Software Engineering Testing

Introduction to Software Testing

Objective

- View the “big picture'' of software quality in the context of a software development project and organization
- Introduce the range of software verification and validation activities
- Software Testing Terminology

Questions

1. Why do we test ?

2. What should we do during testing ?

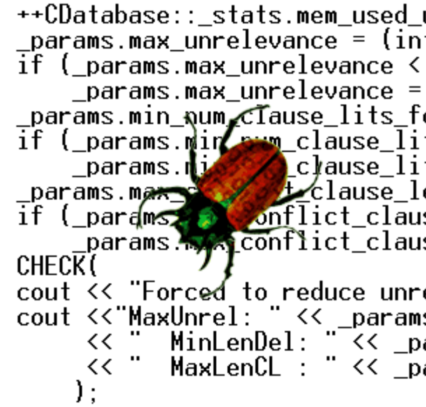
3. How do we get to this future of testing ?

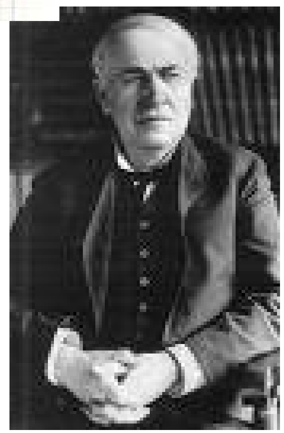
History and Motivation

A few spectacular software failures

The First Bugs

Hopper's "bug" (moth stuck in a relay on an early machine)





"an analyzing process must equally have been performed in order to furnish the Analytical Engine with the necessary operative data; and that herein may also lie a possible source of error. Granted that the actual mechanism is unerring in its processes, the cards may give it wrong orders. " – Ada, Countess Lovelace (notes on Babbage's Analytical Engine)

"It has been just so in all of my inventions. The first step is an intuition, and comes with a burst, then difficulties arise—this thing gives out and *[it is]* then that 'Bugs'—as such little faults and difficulties are called—show themselves and months of intense watching, study and labor are requisite. . ." – Thomas Edison

Failures in Production Software

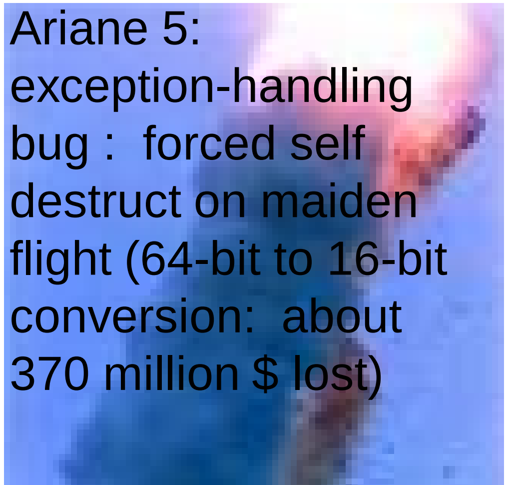
- NASA's Mars lander, September 1999, crashed due to a units integration fault—over $50 m

- Huge losses due to web application fail
  - Financial services : $6.5 million per hour
  - Credit card sales applications : $2.4 million per hour
- In Dec 2006, *amazon.com's* BOGO offer turned into a double discount
- 2007 : Symantec says that most security vulnerabilities are due to faulty software
- Stronger testing could solve most of these problems

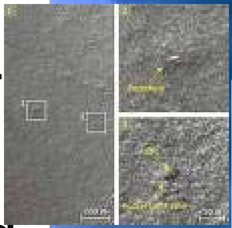
World-wide monetary loss due to poor software is staggering

Why Does Testing Matter?

- NIST report, “The Economic Impacts of Inadequate Infrastructure for Software Testing” (2002)
  - Inadequate software testing costs the US alone between $22 and $59 billion annually
  - Better approaches could cut this amount in half
- Major failures: Ariane 5 explosion, Mars Polar Lander, Intel’s Pentium FDIV bug
- Insufficient testing of safety-critical softwar can cost *lives*:
  - THERAC-25 radiation machine: 3 dead

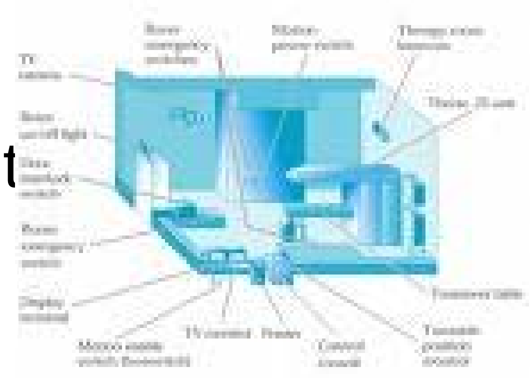


Ariane 5: exception-handling bug : forced self destruct on maiden flight (64-bit to 16-bit conversion: about 370 million $ lost)



Mars Polar Lander crash site?

- We want our programs to be reliab
  - Testing is how, in most cases, we find out if they are



THERAC-25 design

Airbus 319 Safety Critical Software Control

Loss of autopilot

Loss of most flight deck lighting and intercom

Loss of both the commander's and the co-pilot's primary flight and navigation displays

Northeast Blackout of 2003

508 generating units and 256 power plants shut down

Affected 10 million people in Ontario, Canada

Affected 40 million people in 8 US states

Financial losses of $6 Billion USD

The alarm system in the energy management system failed due to a software error and operators were not informed of the power overload in the system

What Does This Mean?

Software testing is getting more important

Testing in the 21st Century

- We are going through a time of change
- Software defines behavior
  - network routers, finance, switching networks, other infrastructure
- Today's software market :
  - is much bigger
  - is more competitive
  - has more users

Industry is going through a revolution in what testing means to the success of software products

- Agile processes put increased pressure on testers

Testing in the 21st Century

- More safety critical, real-time software
- Enterprise applications means bigger programs, more users
- Embedded software is ubiquitous … check your pockets
- Paradoxically, free software increases our expectations !
- Security is now all about software faults
  - Secure software is reliable software
- The web offers a new deployment platform
  - Very competitive and very available to more users
  - Web apps are distributed
  - Web apps must be highly reliable

Industry desperately needs our inventions !

Mismatch in Needs and Goals

- Industry wants testing to be simple and easy
- Testing needs to be done more rigorously
- Agile processes put lots of demands on testing
  - Programmers must unit test – with no training, education or tools!
  - Tests are key components of functional requirements – but who builds those tests ?

Bottom line—lots of crappy software

In real life…

A stack of computer printouts—and no documentation

Cost of Testing

You're going to spend at least half of your development budget on testing, whether you want to or not

- In the real-world, testing is the principle post-design activity
- Restricting early testing usually increases cost
- Extensive hardware-software integration requires more testing

Why Test?

If you don't know why you're conducting a test, it won't be very helpful

- Written test objectives and requirements are rare
- What are your planned coverage levels?
- How much testing is enough?
- Common objective – spend the budget …

Why Test?

If you don’t start planning for each test when the functional requirements are formed, you’ll never know why you’re conducting the test

