Software Testing

OVERVIEW

Program Testing

Testing is intended to

- show that a program does what it is intended to do

… and …

- to discover program defects before it is put into use.

"Testing can only show the presence of errors, NOT their absence."

Program Testing

The goals of testing are to demonstrate to the developer and the customer

- that the software meets its requirements
- to discover defects that cause incorrect or undesirable behavior

- For custom software … there should be at least one test for every requirement in the requirements specification.
- For generic software … there should be tests for all of the individual features, plus combinations of features, in the released product.

Program Testing

The goals of testing are to demonstrate to the developer and the customer

- that the software meets its requirements [*Validation Testing*]
- to discover defects that cause incorrect or undesirable behavior [*Defect or Verification Testing*]

- For custom software … there should be at least one test for every requirement in the requirements specification.
- For generic software … there should be tests for all of the individual features, plus combinations of features, in the released product.

Program Testing

- When you test software, you execute a program using artificial data.
  - Ideally, a known input should produce a known, testable, output.
- You check the results of the test run for errors, anomalies and/or metrics about the program's non-functional attributes.

Program Testing

Testing is part of a more general verification and validation (V&V) process

- May also include static validation techniques such as inspections and reviews

Validation testing:

- Demonstrate to the developer and customer that the software meets its requirements.
- Success shows that the system operates as intended.

Defect / Verification testing:

- Discover faults or defects in the software where the behavior is incorrect or not in agreement with the specification.
- Success means the system performed incorrectly and exposes a defect in the system.

Program Testing

Validation testing:

Are we building the right product?

Verification testing:

Are we building the product right?

Inspections v. Testing

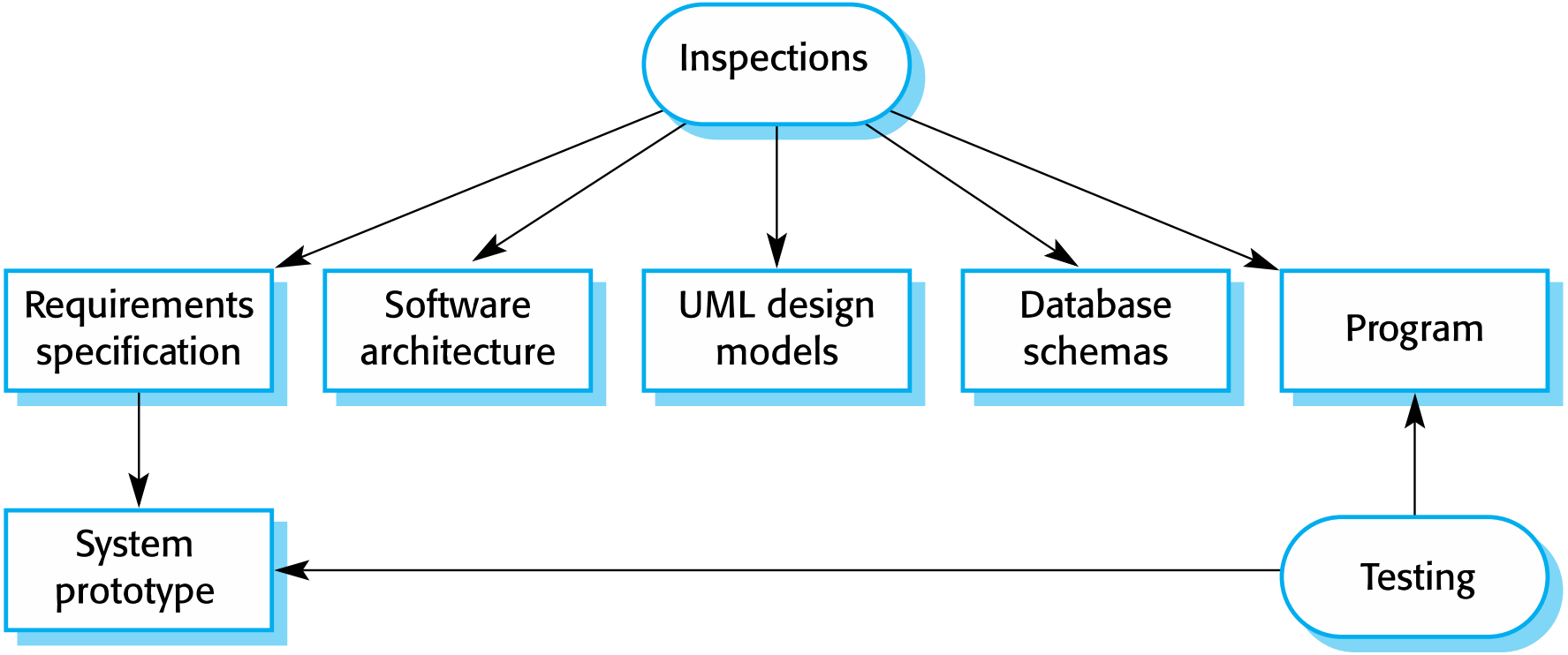
Software inspections:

