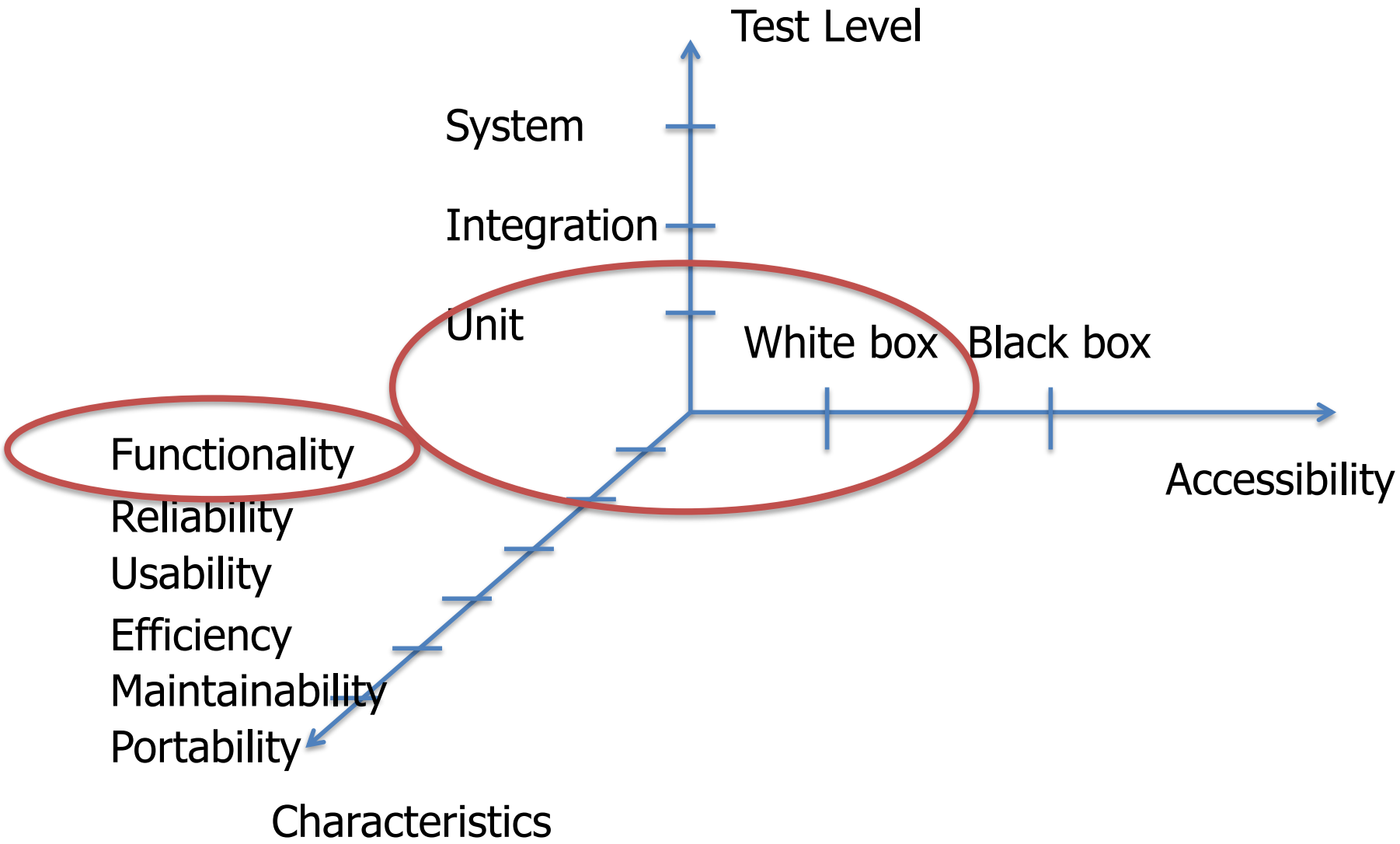
DA616 Advancement in SW development
DA390A Framsteg inom Programvaruutveckling

Lecture on: Unit testing and TDD

Today's lecture

– Testing
  - Techniques: (repetition of) Equivalence class, Boundary Value Analysis and (new) **State transition testing**
  - Code Coverage – (repetition of) Statement, Branch, Path and (new) **Condition**
  - **Unit tests and using test frameworks (Junit)**
  - **TDD – Test Driven Development**