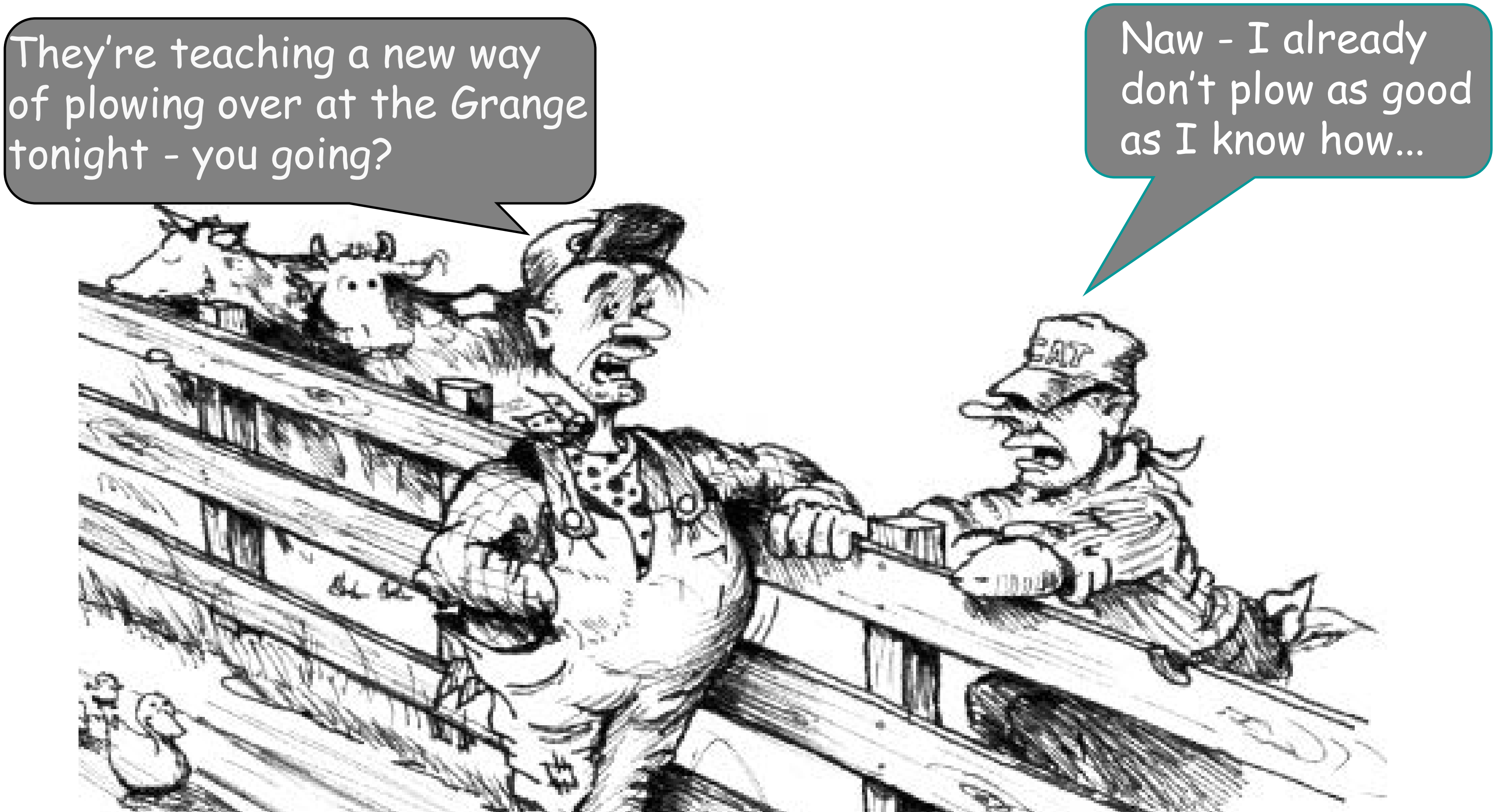
- 1980: “The software shall be easily maintainable”
- Threshold reliability requirements?
- What fact is each test trying to verify?
- Requirements definition teams should include testers!

Cost of NOT Testing

Program Managers often say: "Testing is too expensive."

- Not testing is even more expensive
- Planning for testing after development is prohibitively expensive
- A test station for circuit boards costs half a million dollars …
- Software test tools cost less than $10,000 !!!

Caveat: Impact of New Tools and Techniques



“Knowing is not enough, we must apply. Willing is not enough, **we must do**.” Goethe

<u>What</u> should we do?

1. Types of test activities
2. Software testing terms
3. Changing notions of testing
   – test coverage criteria
   – criteria based on structures

Testing in the 21st Century

- We are going through a time of change
- Software Defines Behavior
  - network routers
  - financial networks
  - telephone switching networks
  - other infrastructure

Testing ideas have matured enough to be used in practice

- Embedded Control Applications
  - airplanes, air traffic control
  - spaceships
  - watches
  - ovens
  - remote controllers
  - PDAs
  - memory seats
  - DVD players
  - garage door openers
  - cell phones
- Safety critical, real-time software
- Web apps must be highly reliable
- And of course … security is now all about software faults !

Types of Test Activities

- Testing can be broken up into four general types of activities
  1. Test Design → 1.a) Criteria-based, 1.b) Human-based
  2. Test Automation
  3. Test Execution
  4. Test Evaluation
- Each type of activity requires different skills, background knowledge, education and training
- No reasonable software development organization uses the same people for requirements, design, implementation, integration and configuration control

Why do test organizations still use the same people for all four test activities?? <u>waste</u> of resources?

1. Test Design – (a) Criteria-Based

Design test values to satisfy coverage criteria or other engineering goal

- This is the most technical job in software testing
- Requires knowledge of :
  - Discrete math
  - Programming
  - Testing
- Requires much of a traditional CS degree
- This is intellectually stimulating, rewarding, and challenging
- Test design is analogous to software architecture on the development side
- Using people who are not qualified to design tests is a sure way to get ineffective tests

1. Test Design – (b) Human-Based

Design test values based on domain knowledge of the program and human knowledge of testing

- This is much harder than it may seem to developers
- Criteria-based approaches can be blind to special situations
- Requires knowledge of :
  - – Domain, testing, and user interfaces
- Requires almost no traditional CS
  - – A background in the domain of the software is essential
  - – An empirical background is very helpful (biology, psychology, …)
  - – A logic background is very helpful (law, philosophy, math, …)
- This is intellectually stimulating, rewarding, and challenging
  - – But not to typical CS majors – they want to solve problems and build things

2. Test Automation

Embed test values into executable scripts

- This is slightly less technical
- Requires knowledge of programming
  - Fairly straightforward programming – small pieces and simple algorithms
- Requires very little theory
- Programming is out of reach for many domain experts
- Who is responsible for determining and embedding the expected outputs (this is a challenge!)?
  - Test designers may not always know the expected outputs
  - Test evaluators need to get involved early to help with this

3. Test Execution

Run tests on the software and record the results

- This is easy – and trivial if the tests are well automated
- Requires basic computer skills
  - Interns
  - Employees with no technical background
- Asking qualified test designers to execute tests is a sure way to convince them to look for a development job
- If, for example, GUI tests are not well automated, this requires a lot of manual labor
- Test executors have to be very careful and meticulous with bookkeeping

4. Test Evaluation

Evaluate results of testing, report to developers

- This is much harder than it may seem
- Requires knowledge of :
  - Domain
  - Testing
  - User interfaces and psychology
- Usually requires almost no traditional CS
  - A background in the domain of the software is essential
  - An empirical background is very helpful (biology, psychology, …)
  - A logic background is very helpful (law, philosophy, math, …)
- This is intellectually stimulating, rewarding, and challenging
  - But not to typical CS majors – they want to solve problems and build things

Other Activities

- Test management : Sets policy, organizes team, interfaces with development, chooses criteria, decides how much automation is needed, …
- Test maintenance : Tests must be saved for reuse as software evolves
  - Requires cooperation of test designers and automators
  - Deciding when to trim the test suite is partly policy and partly technical – and in general, very hard !
  - Tests should be put in configuration control
- Test documentation : All parties participate
  - Each test must document "why" – criterion and test requirement satisfied or a rationale for human-designed tests
  - Traceability throughout the process must be ensured
  - Documentation must be kept in the automated tests

Types of Test Activities – Summary

1a.	Design	Design test values to satisfy engineering goals
	Criteria	Requires knowledge of discrete math, programming and testing
1b.	Design	Design test values from domain knowledge and intuition
	Human	Requires knowledge of domain, UI, testing
2.	Automation	Embed test values into executable scripts
		Requires knowledge of scripting
3.	Execution	Run tests on the software and record the results
		Requires very little knowledge
4.	Evaluation	Evaluate results of testing, report to developers
		Requires domain knowledge

- These four general test activities are quite different
- It is a poor use of resources to use people inappropriately

Most test teams use the same people for ALL FOUR activities !!