- Analysis of the static system representation to discover problems (static verification)
- “Static” testing … don’t have to run the code.
- Applies to the design and requirements document, too.

Software testing:

- Execute and observe the product behavior
- “Dynamic”
- The system is executed with test data and its behavior is observed.

Inspections v. Testing



Inspections

- Examining the source (or document) searching for anomalies and defects.
- Does not require execution of a system so may be used before implementation.
- May apply to any representation of the system (requirements, design, configuration data, test data, etc.).
- Cost effective technique for discovering program errors.

Inspections: Pros

- Testing errors can mask other errors.
  - Inspection is static so you don't have to be concerned with interactions between errors.
- Incomplete systems can be inspected without added cost.
  - You need to develop specialized test harnesses to test the parts that are available in testing.
- Can consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

Inspections and testing are complementary … do both during the V&V process.

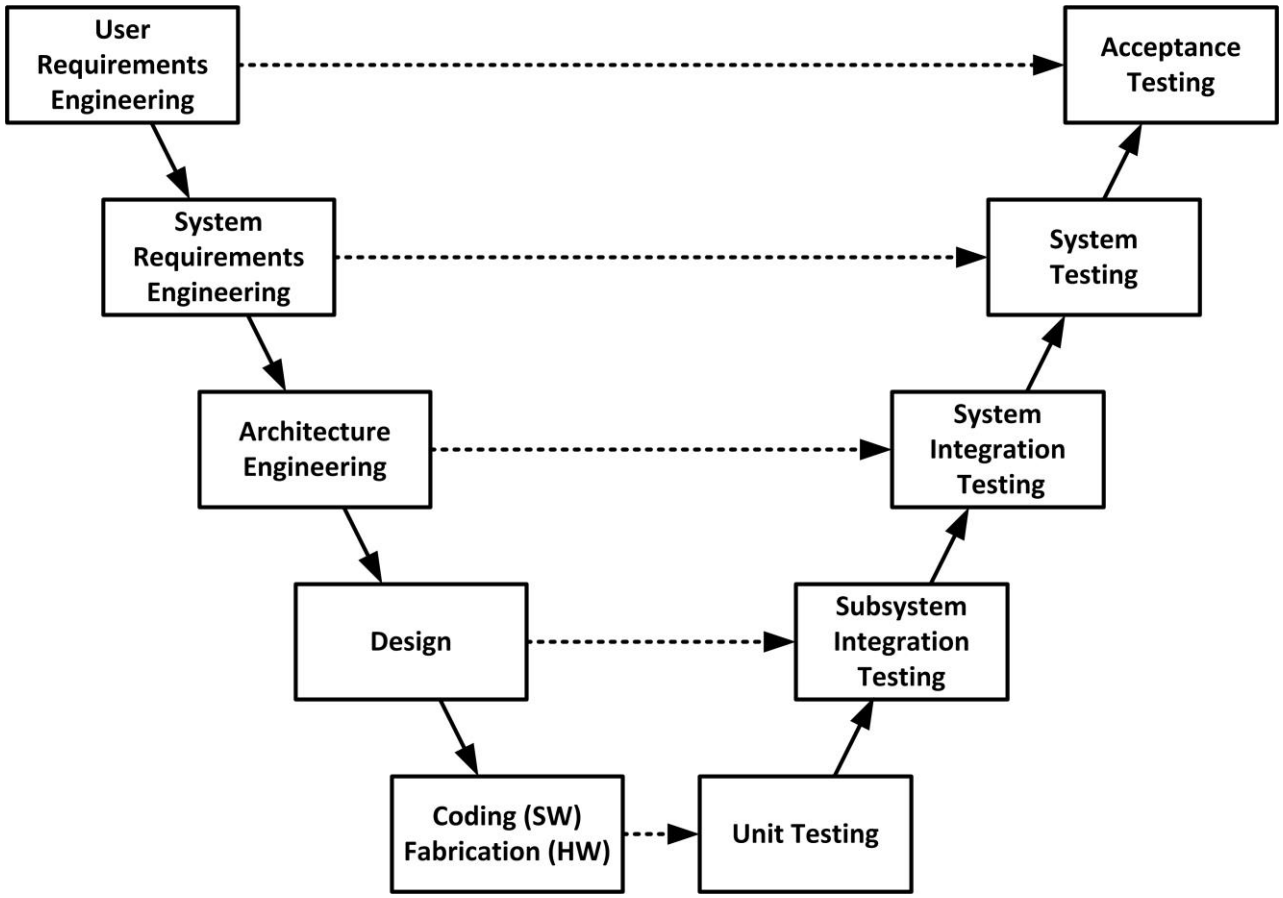
Inspections: Cons

- Can check conformance with a specification but NOT conformance with the customer's real requirements (or expectations).
- Can NOT check non-functional characteristics such as performance, usability, etc.

Testing Stages

- Development testing:
  - Test the system during development to discover bugs and defects
- Release testing:
  - A separate testing team tests a complete version of the system before it is released to users.
- User testing:
  - Users or potential users of a system test the system in their own environment.

Testing Stages



Release Testing

- Test a particular release of a system that is intended for use outside of the development team.
- The primary goal is to convince the supplier of the system (us) that it is good enough for use.
  - Therefore, it has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- Usually *black-box testing* where tests are only derived from the system specification.
  - The code is not considered … an opaque box.

Release Testing

A form of system testing but there are important differences:

- A separate team that has NOT been involved in the system development, should be responsible for release testing.
- System testing by the development team should focus on discovering bugs in the system (defect / verification testing).
  - The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

Release Testing

- Requirements-based testing involves examining each requirement and developing one or more tests for it.
- Mentcare system use-case (format) requirements:
  - *If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.*
  - *If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.*

Release Testing

1. Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system. (No warning)
2. Set up a patient record with a known allergy. Prescribe the medication to that the patient is allergic to, and check that the warning is issued by the system. (One warning)
3. Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued. (One warning per test)
4. Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued. (Two warnings, one test)
5. Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled. (Override)

Scenario Testing

- Type of release testing that forms a ‘typical’ usage scenario.
  - Tries to mimic a normal user’s operation of the system.
  - … Think … role-play.
- Can be recycled from requirements elicitation user stories.
- Often tests many requirements at once … good way to stress the interaction of requirements.

Scenario Testing Example

- During the scenario testing, you role-play and observe the system behavior using the synthetic user behavior.
  - Users make mistakes so these tests should also enter erroneous input to test that the system catches all input errors.

