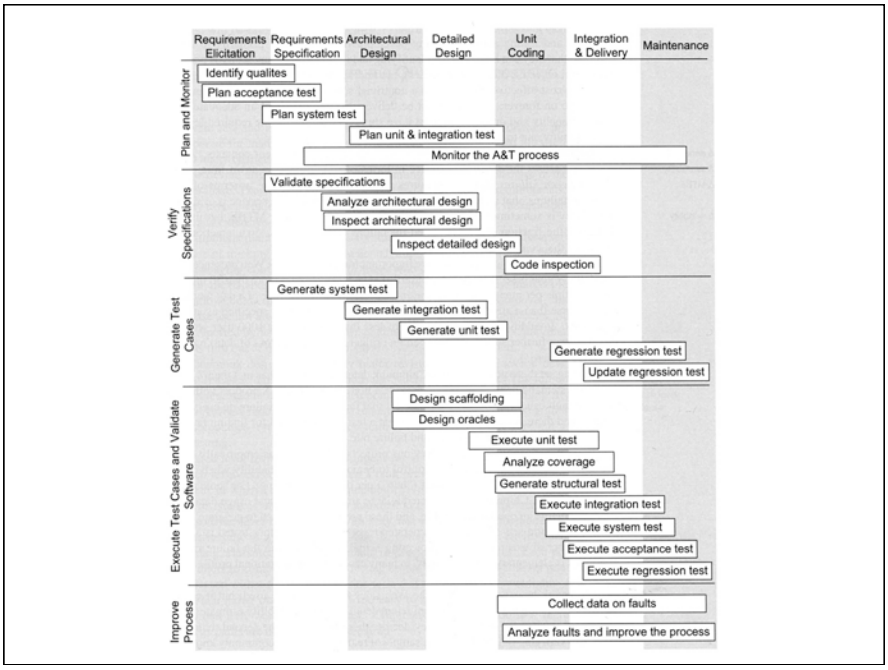


First Code, then Test

- 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

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

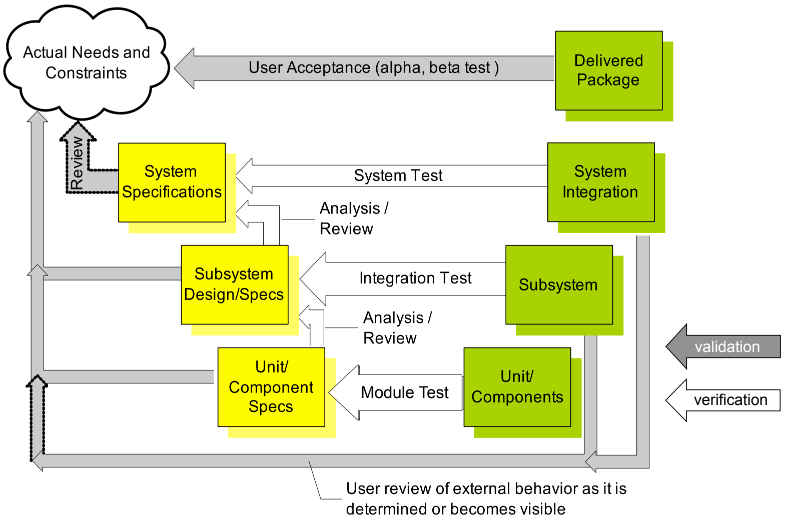
20



Verification and validation activities occur all over the software process (from Pezze + Young, "Software Testing and Analysis")

21

V&V Activities



Actual Needs and Constraints

User Acceptance (alpha, beta test)

Delivered Package

Review

System Specifications

System Test

System Integration

Analysis / Review

Subsystem Design/Specs

Integration Test

Subsystem

Analysis / Review

validation

Unit/ Component Specs

Module Test

Unit/ Components

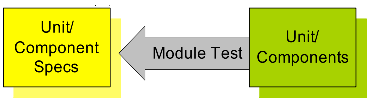
verification

User review of external behavior as it is determined or becomes visible

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)