To use our people effectively

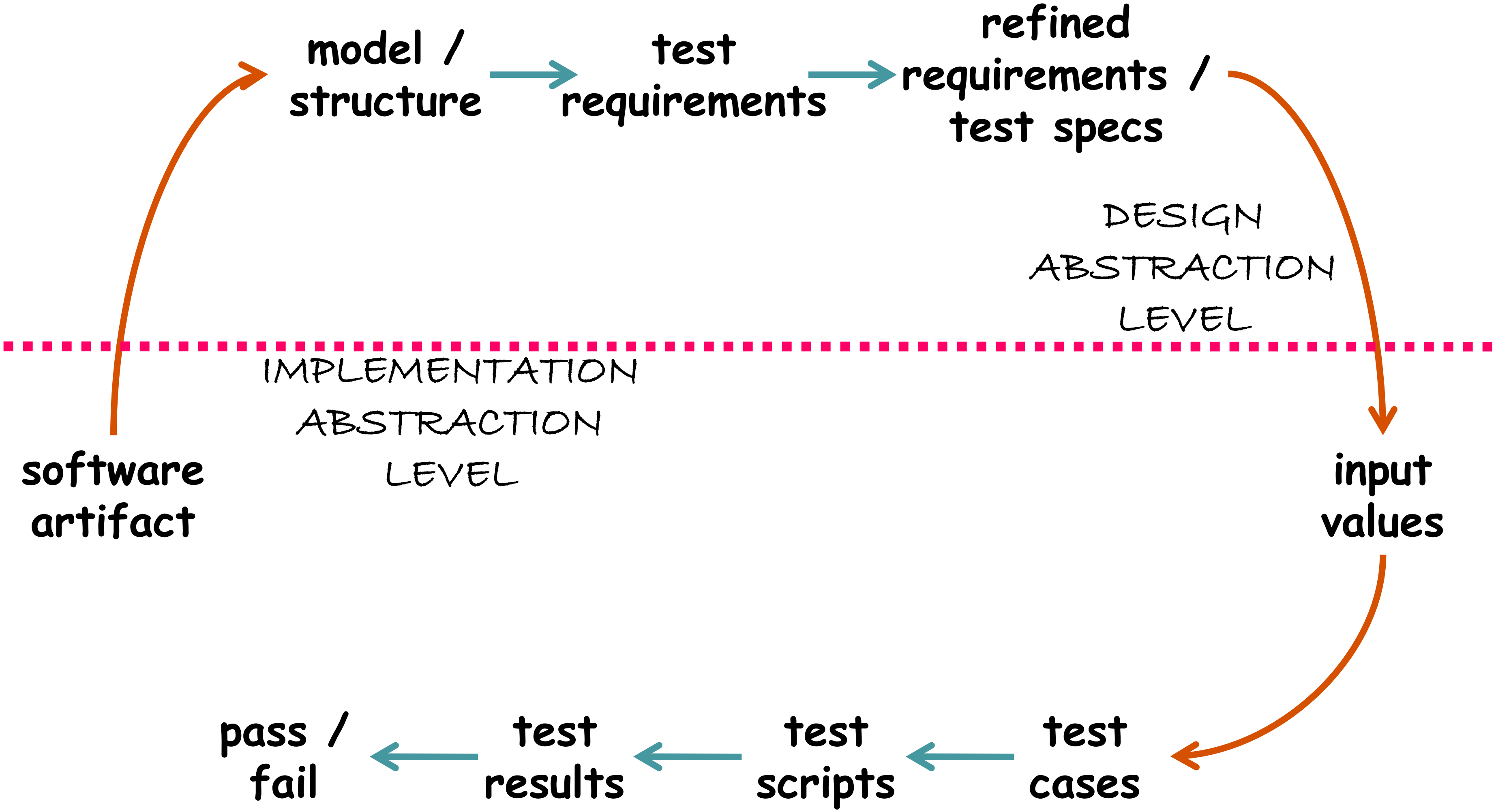
and to test efficiently

we need a process that

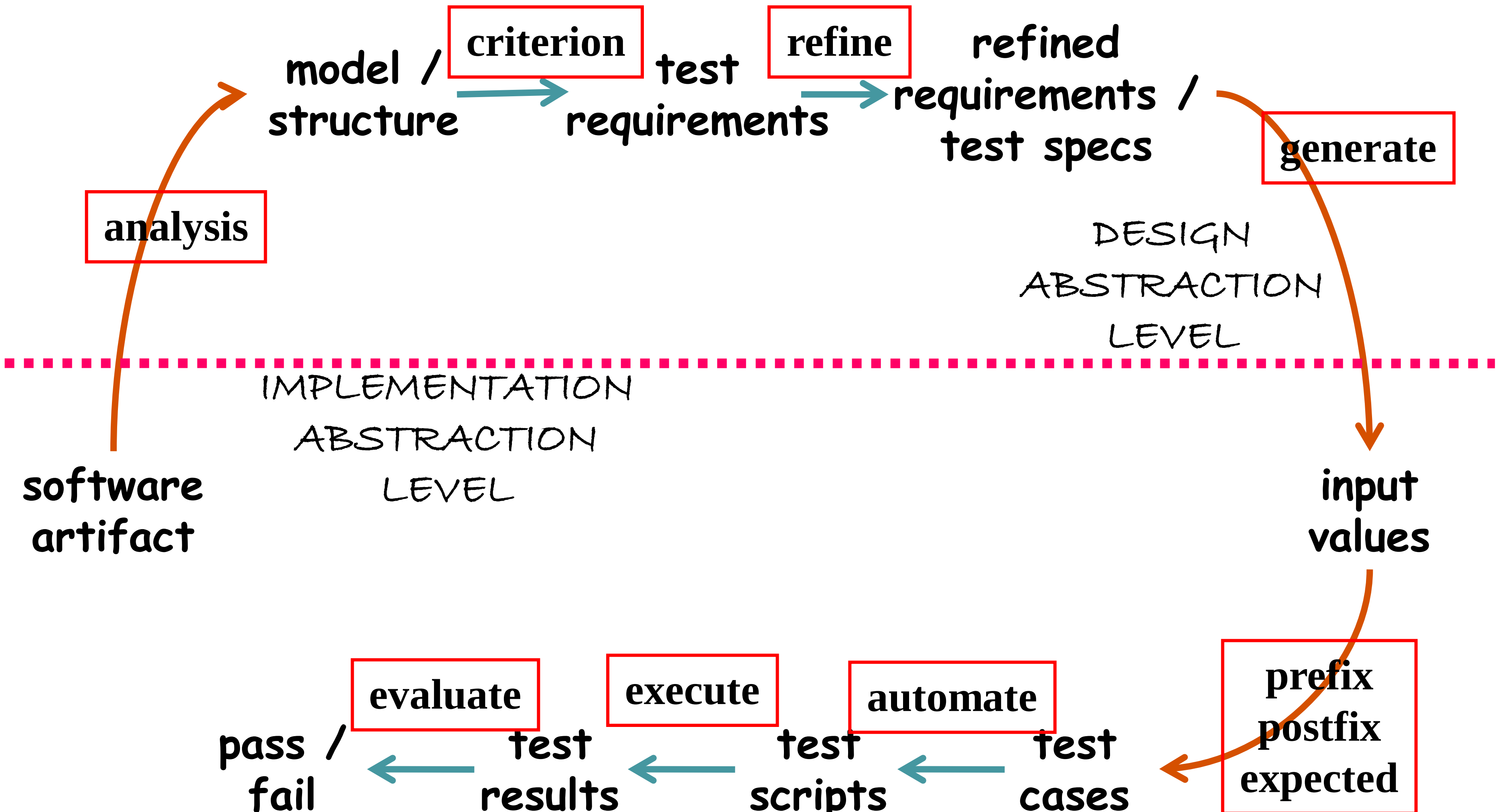
lets test designers

raise their level of abstraction

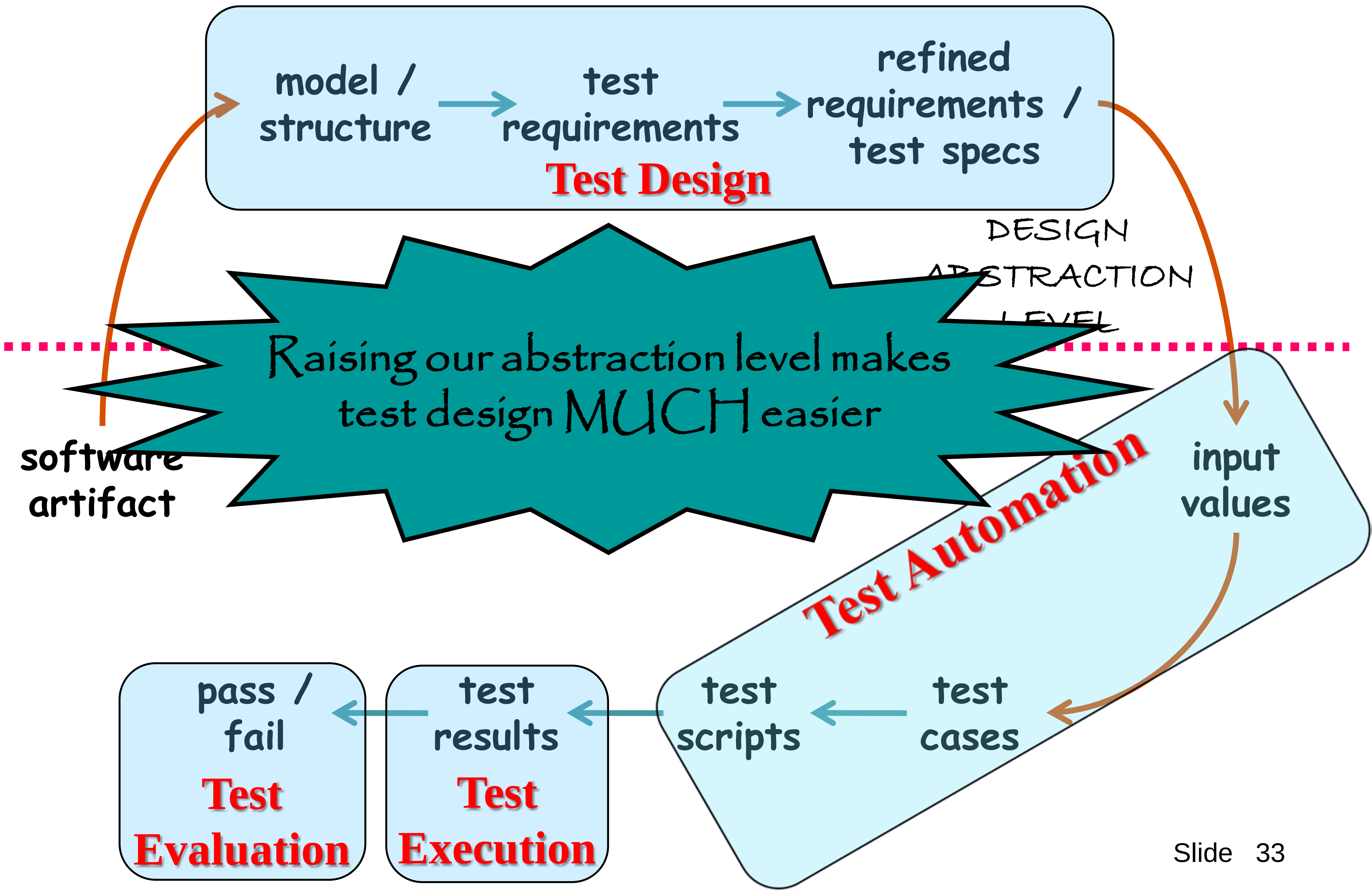
Model-Driven Test Design



Model-Driven Test Design – Steps



Model-Driven Test Design – Activities



Software Testing Terms

- Like any field, software testing comes with a large number of specialized terms that have particular meanings in this context

- Some of the following terms are standardized, some are used consistently throughout the literature and the industry, but some vary by author (of the software), topic, or test organization

- The definitions presented here are intended to be the most commonly used

Terminology (V&V – by IEEE)

- Validation : The process of evaluating software at the end of software development to ensure compliance with intended usage