Scenario Testing Example

George is a nurse who specializes in mental healthcare. One of his responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, George logs into the Mentcare system and uses it to print his schedule of home visits for that day, along with summary information about the patients to be visited. He requests that the records for these patients be downloaded to his laptop. He is prompted for his key phrase to encrypt the records on the laptop.

One of the patients, Jim, is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. George looks up Jim's record and is prompted for his passphrase to decrypt the record. George checks the prescribed drug's side effects. Sleeplessness is a known side effect so he adds a note in Jim's record and suggests that he visits the clinic to have his medication changed. Jim agrees so George enters a reminder to call Jim when he gets back to the clinic to make an appointment with a physician. George ends the consultation and the system re-encrypts Jim's record.

After, finishing his consultations, George returns to the clinic and uploads the records of patients visited to the database. The system generates a list patients that George must contact for follow-up information and make clinic appointments.

Scenario Testing Example

1. Authentication by logging on to the system.
2. Downloading / uploading of select (encrypted) patient records to a mobile device.
3. Create home visit schedule.
4. Encrypt / decrypt patient records on a mobile device.
5. Record retrieval and modification.
6. Links with the drugs database that maintains side-effect information.
7. The system for call prompting.

Performance Testing

- Goal is to test the emergent properties of a system, such as performance and reliability.
  - Some properties may be non-functional requirements such as response time.
- Performance testing usually involve a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behavior.
  - Investigate how far outside the specification the system can go.

User Testing

- User or customer testing: real users or customers provide input and advice on system testing.
- User testing is essential, even when comprehensive system and release testing have been carried out.
- User testing is usually executed in their own environment which may have a major effect on the reliability, performance, usability and robustness of a system.
  - These cannot always be replicated in a testing environment.

User Testing

Alpha testing

- Users of the software work with the development team to test the software, usually at the developer's site.

Beta testing

- A release of the software is made available to select users to allow them to experiment and to raise problems that they discover with the system developers.

Acceptance testing

- Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

Acceptance Testing in Agile

Acceptance testing is usually executed the customer and well removed from the development team.

In agile, the customer is integrated with the development team though. He/she is also responsible for judging the acceptability of the system.

- Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made.
- There is no separate acceptance testing process.
- Main problem here is if the embedded user is ‘typical’ and can represent the interests of all system stakeholders … or, worse, have they ‘gone native’ and think code.

Development Testing

Development testing includes all testing activities that are carried out by the team developing the system.

- *Unit testing*: test individual program units or object classes.
  - focus on testing the functionality of objects or methods.
- *Component testing*: several individual units are integrated to create composite components.
  - focus on testing component interfaces.
- *System testing*: some or all of the system's components are integrated and tested as a whole.
  - focus on testing component interactions.

Unit Testing

Test individual components in isolation.

- Focused on detecting defects
- Developers write these using the code itself (and the specification) to design the tests: *white-box or clear-box (or glass-box) testing*
- Units may be:
  - Individual functions or methods within an object
  - Object classes with several attributes and methods
  - Composite components with defined interfaces used to access their functionality.

Object Class Testing

Complete test coverage of a class involves …

- Testing all operations associated with an object … more on this later.
- Setting and testing all object attributes
- Exercising the object in all possible states
- Inheritance makes it more difficult to test object classes since the information to be tested is no longer local.

Automated Unit Testing

- Unit testing should be automated so that tests can be run and checked without manual intervention.
  - The easier and faster the testing process, the more frequently it will be used.
- Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success or failure of the tests.
  - Example = JUnit

Automated Unit Testing

Executable unit tests consists of 3 parts:

- *Setup*: initialize the system with the test case input and expected outputs.
- *Call*: call the object or method to be tested.
- *Assert*: compare the result of the call with the expected result.

IF the assertion evaluates to true;

THEN the test has been successful;

ELSE it failed;

How to choose unit tests?