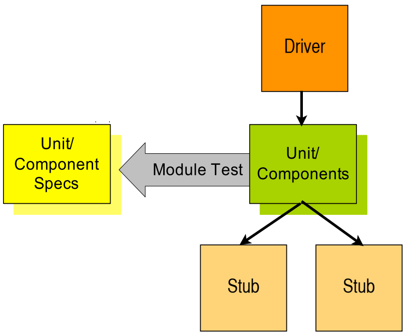
Unit/ Component Specs

Module Test

Unit/ Components

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

Stubs and Drivers



Driver

Unit/ Component Specs

Module Test

Unit/ Components

Stub

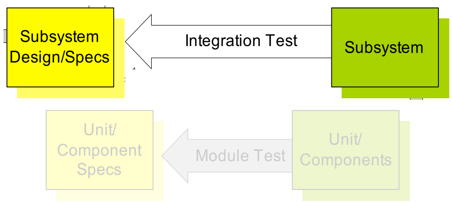
Stub

- 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*



Subsystem Design/Specs

Integration Test

Subsystem

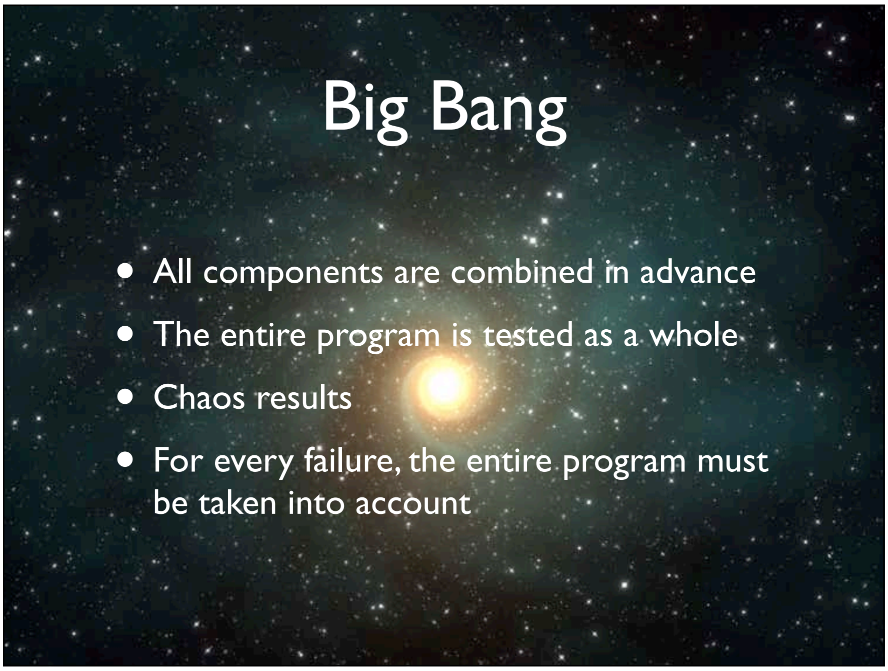
Unit/ Component Specs

Module Test

Unit/ Components

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

25



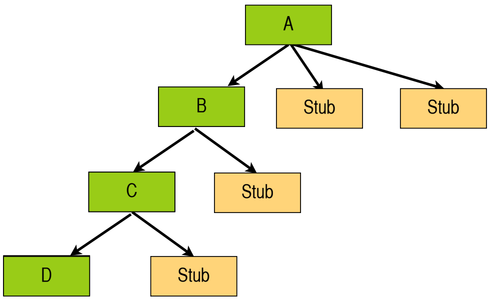
Big Bang

- 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

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

26

Top-Down Integration

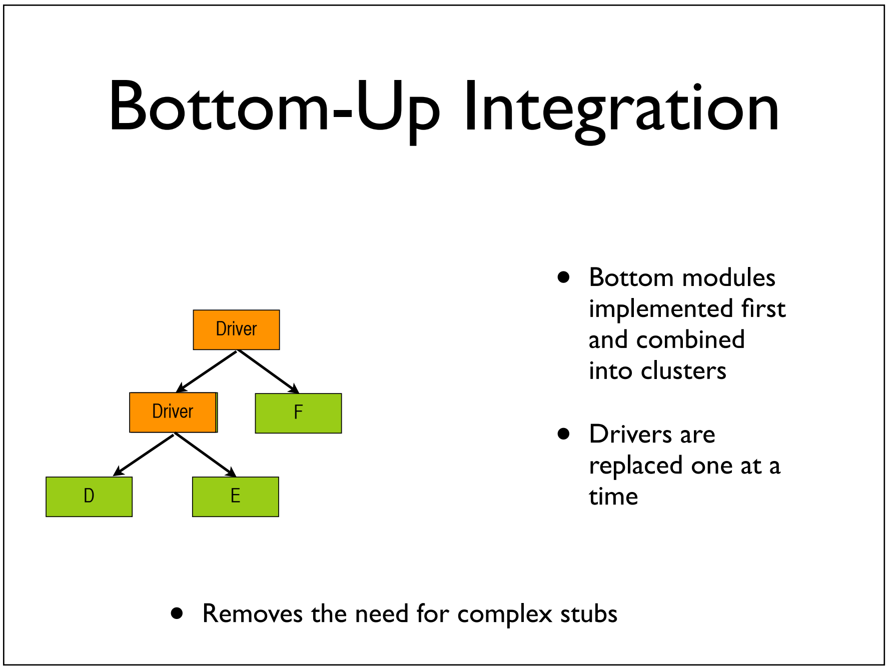


- Top module is tested with stubs (and then used as driver)
- Stubs are replaced one at a time (“depth first”)
- As new modules are integrated, tests are re-run