- Verification : The process of determining whether the products of a given phase of the software development process fulfill the requirements established during the previous phase

- IV&V stands for "*independent verification and validation*"

More on V&V (I)

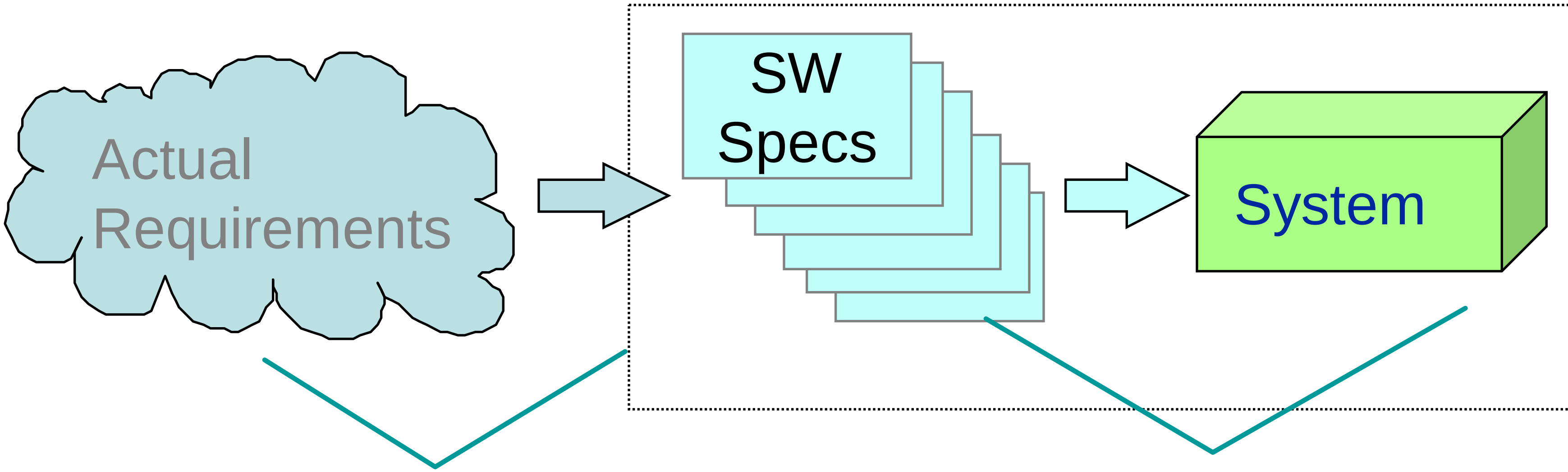
- Validation:
  does the software system meets the user's real needs?

  are we building the right software?

- Verification:
  does the software system meets the requirements specifications?

  are we building the software right?

More on V&V (II)



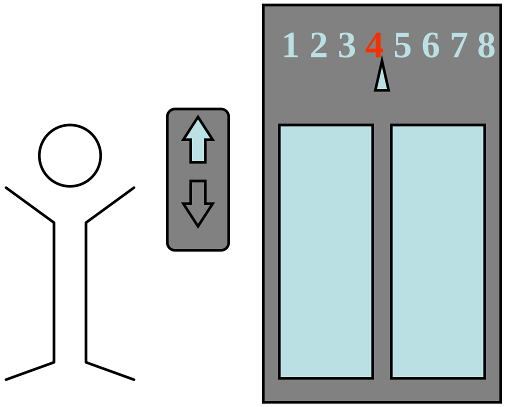
Validation

Includes usability testing, user feedback

Verification

Includes testing, inspections, static analysis

V&V depends on the specification



Example: elevator response

Unverifiable (but validatable) spec: ... if a user presses a request button at floor i, an available elevator must arrive at floor i soon...