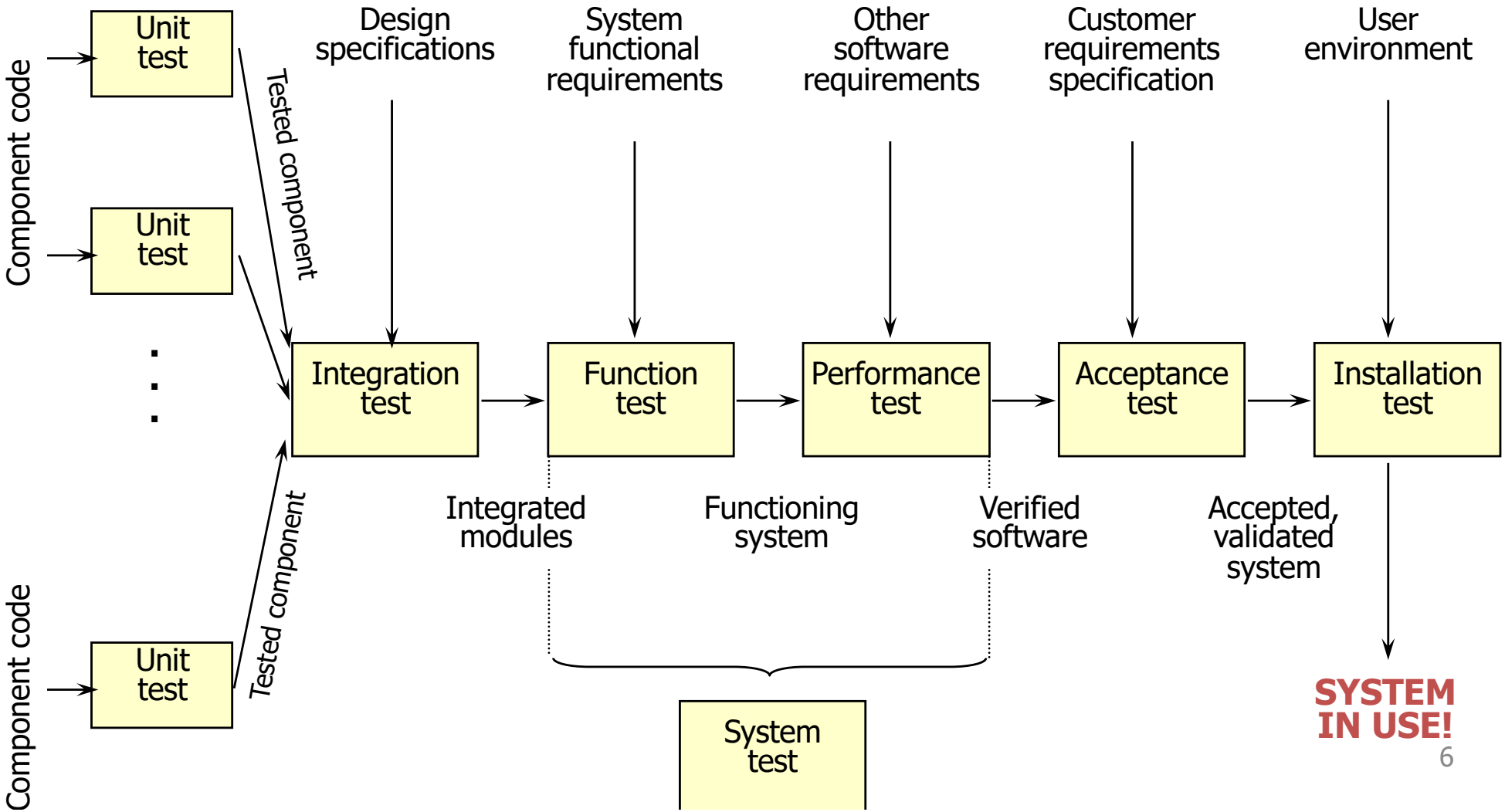
Types of Testing



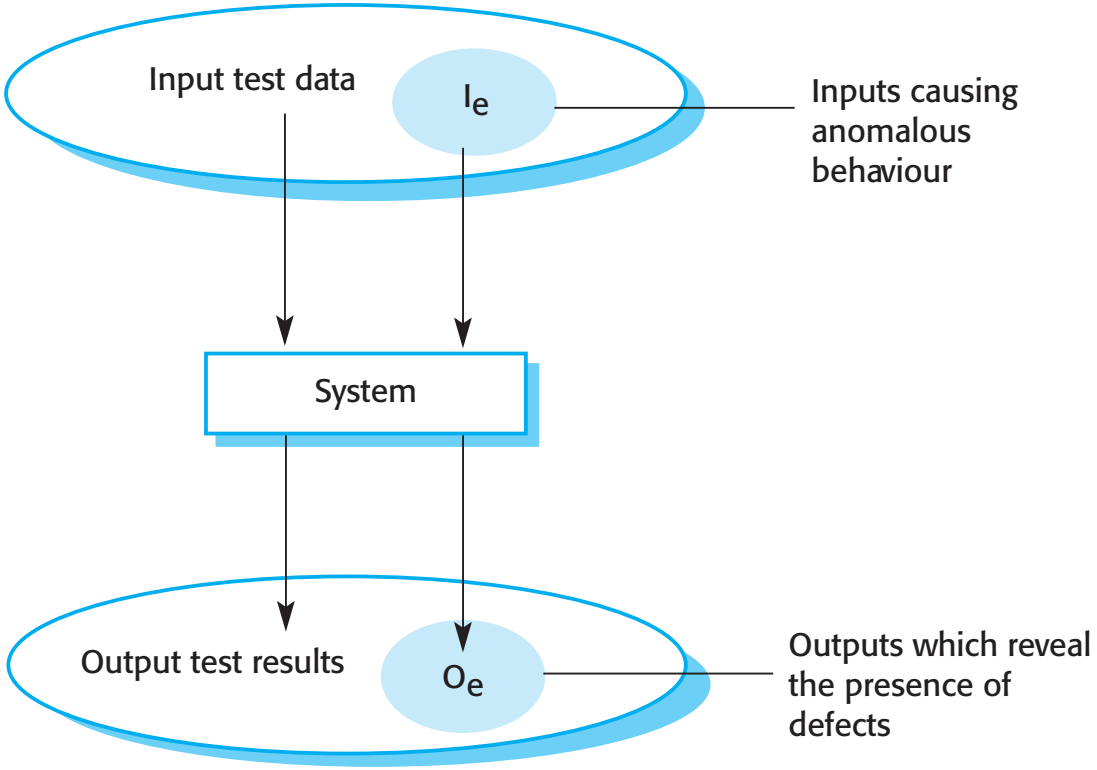
Definitions

- Humans make ERRORS, that may lead to SW containing FAULTs. When executing such SW it may lead to FAILURE.
- FAULTS are also sometimes called “BUGS” or “DEFECTS”.
- When testing, we need TEST CASES, that contains:
  1. a set of input criteria,
  2. execution conditions (including input values) and
  3. expected output.
- When deciding what the expected output shall be, we may need a “TEST ORACLE”
- Also, be careful with the word “TEST”, which sometimes means “EXPERIMENT”. Here we often mean “VERIFY”.