- Allows for early demonstration of capability

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

27



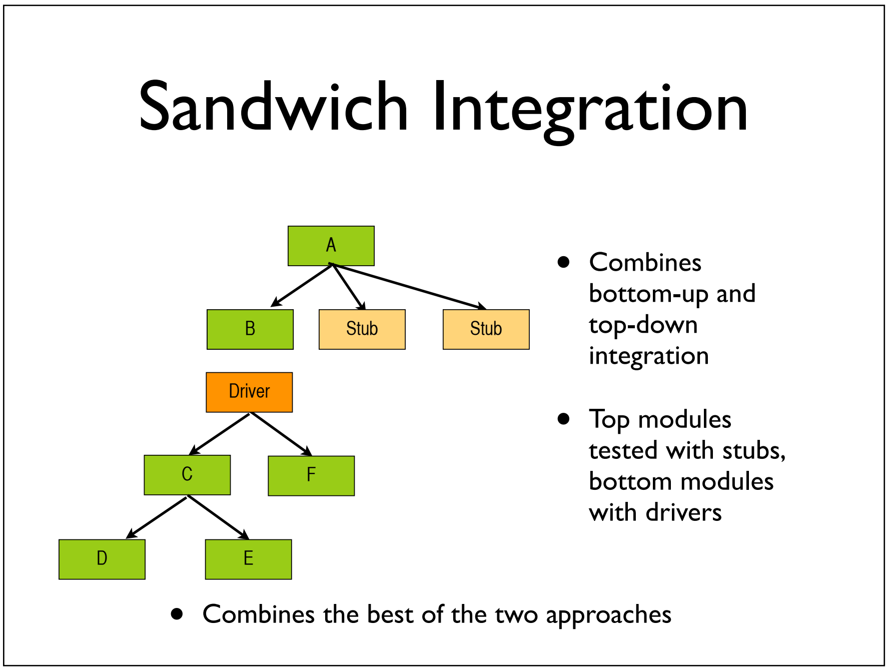
Bottom-Up Integration

- Bottom modules implemented first and combined into clusters
- Drivers are replaced one at a time

- Removes the need for complex stubs

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

28



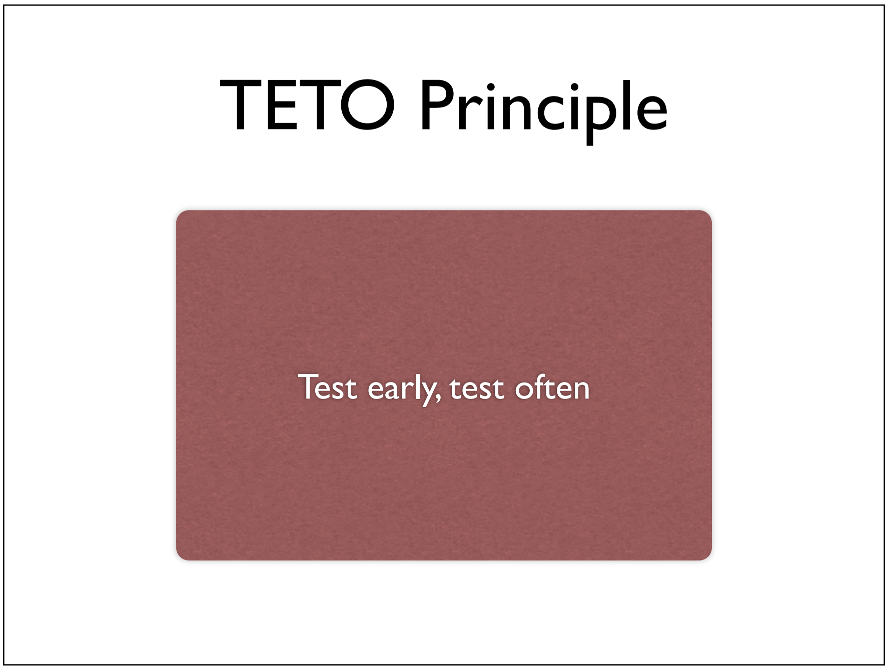
Sandwich Integration

- Combines bottom-up and top-down integration
- Top modules tested with stubs, bottom modules with drivers

- Combines the best of the two approaches

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

29



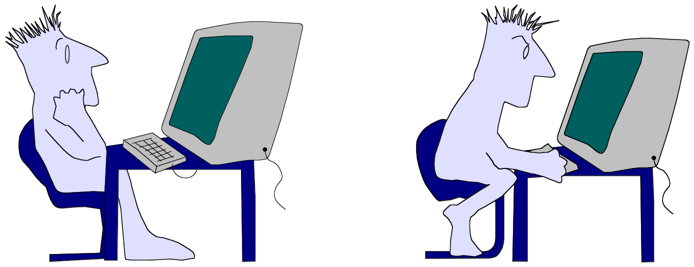
TETO Principle

Test early, test often

Evidence: pragmatic – there is no way a test can ever cover all possible paths through a program

30

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*

31

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

The Ideal Tester

32

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

The Developer

33

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

The Developers

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

34

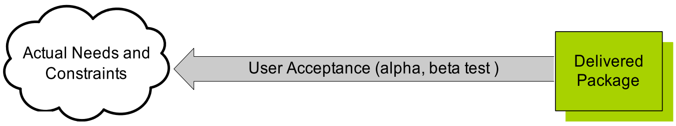
Weinberg’s Law

A developer is unsuited to test his or her code.