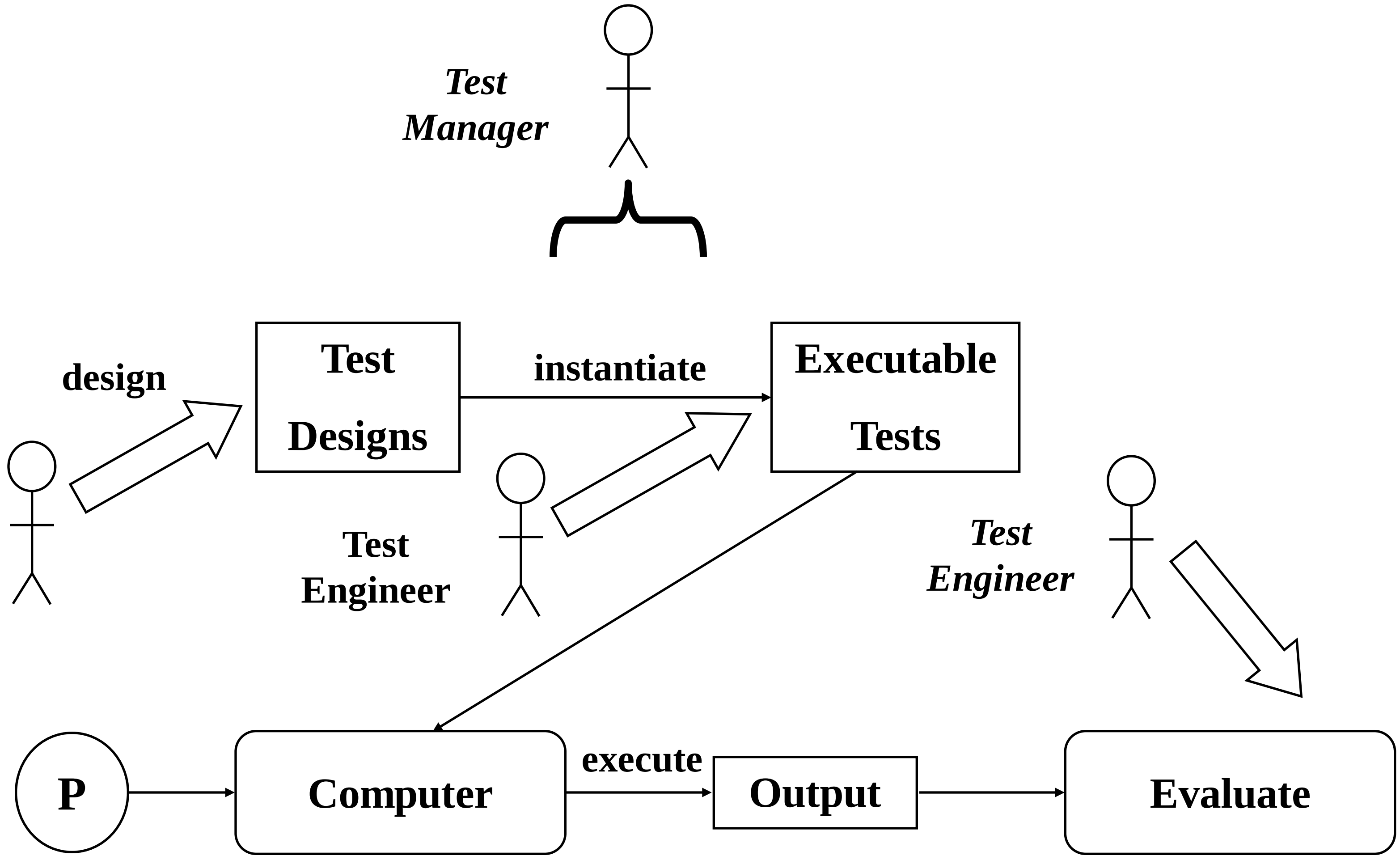
Verifiable spec: ... if a user presses a request button at floor i, an available elevator must arrive at floor i within 30 seconds...

Test Engineer & Test Managers

- Test Engineer : An IT professional who is in charge of one or more technical test activities
  - designing test inputs
  - producing test values
  - running test scripts
  - analyzing results
  - reporting results to developers and managers

- Test Manager : In charge of one or more test engineers
  - sets test policies and processes
  - interacts with other managers on the project
  - otherwise helps the engineers do their work

Test Engineer Activities



Static and Dynamic Testing

- Static Testing : Testing without executing the program
  - This include software inspections and some forms of analyses
  - Very effective at finding certain kinds of problems – especially “potential” faults, that is, problems that could lead to faults when the program is modified
- Dynamic Testing : Testing by executing the program with real inputs

Software Faults, Errors & Failures

- Software Fault : A static defect in the software
- Software Error : An incorrect internal state that is the manifestation of some fault
- Software Failure : External, incorrect behavior with respect to the requirements or other description of the expected behavior

Faults in software are design mistakes and will always exist

Testing & Debugging

- Testing : Finding inputs that cause the software to fail

- Debugging : The process of finding a fault given a failure

Fault & Failure Model

Three conditions necessary for a failure to be observed

1. Reachability : The location or locations in the program that contain the fault must be reached
2. Infection : The state of the program must be incorrect
3. Propagation : The infected state must propagate to cause some output of the program to be incorrect

Test Case

- Test Case Values : The values that directly satisfy one test requirement

- Expected Results : The result that will be produced when executing the test if the program satisfies it intended behavior

Observability and Controllability

- Software Observability : How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components
  - Software that affects hardware devices, databases, or remote files have low observability

- Software Controllability : How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors
  - Easy to control software with inputs from keyboards
  - Inputs from hardware sensors or distributed software is harder
  - Data abstraction reduces controllability and observability

Inputs to Affect Controllability and Observability

- Prefix Values : Any inputs necessary to put the software into the appropriate state to receive the test case values

- Postfix Values : Any inputs that need to be sent to the software after the test case values
- Two types of postfix values
  1. Verification Values : Values necessary to see the results of the test case values
  2. Exit Commands : Values needed to terminate the program or otherwise return it to a stable state

- Executable Test Script : A test case that is prepared in a form to be executed automatically on the test software and produce a report

Top-Down and Bottom-Up Testing

- Top-Down Testing : Test the main procedure, then go down through procedures it calls, and so on

- Bottom-Up Testing : Test the leaves in the tree (procedures that make no calls), and move up to the root.
  - Each procedure is not tested until all of its children have been tested

White-box and Black-box Testing

- Black-box testing : Deriving tests from external descriptions of the software, including specifications, requirements, and design

- White-box testing : Deriving tests from the source code internals of the software, specifically including branches, individual conditions, and statements

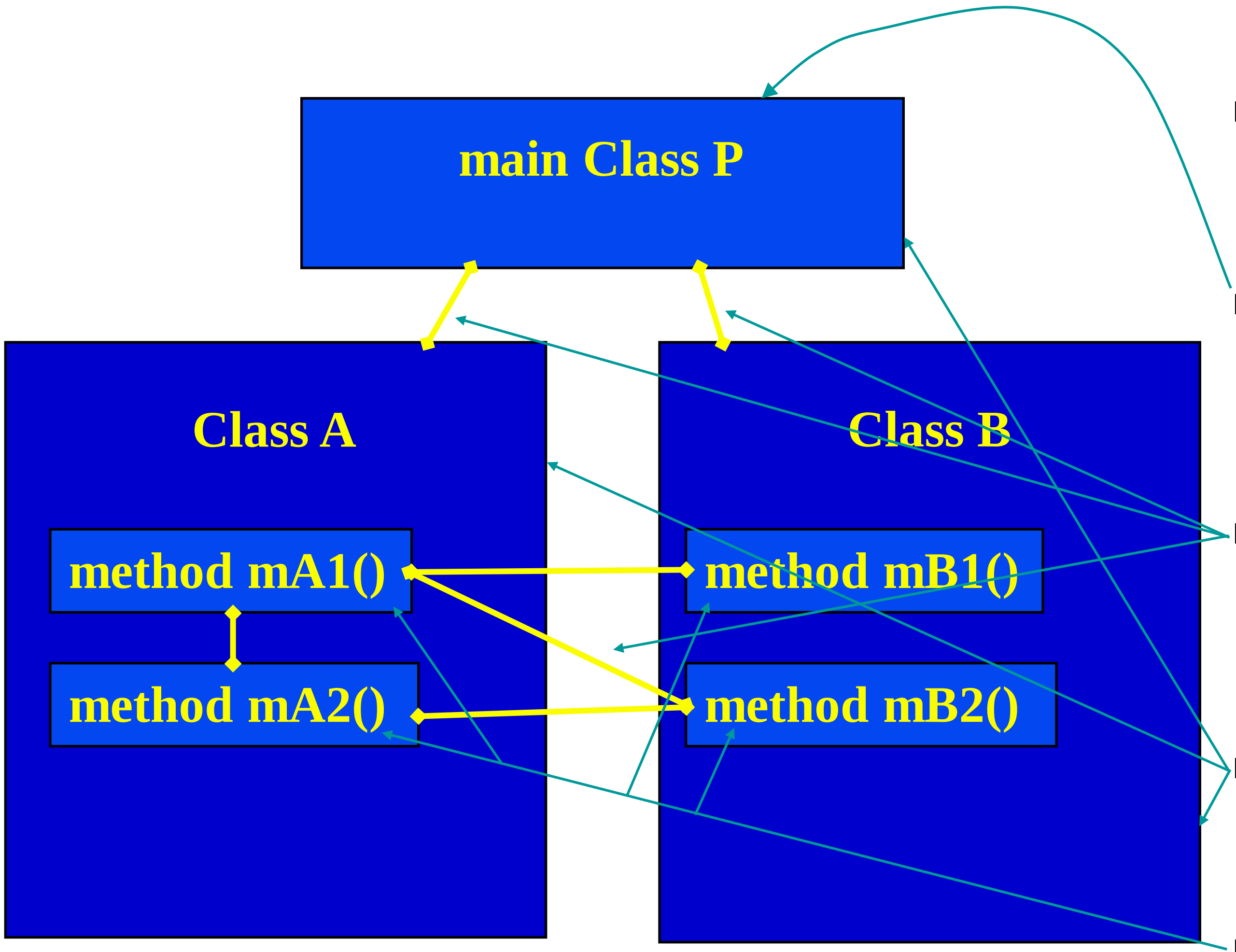
This view is really out of date.