- The test cases should show that, when used as defined, the component that you are testing does what it is supposed to do.
- If there are defects in the component, these should be revealed by test cases.
- … and we don't want infinite tests!
- Two types of unit test case:
  1. Should reflect normal operation of a program and should show that the component works as expected. (Partition testing)
  2. Should use knowledge of where common problems arise. Use abnormal inputs to check that these are properly processed and do not crash the component. (Guideline testing)

How to choose unit tests?

- *Partition testing*: identify groups of inputs with common characteristics that "should" be processed in the same way.
  - Choose tests from within each of these groups.
- *Guideline testing*: testing guidelines to choose test cases.
  - Guidelines reflect prior knowledge of common programming errors.
    - Example: test for null lists or empty lists

Partition Testing

Partition testing: identify groups of inputs with common characteristics that "should" be processed in the same way.

- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition (or domain) … the program behaves in an equivalent way for members of the EP's.
- Choose a test case from each partition.

Partition Testing

Example: convert % grade to letter grade

- The set {93-100} all behave the same … A.
- The set {83-86} all behave similar … B.

A	93
A-	90
B+	87
B	83
B-	80
C+	77
C	73
C-	70
D+	67
D	63

- *Pick values in the middle of the EP's and use those as unit tests.*
- *One test for each EP.*
- *No need for 100 separate tests.*

Partition Testing

Example: convert % grade to letter grade

- The set {92-94} straddle EP's.
- The set {86-88} straddle EP's.

A	93
A-	90
B+	87
B	83
B-	80
C+	77
C	73
C-	70
D+	67
D	63

- *Also pick values that straddle EP's*
  - *A.K.A. … Bounding Value Analysis (BVA)*
- *Common error: X < Y instead of X <= Y*

"Bugs lurk in corners and congregate at boundaries."[*]

Testing with guidelines

Sequences:

- Test the common errors often overlooked with lists and loops.
  - We usually think of sequences are 2+ objects and we often program with that assumption.
  - Test with sequences with only a single value.
  - Use sequences of different sizes in different tests.
  - Design tests to touch the first, middle and last elements of the sequence.
  - Test with zero length sequences.

Testing with guidelines

- Choose inputs that force the system to throw all error messages.
- Design inputs that cause input buffers to overflow.
  - Security and stability test
- Repeat the same input or series of inputs numerous times.
  - This can detect errors in the object state.
- Force invalid outputs to be generated.
- Force computation results to be too large or too small.
  - Under- or overflow conditions, especially with object counters or sequence lengths.

Testing with guidelines

Branches:

- Choose inputs to force True and False for all conditionals.
- This can have explosive (exponential) growth so usually must narrow scope to a single member or class
  - 2^N cases if all Boolean expressions are independent.
- Helpful to build decision table with possible inputs and outputs (the *input/output oracle*).

Testing with guidelines

Example: Monopoly Game …

If a Player (A) lands on property owned by another player (B), (A) must pay rent to (B). If (A) does not have enough money to pay (B), (A) is out of the game.

Conditions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
A lands on B's property	T	T	F	F
A has enough money	T	F	T	F
Actions				
A stays in game	T	F	T	T

Rule 4 is not needed due to conditional dependence

Mechanics of Testing

We often generate *scaffolding* code to support testing.

Code and data created only for testing and not for the product.

- *Driver*: Instantiates objects, delivers input and tests output.
- *Stub*: code with matching signature but w/o the implemented logic. Returns valid (and controllable) results.
  - Lets us test component interfaces that do not exist (yet).
- *Mock objects*: Often need to create stand-ins for data producers or consumers components to ease testing
  - Example: replace database component with simple array object

Component Testing

Test for interface errors that lead to defects.

- Interface test types
  - Parameters: data passed to method from caller.
  - Shared memory: Block of memory is shared between functions or objects … for functions that are not pure.
  - Message passing: sub-systems request services from other sub-systems (e.g., client-server timing errors)

Component Testing

Guidelines:

- Pass arguments are the extreme ends of their ranges.
  - Good for testing overflow parameter errors.
- Pass null for pointer parameters.
- Design tests that cause the component to fail.
- Stress test message passing systems.
  - Large/empty messages and exceed specified rate.
- Vary the sequence in which components are activated or data accessed in shared-memory systems.
  - Embedded systems often use shared-memory pools.
  - Sequence and timing of locks and mods can alter behavior.