General: Testing Phases



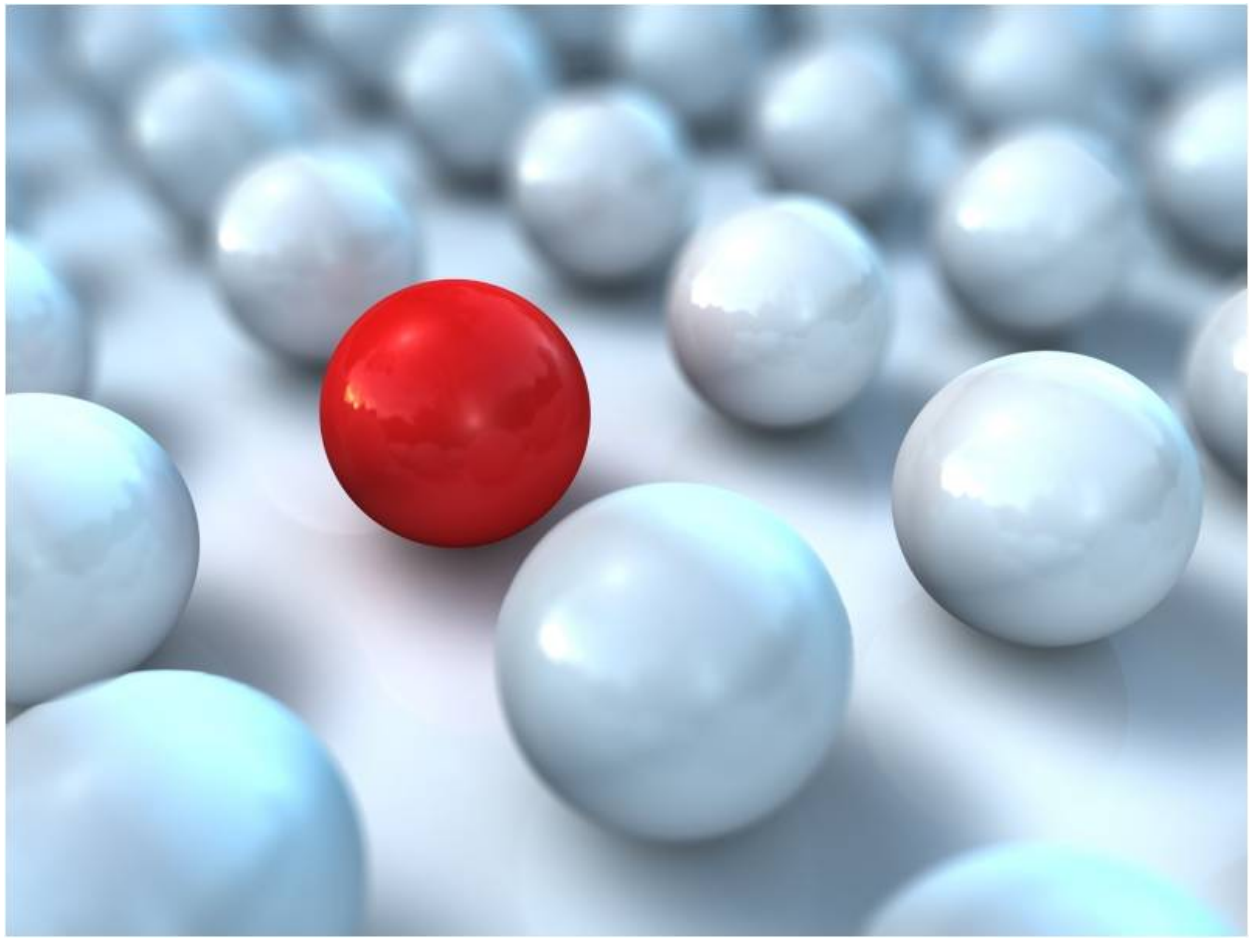
General: An input-output model of SW testing



The difficult part of testing is to select input test data so that defects can be detected. It may also be difficult to decide if the output is correct of if it shows indication of a defect in the SW.

Unit Test

Parts of the code behaves as intended.



Unit Testing

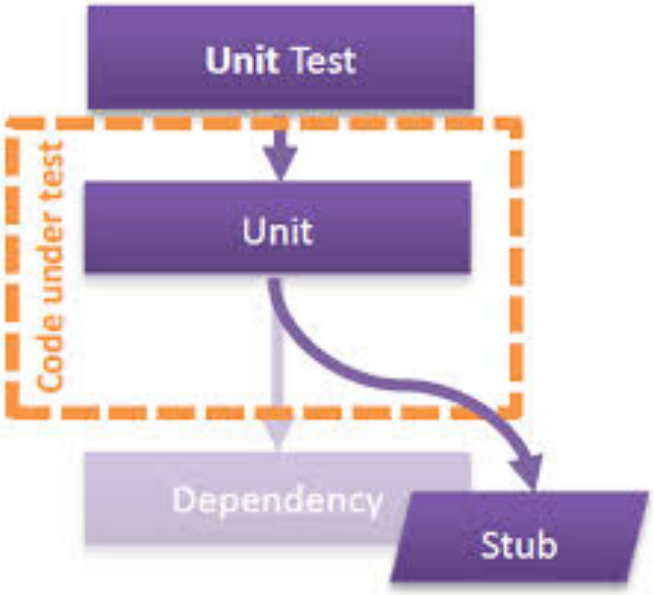
Definition: Tests the smallest individually executable code units in isolation.

Objective: Find faults in the units. Assure correct functional behavior of units.