The more general question is: *from what level of abstraction to we derive tests*?

Changing Notions of Testing

- Old view of testing is of testing at specific software development phases
  - Unit, module, integration, system …
- New view is in terms of structures and criteria
  - Graphs, logical expressions, syntax, input space

Old : Testing at Different Levels



- **<u>Acceptance testing</u>: Is the software acceptable to the user?**
- **<u>System testing</u>: Test the overall functionality of the system**
- **<u>Integration testing</u>: Test how modules interact with each other**
- **<u>Module testing</u>: Test each class, file, module or component**
- **<u>Unit testing</u>: Test each unit (method) individually**

Old : Find a Graph and Cover It

- Tailored to:
  - a particular software artifact
    - code, design, specifications
  - a particular phase of the lifecycle
    - requirements, specification, design, implementation
- This viewpoint obscures underlying similarities
- Graphs do not characterize all testing techniques well
- Four abstract models suffice …

New : Test Coverage Criteria

A tester's job is <u>simple</u> : Define a model of the software, then find ways to cover it

- <u>Test Requirements</u> : Specific things that must be satisfied or covered during testing

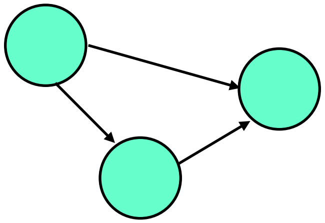
- <u>Test Criterion</u> : A collection of rules and a process that define test requirements

Testing researchers have defined dozens of criteria, but they are all really just a few criteria on four types of structures …

New : Criteria Based on Structures

Structures : Four ways to model software

1. Graphs



2. Logical Expressions

(not X or not Y) and A and B

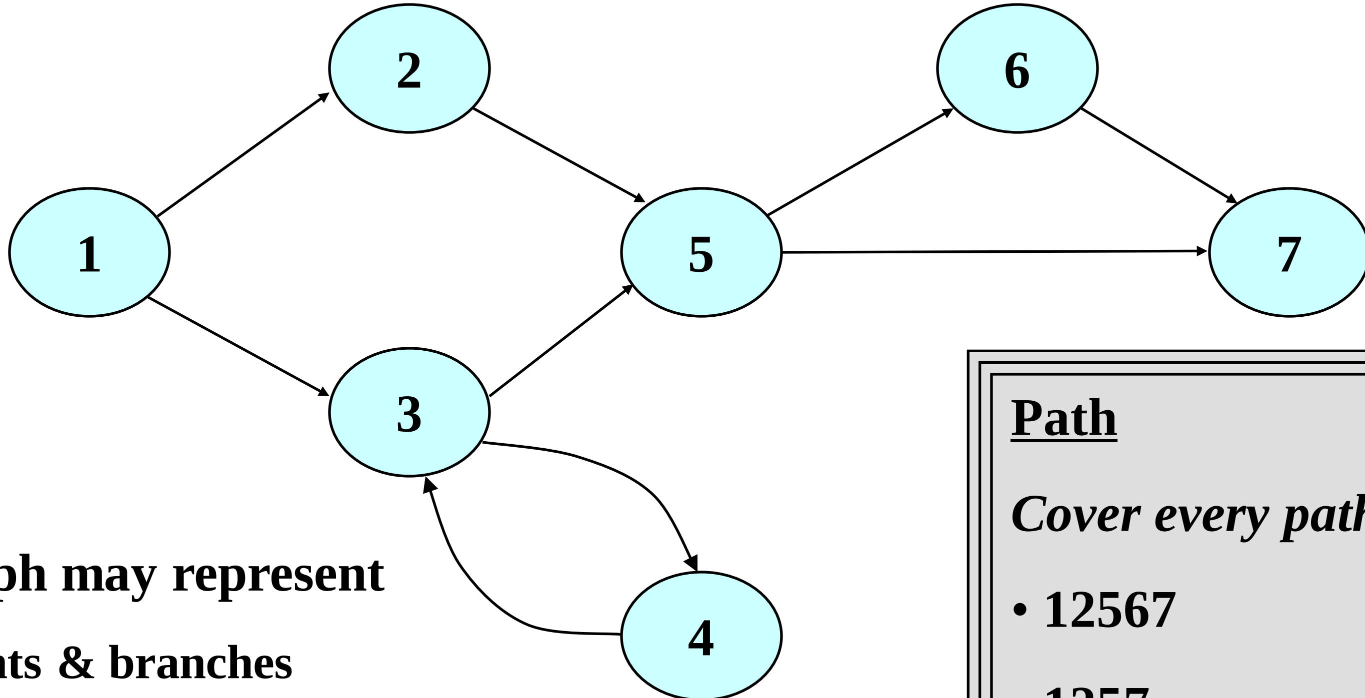
3. Input Domain Characterization

A: {0, 1, >1}
B: {600, 700, 800}
C: {swe, cs, isa, infs}

4. Syntactic Structures

```
if (x > y)
   z = x - y;
else
   z = 2 * x;
```

1. Graph Coverage – Structural



This graph may represent

- **statements & branches**
- **methods & calls**
- **components & signals**
- **states and transitions**