Test driven development (TDD)

- Test-driven development (TDD) interleaves testing and code development.
- Tests are designed and written before the code.
- You develop code incrementally, along with a test for that increment. Don’t move on to the next increment until the code that you have developed passes its test.
- TDD was introduced as part of agile methods such as XP.
  - .. but it can also be used in plan-driven development.

Test driven development (TDD)

- Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code. (i.e., a single task)
- Write a test for this functionality (i.e., a driver) as an automated test.
- Run the new test, along with all prior tests. It should fail initially since you haven't implemented the functionality.
  - This is a good test of the test.
- Now, implement the functionality and rerun the test.
- When all tests pass, you move on to implementing the next chunk of functionality.

TDD Pros

- Code coverage
  - Every code segment that you write has at least one associated test so all code written has at least one test.
- Regression testing
  - A regression test suite is developed incrementally as a program is developed.
- Simplified debugging
  - When a test fails, it should be obvious where the fault lies.
- System documentation
  - The tests themselves are a form of documentation that describe what the code should be doing. (i.e. Agile manifesto)

Testing Summary

- Testing can only show the presence of errors in a program … **not their absence.**
- Development testing is the responsibility of the software development team.
  - A separate team should be responsible for testing a system before it is released to customers.
  - Unit testing: test individual objects and methods
  - Component testing: test related groups of objects
  - System testing: test partial or complete systems.

Testing Summary

- When testing, try to **break** the software by choosing types of test cases that test edge (or boundary) cases.
- Wherever possible, automate the tests.
  - Embedded/automated tests can be run every time a change is made to a system.
- In test-first development (TDD), tests are written before the code to be tested.
- Scenario testing involves inventing a typical usage scenario and using this to derive test cases.
- Acceptance testing's aim is to decide if the software is good enough to be deployed and used in its operational environment by the intended users.

Testing Summary

Testing Type	Specification	General Scope	Opacity	Who generally does it?
Unit	Low-Level Design Actual Code Structure	Small unit of code no larger than a class	White Box	Programmer who wrote code
Integration	Low-Level Design High-Level Design	Multiple classes	White Box Black Box	Programmers who wrote code
Functional	High Level Design	Whole product	Black Box	Independent tester
System	Requirements Analysis	Whole product in representative environments	Black Box	Independent tester
Acceptance	Requirements Analysis	Whole product in customer's environment	Black Box	Customer
Beta	Ad hoc	Whole product in customer's environment	Black box	Customer
Regression	Changed Documentation High-Level Design	Any of the above	Black Box White Box	Programmer(s) or independent testers

How to Derive Test Cases?

We want to achieve high coverage … but at manageable costs.

Coverage? … the measure/metric of the completeness of the set of test cases.

- Method coverage: % of methods tests. (Very weak)
- Statement coverage: % of statements executed. (Weak)
- Branch coverage: % of decision/branch statements evaluated at T or F. (Stronger, may be hard to test 100%)
- Conditional coverage: % of Boolean expressions evaluated as T of F. (Strong but expensive)

How to Derive Test Cases?

Method coverage: 100% coverage with `F1(0,0,0,0)`

Test #	Test	Result	Coverage
1	F1(0,0,0,0)	0	100% Method

```
1 int F1(int a, int b, int c, int d){
2       if (a == 0)
3           return 0;
4       int x = 0;
5       if ((a == b) OR
6           ((c == d) AND F2(a)))
7           x = 1;
8       float e = 1 / x;
9       return e;
10}
```

How to Derive Test Cases?