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”

35

Acceptance Testing


Actual Needs and Constraints
User Acceptance (alpha, beta test)
Delivered Package

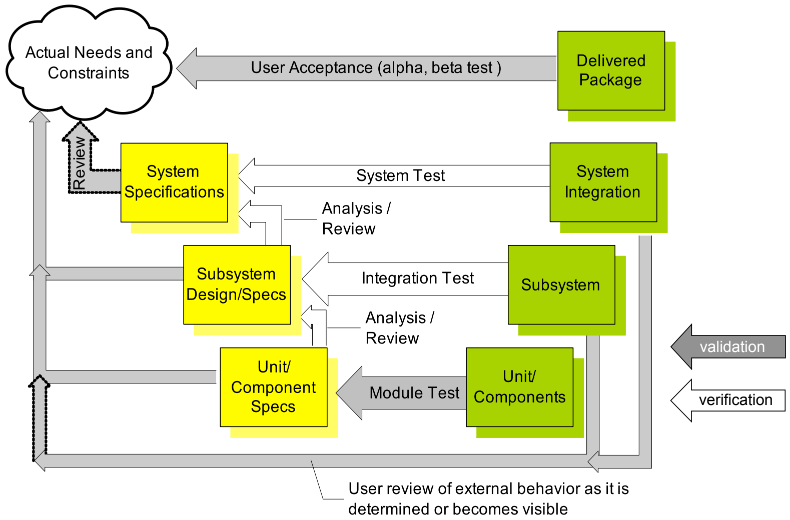
- Acceptance testing checks whether the *contractual requirements* are met
- Typically incremental
  (*alpha test* at production site, *beta test* at user’s site)
- Work is over when acceptance testing is done

36

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



Actual Needs and Constraints

User Acceptance (alpha, beta test)

Delivered Package

Review

System Specifications

System Test

System Integration

Analysis / Review

Subsystem Design/Specs

Integration Test

Subsystem

Analysis / Review

Unit/ Component Specs

Module Test

Unit/ Components

validation

verification

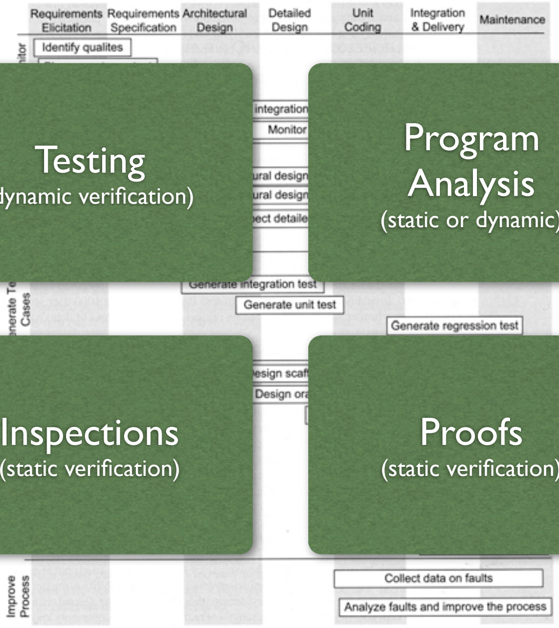
User review of external behavior as it is determined or becomes visible

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?

Which techniques should be applied?



Testing
(dynamic verification)

Program Analysis
(static or dynamic)

Inspections
(static verification)

Proofs
(static verification)

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.

40

Why V&V is hard
(on software)

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

41

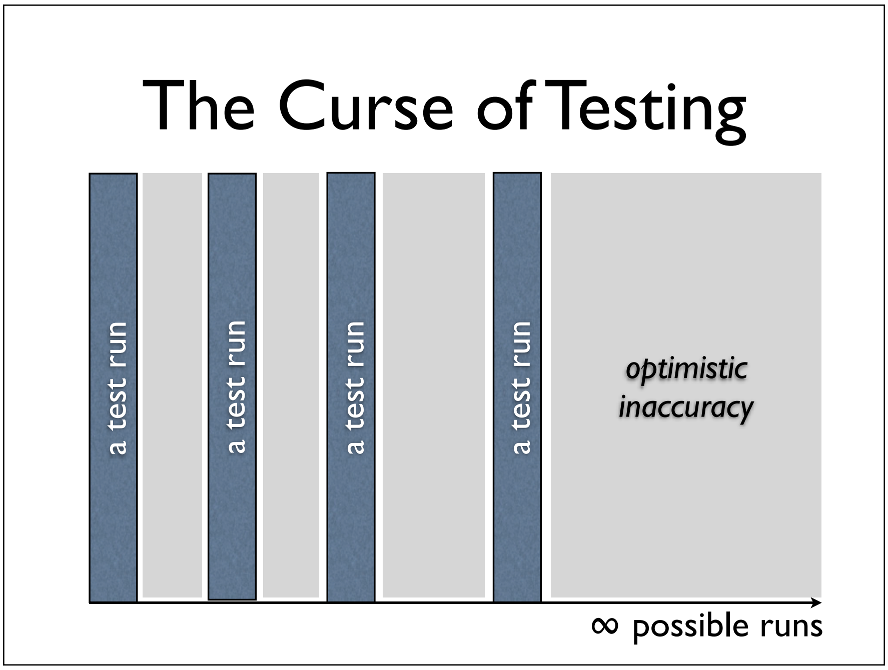
Compare

can load 1,000 kg

can sort 256 elements

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")