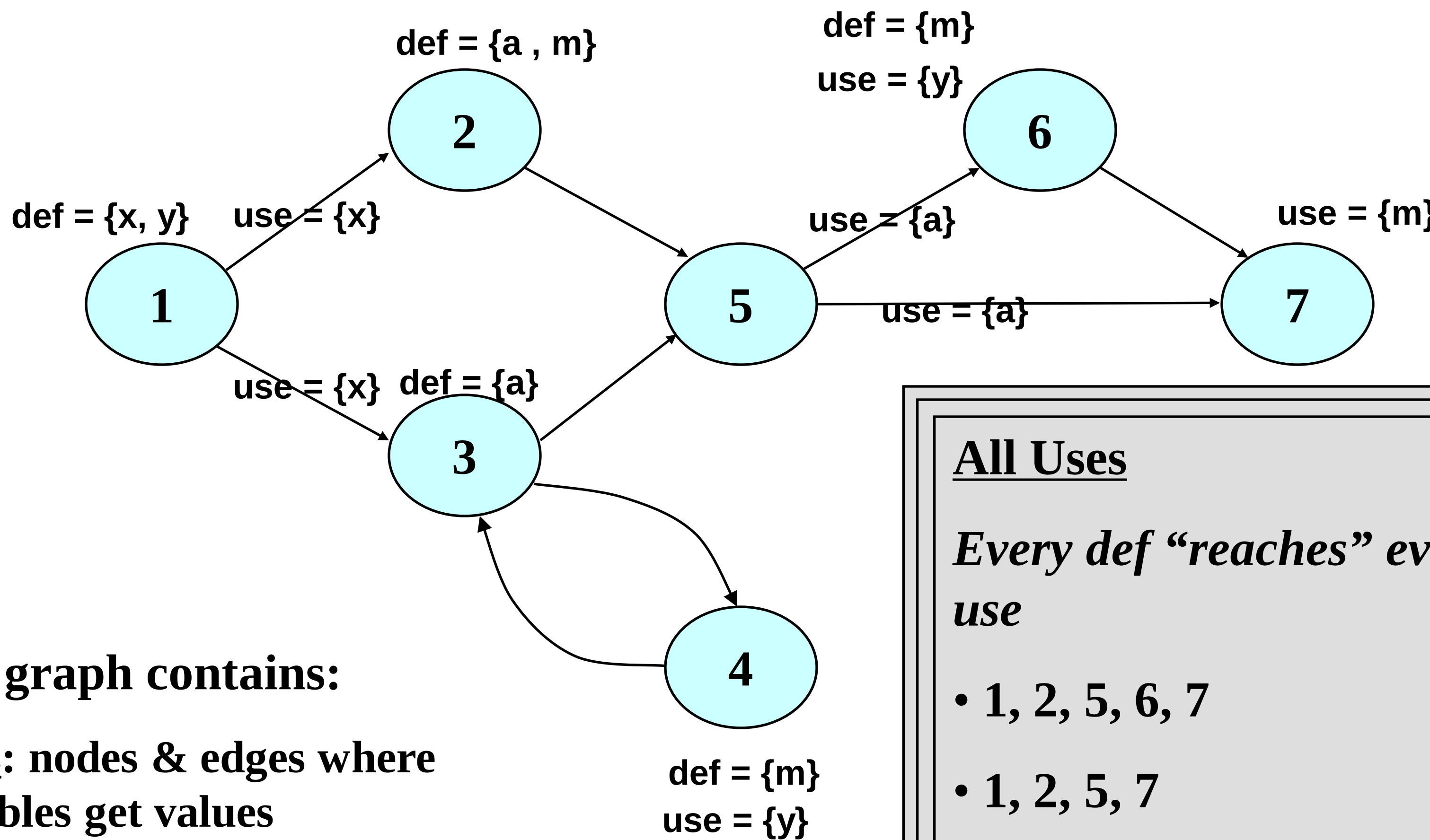
•
•
•

Path

Cover every path

- **12567**
- **1257**
- **13567**
- **1357**
- **1343567**
- **134357 …**

1. Graph Coverage – Data Flow



This graph contains:

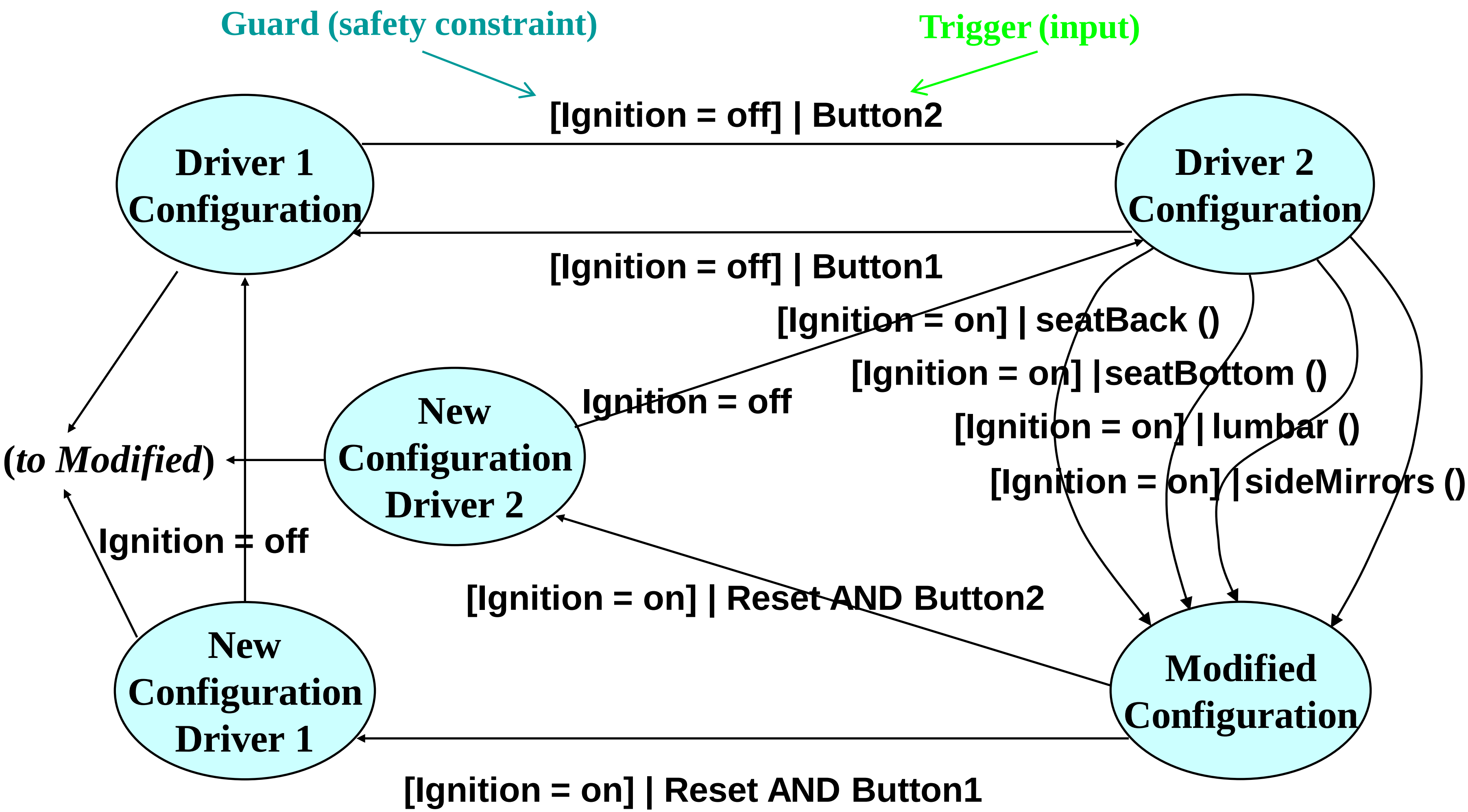
- **<u>defs</u>: nodes & edges where variables get values**
- **<u>uses</u>: nodes & edges where values are accessed**

<u>All Uses</u>

Every def "reaches" every use

- **1, 2, 5, 6, 7**
- **1, 2, 5, 7**
- **1, 3, 5, 6, 7**
- **1, 3, 5, 7**
- **1, 3, 4, 3, 5,7**

1. Graph - FSM Example
Memory Seats in a Lexus ES 300



2. Logical Expressions

$$((a > b) \text{ or } G) \text{ and } (x < y)$$

Transitions

Program Decision Statements

Software Specifications

Logical Expressions

$$((a > b) \text{ or } G) \text{ and } (x < y)$$

- Predicate Coverage : Each predicate must be true and false
  - $((a>b) \text{ or } G) \text{ and } (x < y)$ = True, False
- Clause Coverage : Each clause must be true and false
  - $(a > b)$ = True, False
  - G = True, False
  - $(x < y)$ = True, False
- Combinatorial Coverage : Various combinations of clauses
  - *Active Clause Coverage*: Each clause must determine the predicate's result

With these values for G and (x<y), (a>b) determines the value of the predicate

$((a > b)$ or $G)$ and $(x < y)$			
1	T	F	T
2	F	F	T
3	F	T	T
4	F	F	T
5	T	T	T
6	T	T	F

duplicate

- Describe the input domain of the software
  - Identify inputs, parameters, or other categorization
  - Partition each input into finite sets of representative values
  - Choose combinations of values
- System level
  - Number of students { *0, 1, >1* }
  - Level of course { *600, 700, 800* }
  - Major { *swe, cs, isa, infs* }
- Unit level
  - Parameters *F (int X, int Y)*
  - Possible values X*: { <0, 0, 1, 2, >2 }, Y : { 10, 20, 30 }*
  - Tests
    - *F (-5, 10), F (0, 20), F (1, 30), F (2, 10), F (5, 20)*

- Based on a grammar, or other syntactic definition
- Primary example is mutation testing
  1. Induce small changes to the program: mutants
  2. Find tests that cause the mutant programs to fail: killing mutants
  3. Failure is defined as different output from the original program
  4. Check the output of useful tests on the original program
- Example program and mutants