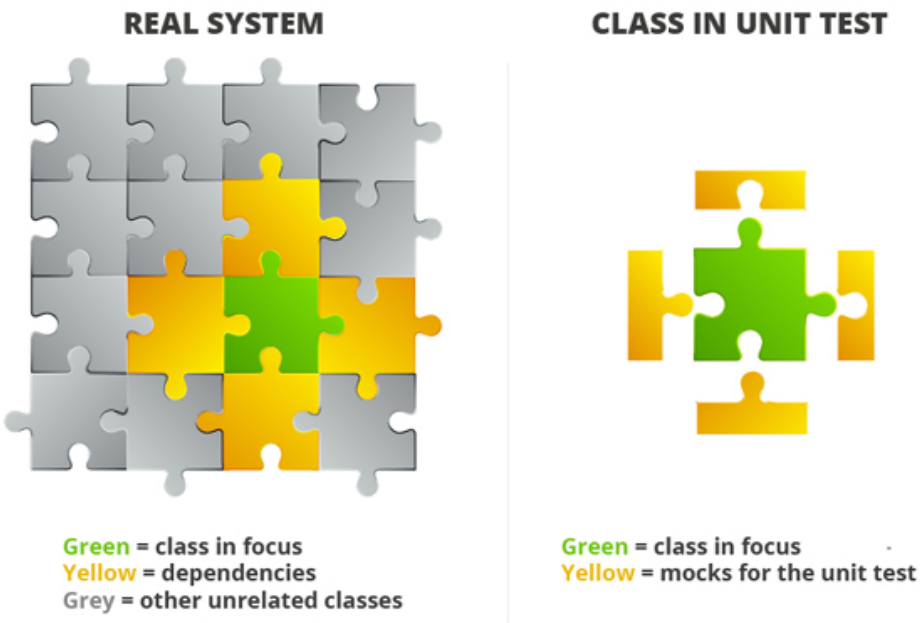
By: Usually programmers.

- Units may be:
  - Individual functions or methods within a class/object
  - Object classes with several attributes and methods
  - Composite components (consisting of several classes) with defined interfaces used to access their functionality.
- Complete test coverage of a class involves
  - Testing all operations/methods associated with an object
  - Setting and interrogating all object attributes
  - Exercising the object in all its possible states.
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised

Unit test – test code in isolation…



- ”In isolation” means that dependencies must be broken. This can be done through the use of so called ”stubs” (or ”mocks”).
- The stub contains the same interface as the dependent module, but no functionalty. (the stub may provide fixed answers to requests, or use ”null” as a response)



Unit test – techniques to be used

- “Testing all operations/methods” is easy, since this is a finite set.
- “Setting and interrogating all object attributes” is more tricky, since we must decide which values to use!
- “Exercising the object in all its possible states” adds a restriction to which values we can choose for our tests.
- So, choosing input values in the key to successful testing!
- Let’s look at some techniques to find good input values

Method: Equivalence Partitioning (EP)

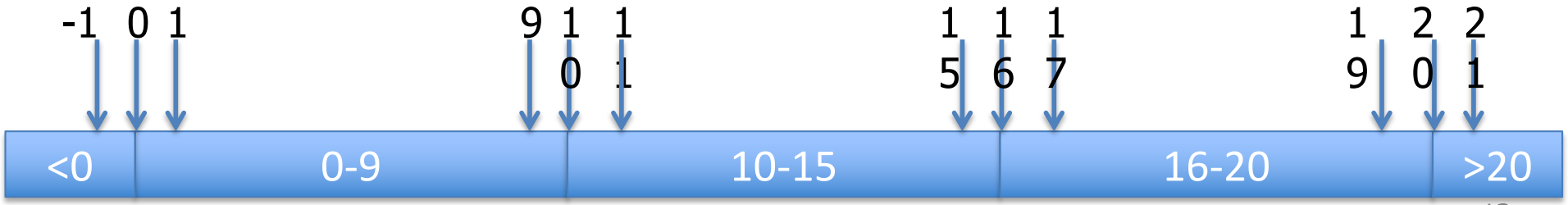
- ”Equivalence partitioning is based on the premise that the inputs and outputs of a component can be partitioned into classes that, according to the component's specification, will be treated similarly by the component. Thus the result of testing a single value from an equivalence partition is considered representative of the complete partition”. (from BS7925-2*)
- Example: student exam having the following limits: below 10p = U, 10-15 =G and 16 and above =VG (max being 20. Only whole points given)

- Choose on value from each “group”: e.g. 5, 12 and 18
- However, don’t forget the “invalid” group! We also need to consider <0, >20 and non-integers!
- So, a good test set could be:
  - **valid values 5, 12, 18** **Should result in valid output**
  - **Invalid values -1, 22, 3.14, “hello”** **Should yield “fault message”**

* http://www.testingstandards.co.uk/Component%20Testing.pdf

Method: Boundary Value Analysis (BVA)

- Faults tend to lurk near boundaries – a good place to look for faults.
- (from BS7925-2): BVA is based on Equivalence Partitioning and, secondly that developers are prone to making errors in their treatment of the boundaries of these classes.
- In our case from last slide, we would get:
  - valid values 0, 1, 9, 10, 11, 15, 16, 17, 19, 20 (on border and either side)
  - Invalid values -1, 21
  - Remaining from EP: 3.14, “hello”
  - Note that EP valid values 5, 12 and 18 are now not needed - replaced by 0/1/9, 10/11/15 and 16/17/19/20 respectively