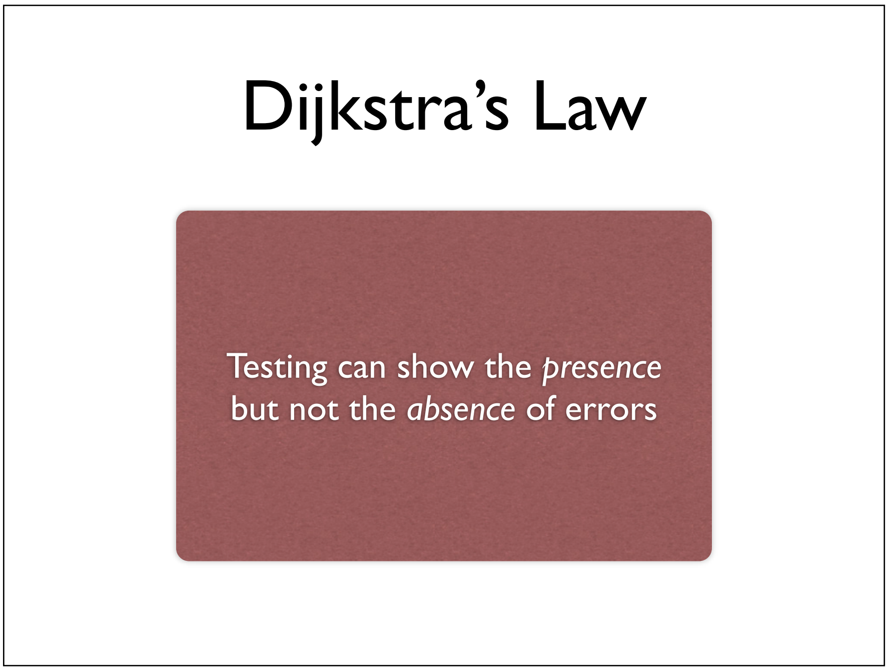
42



The Curse of Testing

Every test can only cover a single run

43

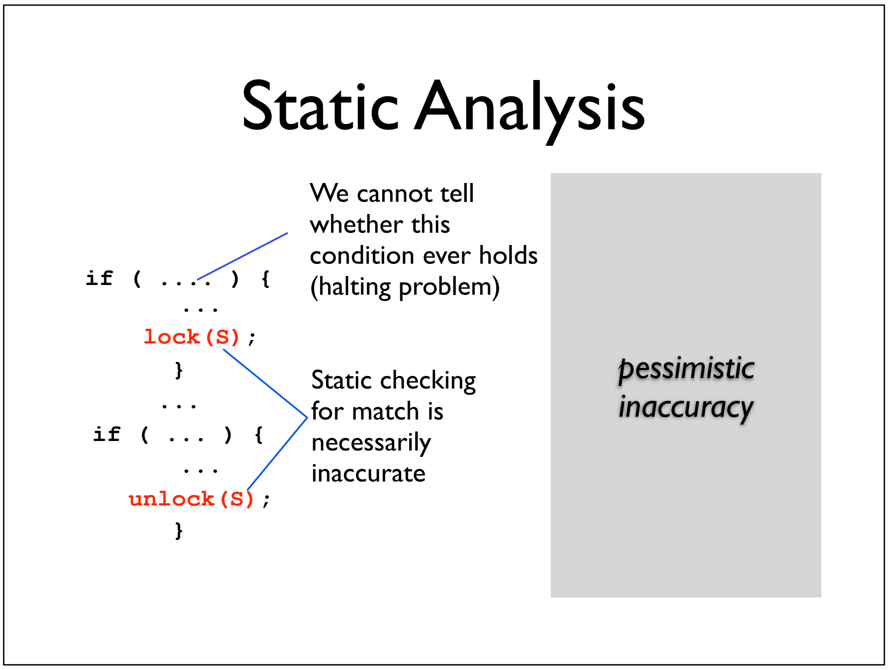


Dijkstra’s Law

Testing can show the *presence* but not the *absence* of errors

Evidence: pragmatic – there is no way a test can ever cover all possible paths through a program

44



Static Analysis

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

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

Static checking for match is necessarily inaccurate

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”)