Branch coverage: 37% coverage with on `F1(0,0,0,0)`

Test #	Test	Result	Coverage
1	F1(0,0,0,0)	0	37%
2	F1(1,1,1,1)	1	100%

```
1 int F1(int a, int b, int c, int d){
2       if (a == 0)
3           return 0;
4       int x = 0;
5       if ((a == b) OR
6           ((c == d) AND F2(a)))
7           x = 1;
8       float e = 1 / x;
9       return e;
10}
```

How to Derive Test Cases?

Statement coverage: 50% coverage with Tests 1-2

Test #	Test	Result	Coverage
1	F1(0,0,0,0)	0	
2	F1(1,1,1,1)	1	50%
3	F1(1,2,1,2)	NaN/Error	100%

Test #	Line 2	Line 5-6
1	T	--
2	F	T
3	F	F

```
1 int F1(int a, int b, int c, int d){
2       if (a == 0)
3           return 0;
4       int x = 0;
5       if ((a == b) OR
6           ((c == d) AND F2(a)))
7           x = 1;
8       float e = 1 / x;
9       return e;
10}
```

How to Derive Test Cases?

Test #	Test	Result	Coverage
1	F1(0,0,0,0)	0	20%
2	F1(1,1,1,1)	1	40%
3	F1(1,2,1,2)	NaN/Error	60%
4	F1(1,2,1,1)	1	80%
5	F1(2,2,1,1)	NaN/Error	100%

Test #	a==0	a==b	c==d	F2(a)
1	T	--	--	--
2	F	T	--	--
3	F	F	F	--
4	F	F	T	T
5	F	F	T	F

```
1 int F1(int a, int b, int c, int d){
2       if (a == 0)
3           return 0;
4       int x = 0;
5       if ((a == b) OR
6           ((c == d) AND F2(a)))
7           x = 1;
8       float e = 1 / x;
9       return e;
10}
```

Short Circuit Conditions

4 Booleans
would require
2^4 tests for exhaustive coverage

But Boolean short-circuit eliminates many paths.

a==0	a==b	c==d	F2(a)
F	F	F	F
T	F	F	F
F	T	F	F
T	T	F	F
F	F	T	F
T	F	T	F
F	T	T	F
T	T	T	F
F	F	F	T
T	F	F	T
F	T	F	T
T	T	F	T
F	F	T	T
T	F	T	T
F	T	T	T
T	T	T	T

Short Circuits

Tests 4 and 5 test 1 condition each

Test #	a==0	a==b	c==d	F2(a)
1	T	--	--	--
2	F	T	--	--
3	F	F	F	--
4	F	F	T	T
5	F	F	T	F

a==0	a==b	c==d	F2(a)
F	F	F	F
T	F	F	F
F	T	F	F
T	T	F	F
F	F	T	F
T	F	T	F
F	T	T	F
T	T	T	F
F	F	F	T
T	F	F	T
F	T	F	T
T	T	F	T
F	F	T	T
T	F	T	T
F	T	T	T
T	T	T	T

Short Circuits

Test 3 tests 2 conditions at once since F2 is shorted.

Test #	a==0	a==b	c==d	F2(a)
1	T	--	--	--
2	F	T	--	--
3	F	F	F	--
4	F	F	T	T
5	F	F	T	F

a==0	a==b	c==d	F2(a)
F	F	F	F
T	F	F	F
F	T	F	F
T	T	F	F
F	F	T	F
T	F	T	F
F	T	T	F
T	T	T	F
F	F	F	T
T	F	F	T
F	T	F	T
T	T	F	T
F	F	T	T
T	F	T	T
F	T	T	T
T	T	T	T

Short Circuits

Test 2 tests 4 conditions at once since c==d and F2 are shorted.