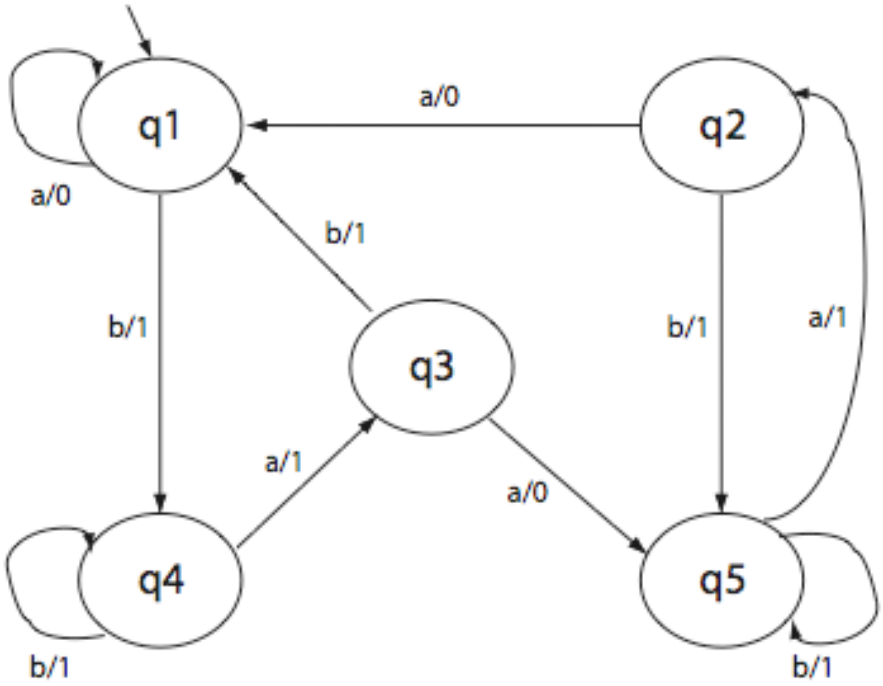


Boundary/Border testing

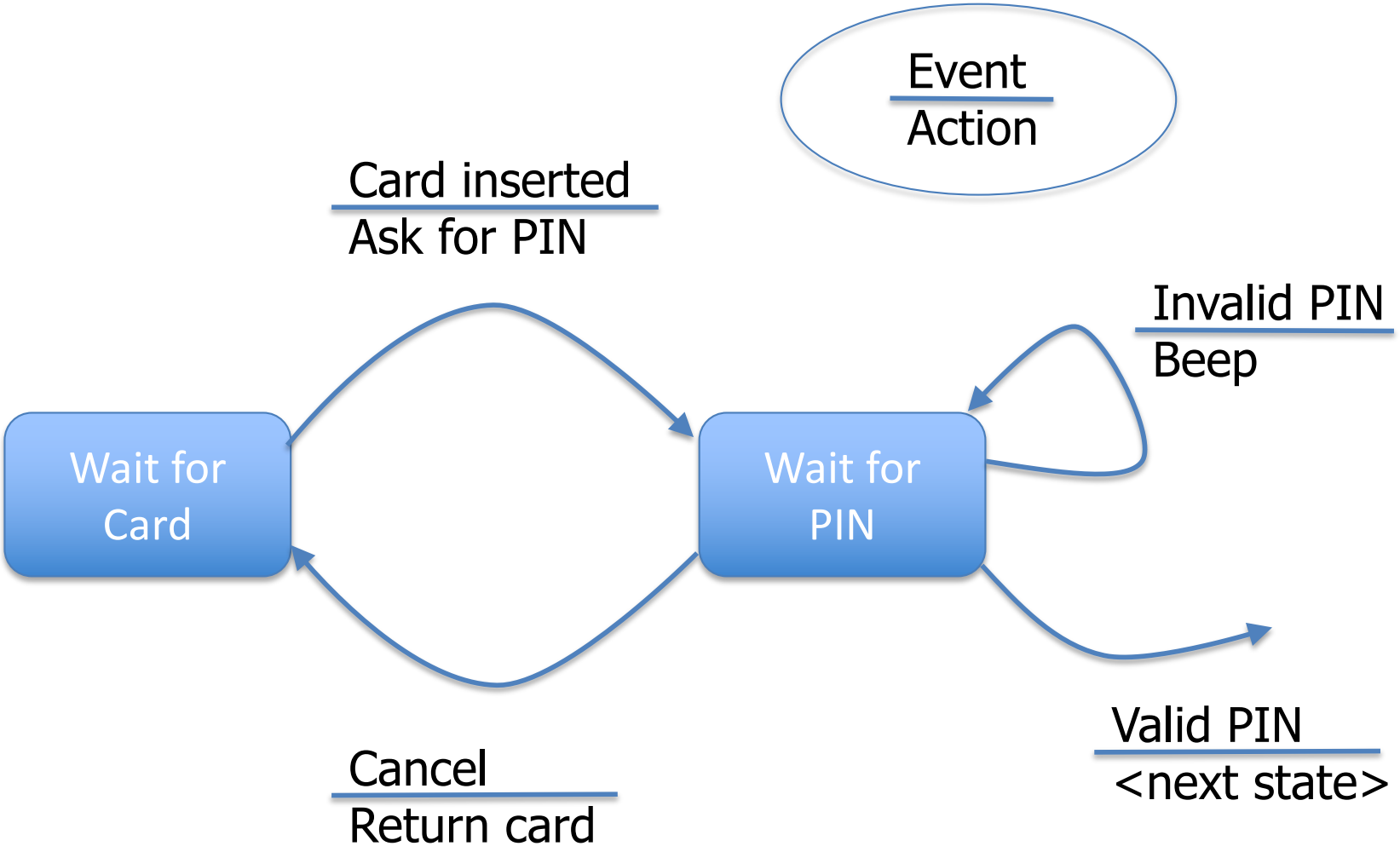
- **boundary value analysis**: Testing conditions on bounds between classes of inputs. (I often call it “Border Value Analysis”)
  - If number of test cases need to be minimised, use 2 test cases/border (Below/on border – above/on border, depending on how border is defined. In my example U= value **<** 10, G= value **>=** 10)
- Why is it useful to test near boundaries?
  - likely source of programmer errors (< vs. <=, etc.)
  - language has many ways to implement boundary checking
  - requirement specs may be fuzzy about behavior on boundaries
  - often uncovers internal hidden limits in code
    - example: array list must resize its internal array when it fills capacity