45

Pessimistic Inaccuracy

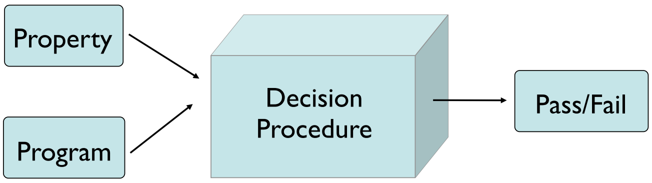
```
static void questionable() {
    int k;

    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?

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”)

You can’t ~~always~~ ever get what you want



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

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

Simplified Properties

original problem

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

Static checking for match is necessarily inaccurate

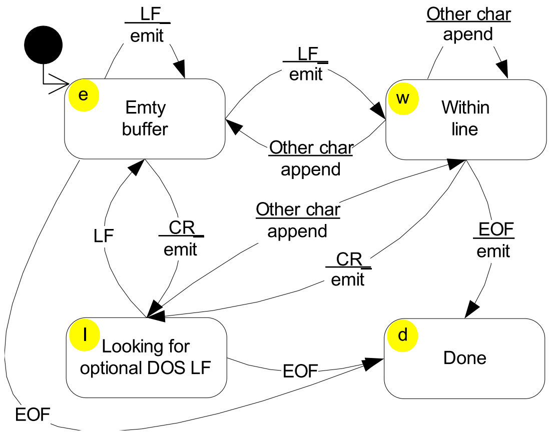
simplified property

Java prescribes a more restrictive, but statically checkable construct.

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

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

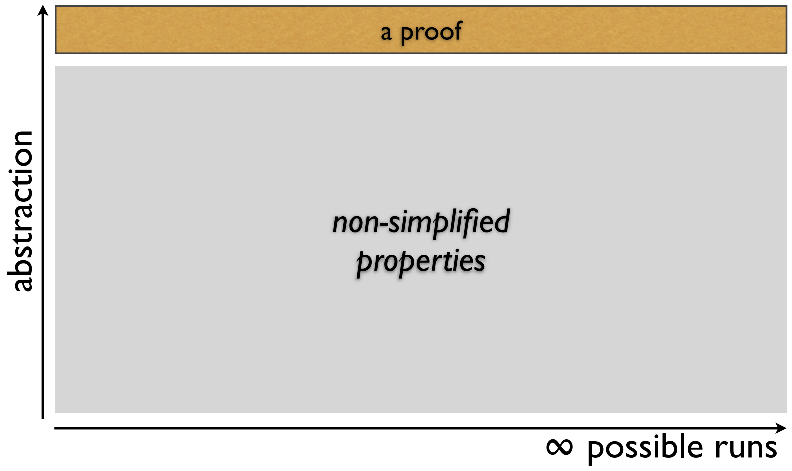
Simplified Properties



If you can turn your program into a finite state machine, for instance, you can prove all sorts of properties (from Pezze + Young, “Software Testing and Analysis”)

49

Static Verification

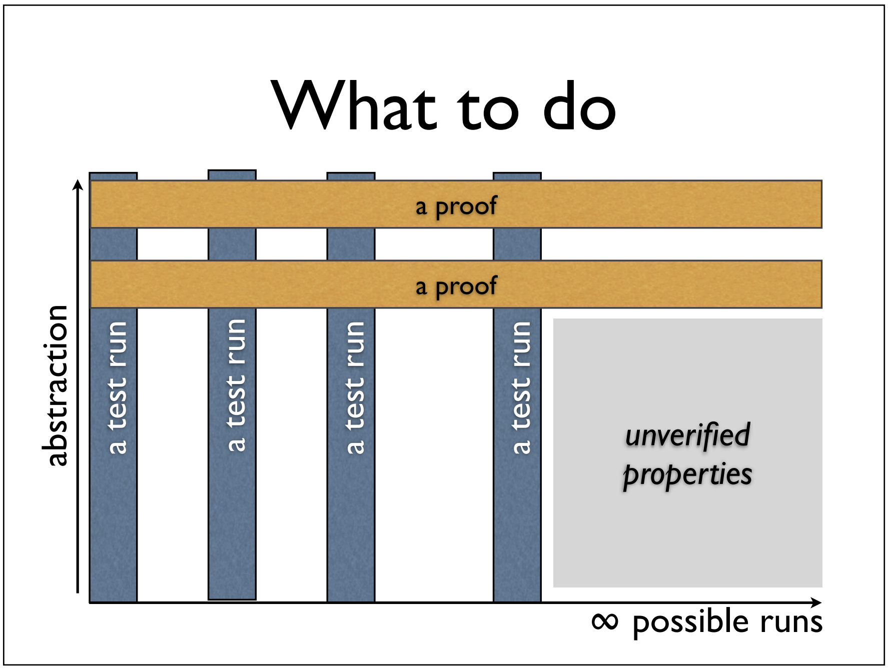


A proof can cover all runs – but only at a higher abstraction level

50

In some way, fear, uncertainty and doubt will thus prevail…

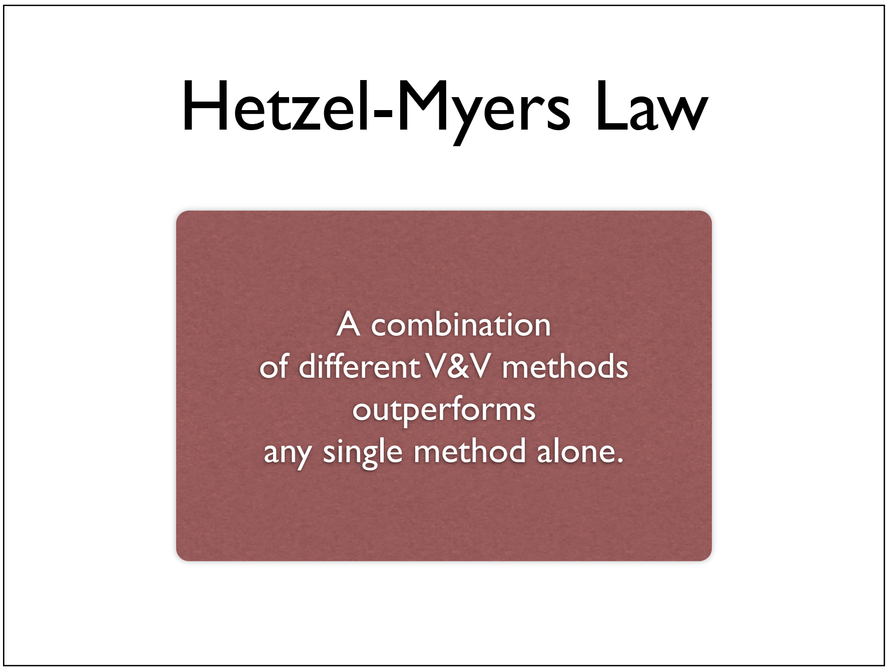
51



What to do

…but we can of course attempt to cover as many runs – and abstractions – as possible!

52

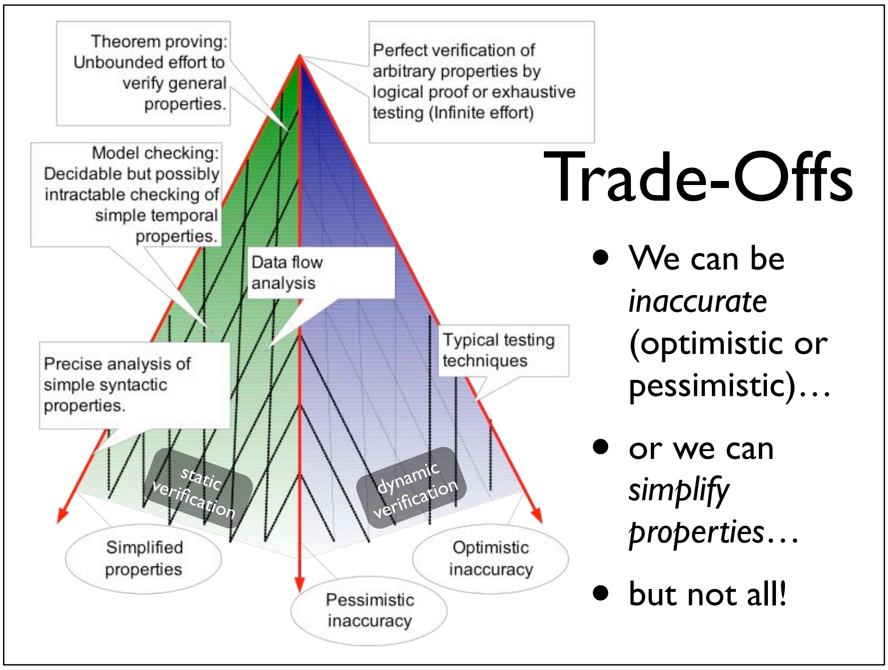


Hetzel-Myers Law

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, different "aspects".