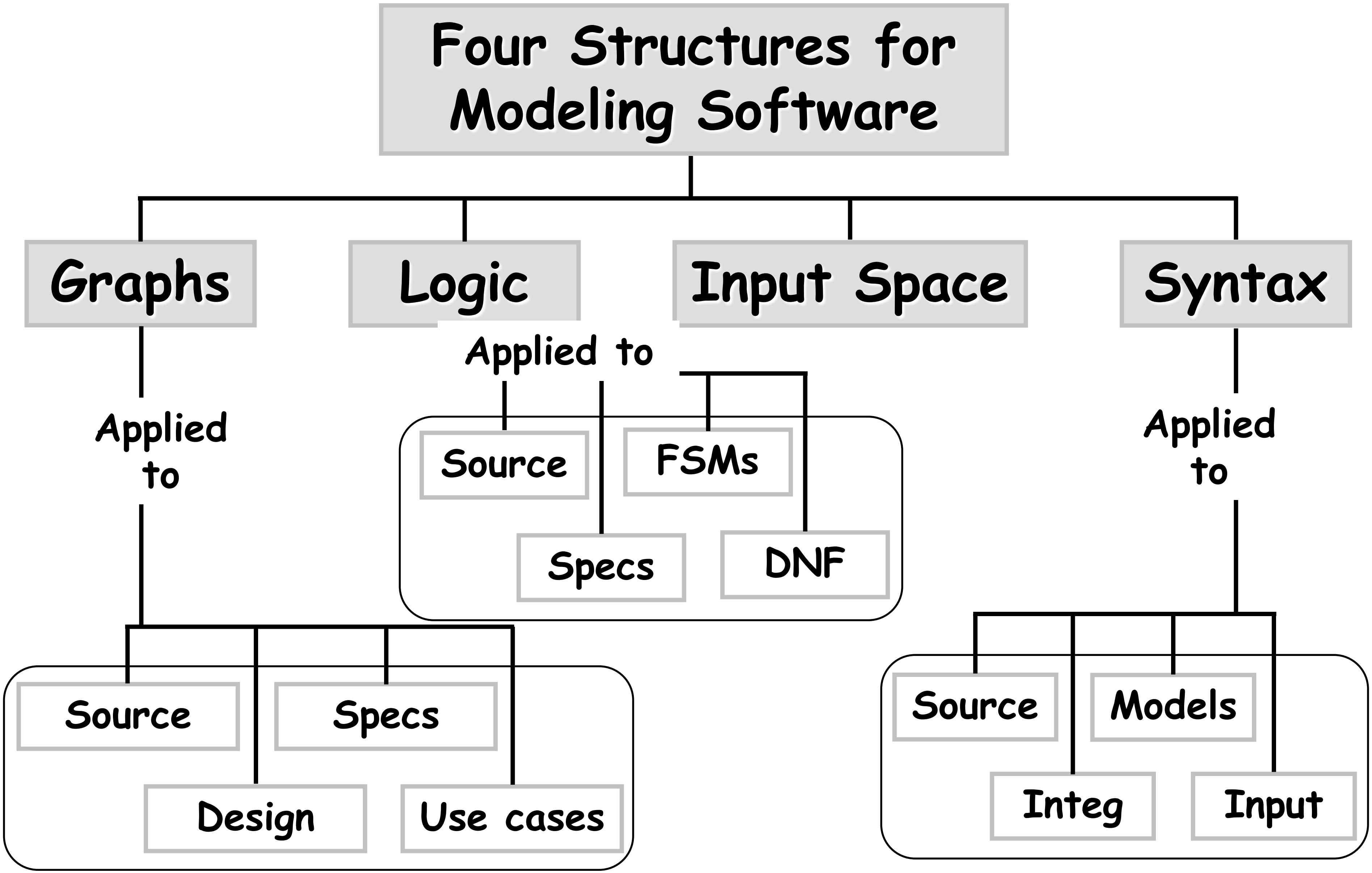
```
if (x > y)
   z = x - y;
else
   z = 2 * x;
```

```
if (x > y)
Δif (x >= y)
   z = x - y;
   Δ z = x + y;
   Δ z = x – m;
else
   z = 2 * x;
```

Source of Structures

- These structures can be extracted from lots of software artifacts
  - Graphs can be extracted from UML use cases, finite state machines, source code, …
  - Logical expressions can be extracted from decisions in program source, guards on transitions, conditionals in use cases, …
- *Model-based testing* derives tests from a model that describes some aspects of the system under test
  - The model usually describes part of the behavior
  - The source is usually *not* considered a model

Coverage Overview



Coverage

Given a set of test requirements *TR* for coverage criterion *C*, a test set *T* satisfies *C* coverage if and only if for every test requirement *tr* in *TR*, there is at least one test *t* in *T* such that *t* satisfies *tr*

- Infeasible test requirements : test requirements that cannot be satisfied
  - No test case values exist that meet the test requirements
  - Dead code
  - Detection of infeasible test requirements is formally undecidable for most test criteria
- Thus, 100% coverage is impossible in practice

Two Ways to Use Test Criteria

1. Directly generate test values to satisfy the criterion often assumed by the research community most obvious way to use criteria very hard without automated tools

2. Generate test values externally and measure against the criterion usually favored by industry
   - sometimes misleading
   - if tests do not reach 100% coverage, what does that mean?

Test criteria are sometimes called metrics

Generators and Recognizers

- Generator : A procedure that automatically generates values to satisfy a criterion
- Recognizer : A procedure that decides whether a given set of test values satisfies a criterion

- Both problems are provably undecidable for most criteria
- It is possible to recognize whether test cases satisfy a criterion far more often than it is possible to generate tests that satisfy the criterion
- Coverage analysis tools are quite plentiful

Comparing Criteria with Subsumption

- Criteria Subsumption : A test criterion *C1* subsumes *C2* if and only if every set of test cases that satisfies criterion *C1* also satisfies *C2*

- Must be true for every set of test cases
- *Example* : If a test set has covered every branch in a program (satisfied the branch criterion), then the test set is guaranteed to also have covered every statement

Test Coverage Criteria

- Traditional software testing is expensive and labor-intensive
- Formal coverage criteria are used to decide which test inputs to use
- More likely that the tester will find problems
- Greater assurance that the software is of high quality and reliability
- A goal or stopping rule for testing
- Criteria makes testing more efficient and effective

How ?

Now we know why and what ...

How do we get there ?

Testing Levels Based on Test Process Maturity

- Level 0 : There's no difference between testing and debugging
- **Level 1 : The purpose of testing is to show correctness**
- **Level 2 : The purpose of testing is to show that the software doesn't work**
- **Level 3 : The purpose of testing is not to prove anything specific, but to reduce the risk of using the software**
- **Level 4 : Testing is a mental discipline that helps all IT professionals develop higher quality software**

Level 0 Thinking

- Testing is the same as debugging
- Does not distinguish between incorrect behavior and mistakes in the program
- Does not help develop software that is reliable or safe

Level 1 Thinking

- Purpose is to show correctness
- Correctness is impossible to achieve
- What do we know if no failures?
  - Good software or bad tests?
- Test engineers have no:
  - Strict goal
  - Real stopping rule
  - Formal test technique
  - Test managers are powerless

This is what hardware engineers often expect

Level 2 Thinking

- Purpose is to show failures
- Looking for failures is a negative activity
- Puts testers and developers into an adversarial relationship
- What if there are no failures?

This describes most software companies.

How can we move to a *team approach* ??

Level 3 Thinking

- Testing can only show the presence of failures
- Whenever we use software, we incur some risk
- Risk may be small and consequences unimportant
- Risk may be great and the consequences catastrophic
- Testers and developers work together to reduce risk

This describes a few "enlightened" software companies

Level 4 Thinking

<u>A mental discipline that increases quality</u>

- Testing is only one way to increase quality
- Test engineers can become <u>technical leaders</u> of the project
- Primary responsibility to <u>measure and improve</u> software quality
- Their expertise should <u>help the developers</u>

This is the way "traditional" engineering works

Summary

- Why do we test – to reduce the risk of using the software
- Four types of test activities – test design, automation, execution and evaluation
- Software terms – faults, failures, the RIP model, observability and controllability
- Four structures – test requirements and criteria
- Test process maturity levels – level 4 is a mental discipline that improves the quality of the software

References

- Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008
- Mauro Pezze and Michal Young, *Software Testing and Analysis: Process, Principles and Techniques*, John Willy & Sons, 2007