Test #	a==0	a==b	c==d	F2(a)
1	T	--	--	--
2	F	T	--	--
3	T	F	F	--
4	T	F	T	T
5	T	F	T	F

a==0	a==b	c==d	F2(a)
F	F	F	F
T	F	F	F
F	T	F	F
T	T	F	F
F	F	T	F
T	F	T	F
F	T	T	F
T	T	T	F
F	F	F	T
T	F	F	T
F	T	F	T
T	T	F	T
F	F	T	T
T	F	T	T
F	T	T	T
T	T	T	T

Short Circuits

Test 2 tests 4 conditions at once since c==d and F2 are shorted.

Test #	a==0	a==b	c==d	F2(a)
1	T	--	--	--
2	F	T	--	--
3	T	F	F	--
4	T	F	T	T
5	T	F	T	F

a==0	a==b	c==d	F2(a)
F	F	F	F
T	F	F	F
F	T	F	F
T	T	F	F
F	F	T	F
T	F	T	F
F	T	T	F
T	T	T	F
F	F	F	T
T	F	F	T
F	T	F	T
T	T	F	T
F	F	T	T
T	F	T	T
F	T	T	T
T	T	T	T

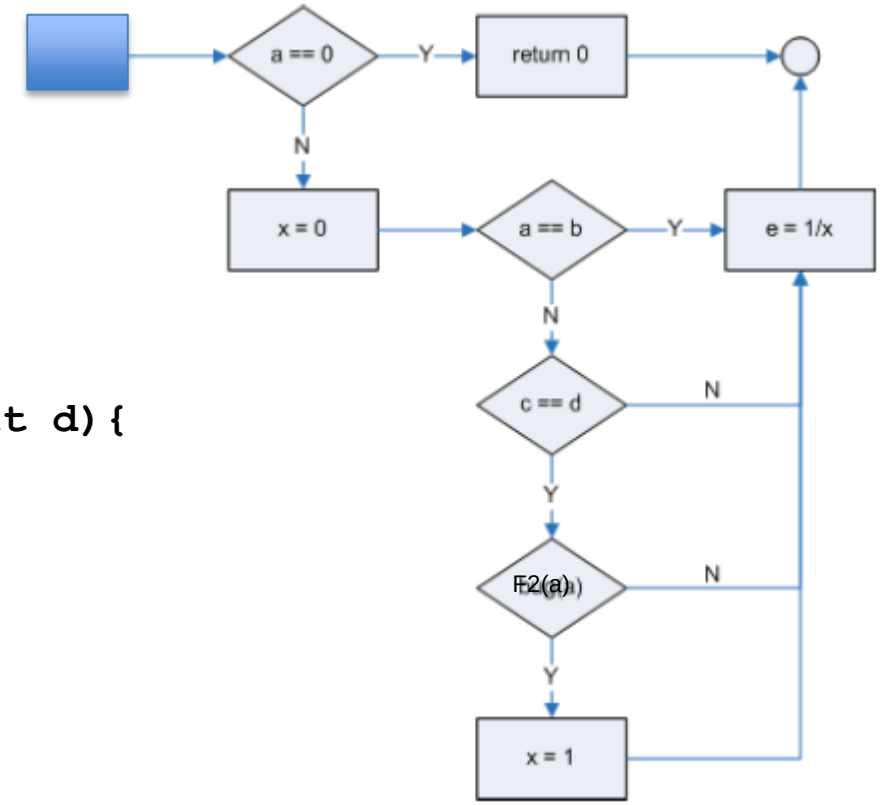
Flow Charts

Basis Path Testing is a strategy for ensuring that all independent code paths have been tested.

- Edges – Nodes + 2 = cyclomatic # … gives lower bound on the # of tests required.
- At least 1 test per independent path.

Flow Charts

- 10 Nodes
- 13 Edges
- Cyclomatic # = 5 … 5 tests covered the paths.




```
1 int F1(int a, int b, int c, int d){
2       if (a == 0)
3           return 0;
4       int x = 0;
5       if ((a == b) OR
6           ((c == d) AND F2(a)))
7           x = 1;
8       float e = 1 / x;
9       return e;
10}
```