53



Trade-Offs

Theorem proving: Unbounded effort to verify general properties.
Perfect verification of arbitrary properties by logical proof or exhaustive testing (Infinite effort)
Model checking: Decidable but possibly intractable checking of simple temporal properties.
Data flow analysis
Typical testing techniques
Precise analysis of simple syntactic properties.
static verification
dynamic verification
Simplified properties
Optimistic inaccuracy
Pessimistic inaccuracy

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

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

54

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?

How do we know a product is ready?

Readiness in Practice

Let the customer test it :-)

Readiness in Practice

We're out of time.

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 ≥ 0.995.

This is the type of argument we aim for. From Pressman, “Software Engineering – a practitioner’s approach”, Chapter 13

58

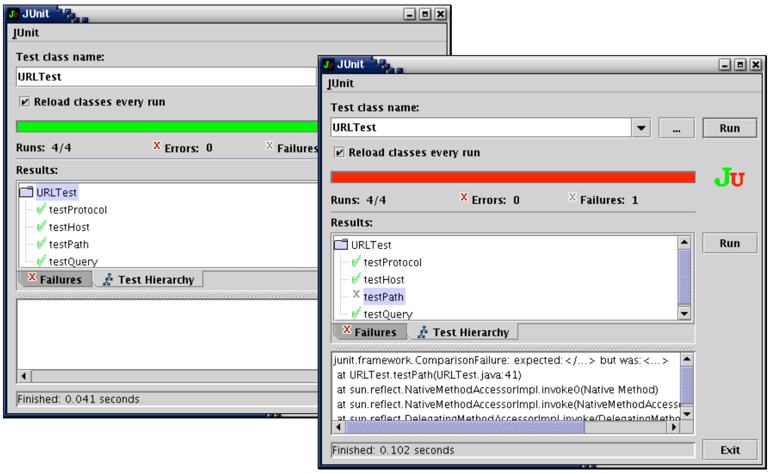
Basic Questions

- 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?

How can we control the quality of successive releases?

59

Regression Tests



Test class name: URLTest

☑ Reload classes every run

Runs: 4/4 Errors: 0 Failures

URLTest
- testProtocol
- testHost
- testPath
- testQuery

Finished: 0.041 seconds

Test class name: URLTest

☑ Reload classes every run

Runs: 4/4 Errors: 0 Failures: 1

URLTest
- testProtocol
- testHost
- testPath
- testQuery

```
junit.framework.ComparisonFailure: expected:</...> but was:<...>
 at URLTest.testPath(URLTest.java:41)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccess
 at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMetho
```

Finished: 0.102 seconds

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

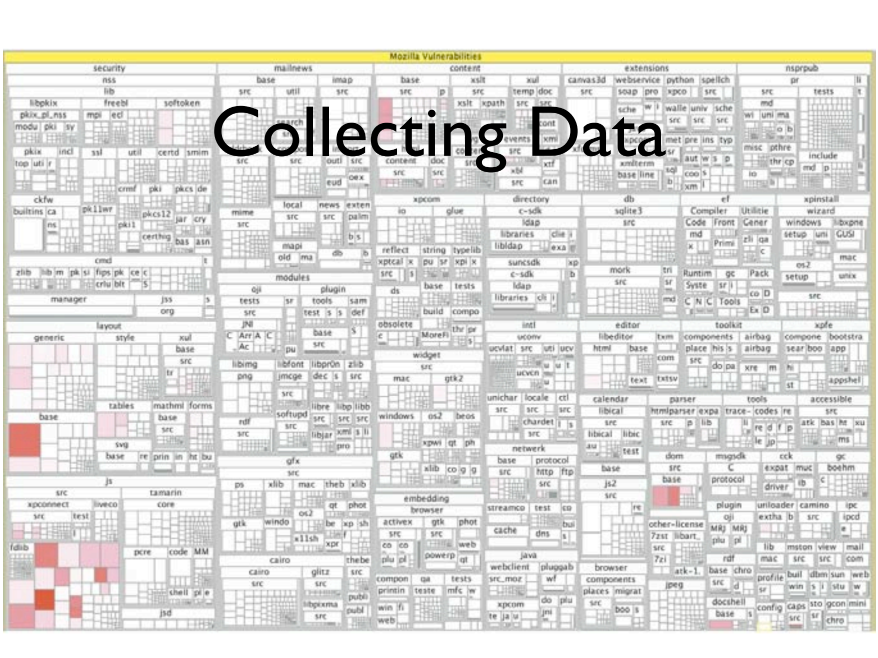
60

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?

How can we improve development?

61



Collecting Data

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.)

62

Pareto’s Law

Approximately 80% of defects come from 20% of modules

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

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?

64

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"

From Pressman, "Software Engineering – a practitioner's approach", Chapter 13

65

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

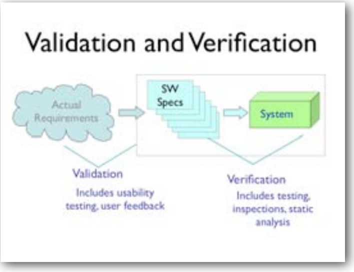
From Pressman, "Software Engineering – a practitioner's approach", Chapter 13

66

Design for Testing

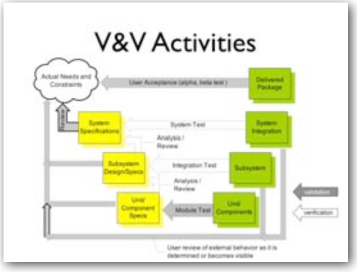
- *OO design principles* also improve testing
  Encapsulation leads to good unit tests
- Provide *diagnostic methods*
  Primarly used for debugging, but may also be useful as regular methods
- *Assertions* are great helpers for testing
  Test cases may be derived automatically

Summary

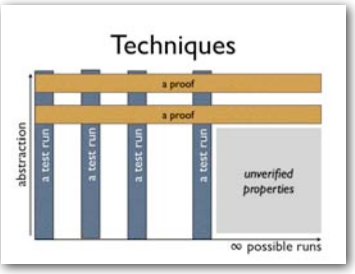


Validation and Verification

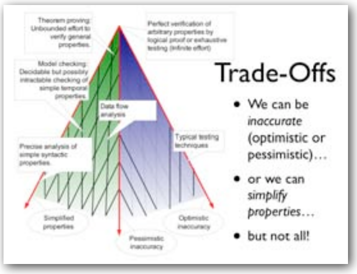
Actual Requirements
SW Specs
System
Validation
Includes usability testing, user feedback
Verification
Includes testing, inspections, static analysis



V&V Activities



Techniques



Theorem proving: Unbounded effort to verify general properties.
Perfect verification of arbitrary properties by logical proof or exhaustive testing (Infinite effort)
Model checking: Decidable but possibly intractable checking of simple temporal properties.
Data flow analysis
Typical testing techniques
Precise analysis of simple syntactic properties.
Simplified properties
Optimistic inaccuracy
Pessimistic inaccuracy

Trade-Offs

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