State-Based Testing

- Model functional behavior in a state machine (e.g. a communication – protocol …)
- Select test cases in order to cover the graph
  - Each node
  - Each transition
  - Each pair of transitions
  - Each chain of transitions of length *n*

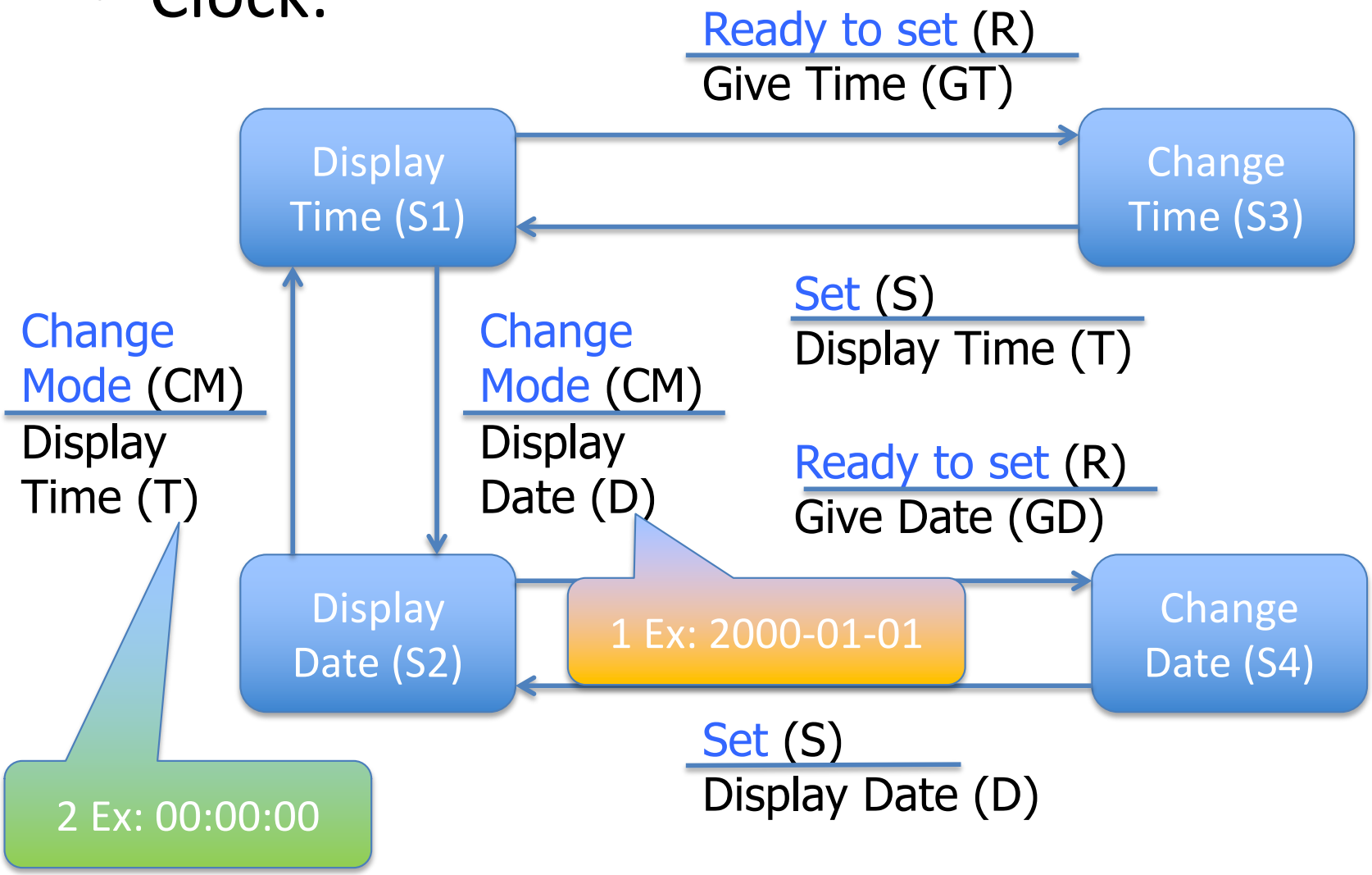


State Transition Testing - model



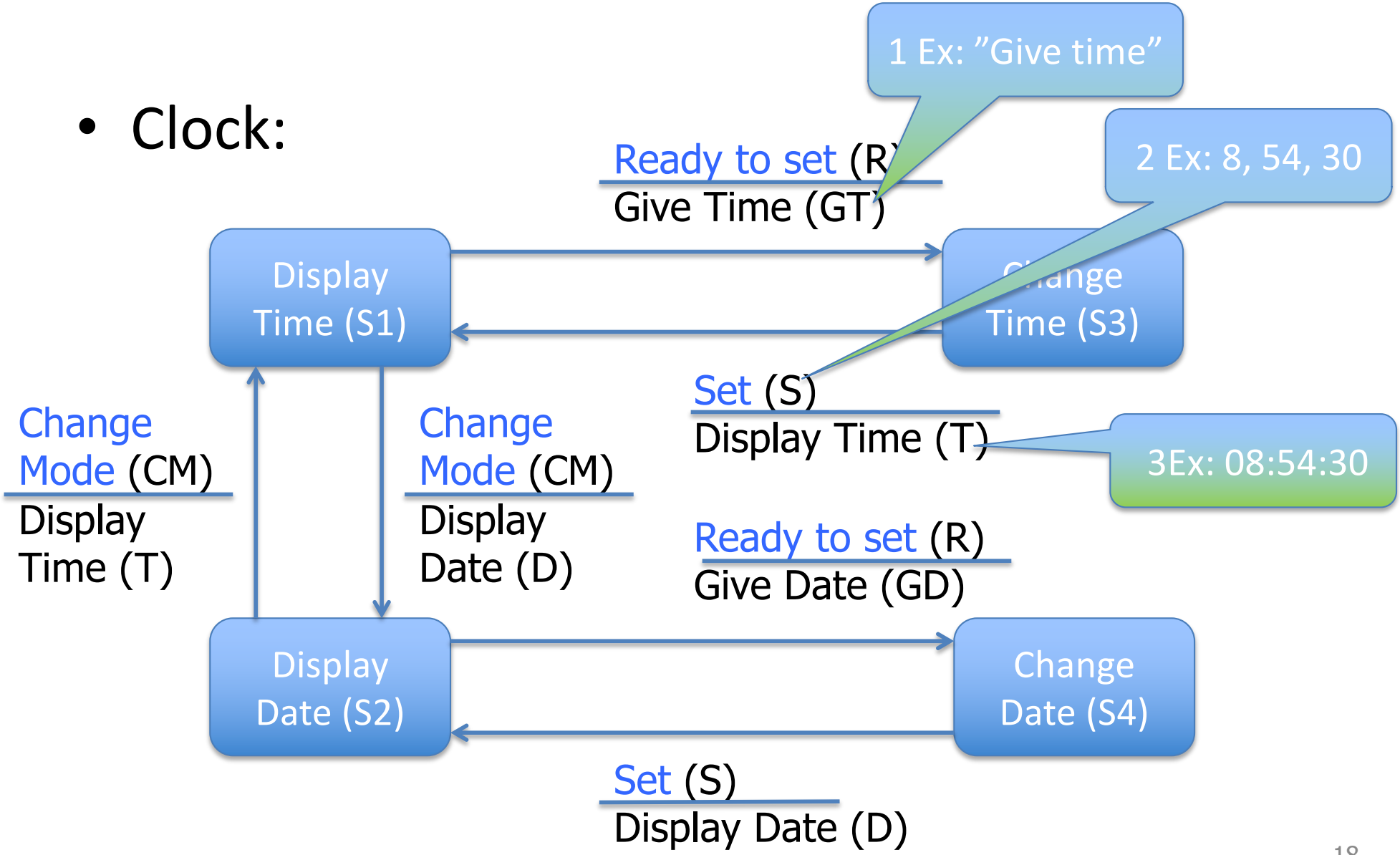
State Transition Testing - example

- Clock:



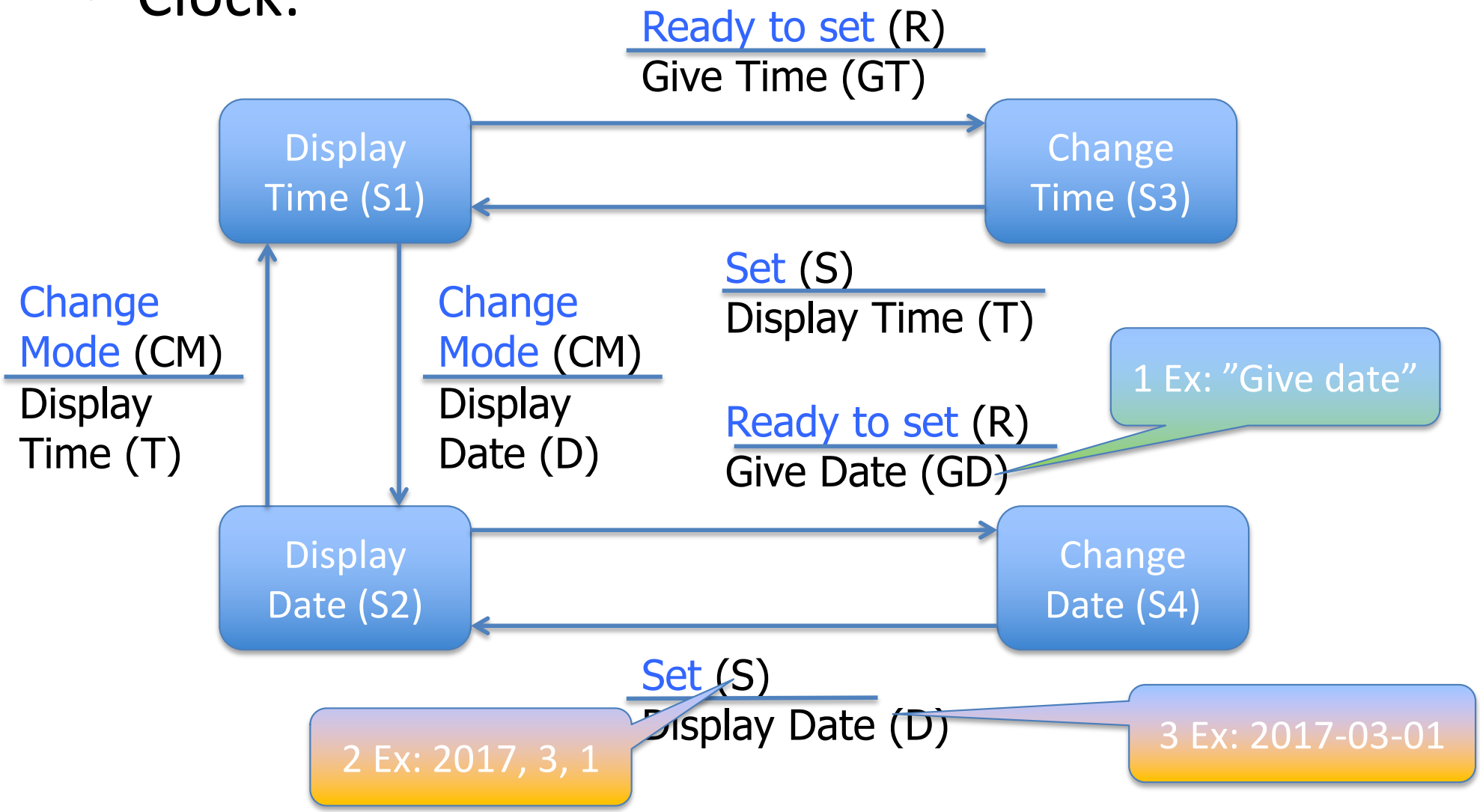
State Transition Testing - example

- Clock:

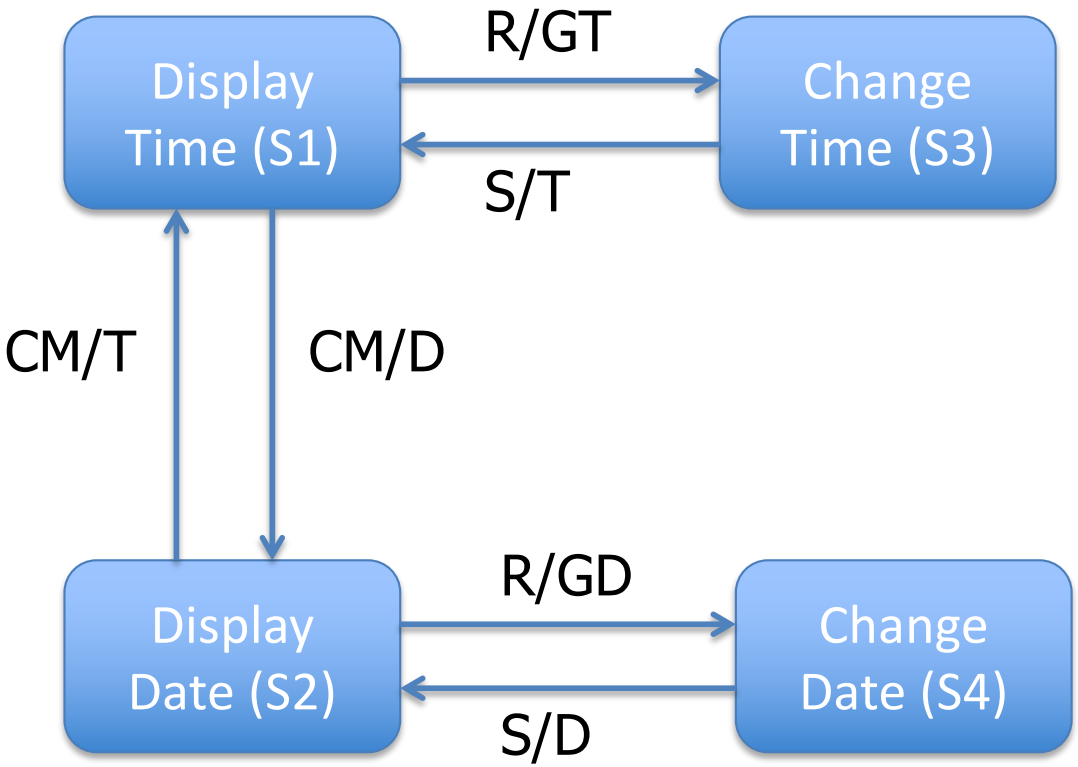


State Transition Testing - example

- Clock:



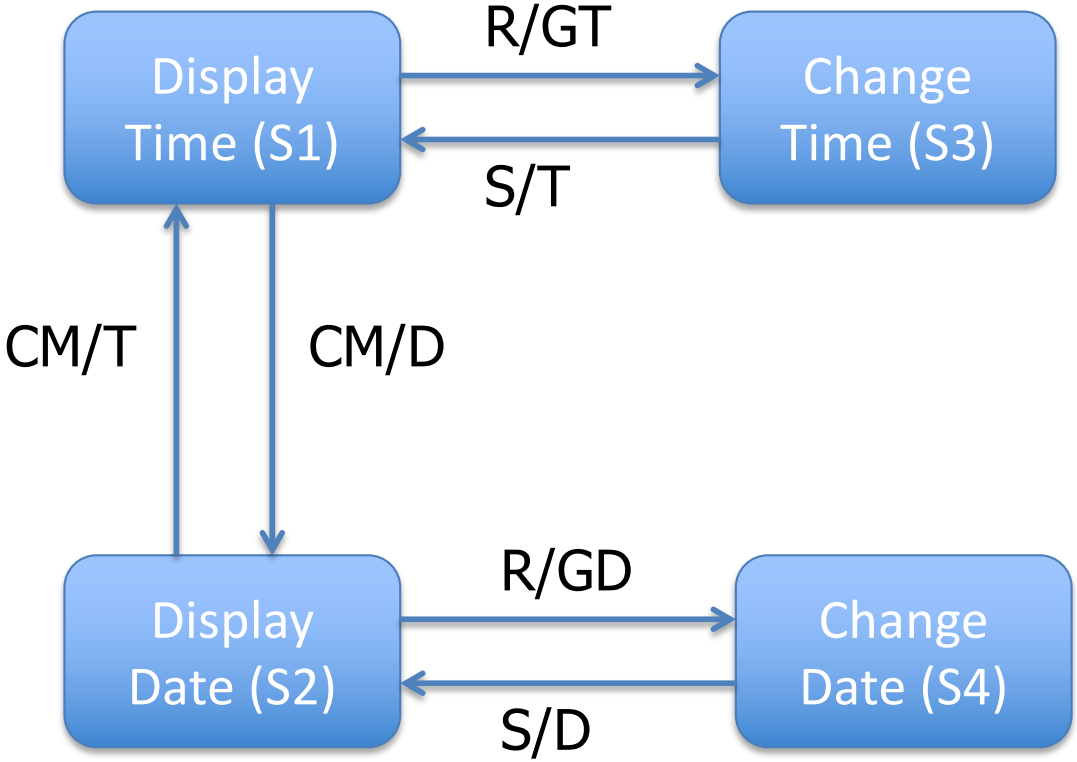
Possible Transitions



Transition	Start state	Event Action	End state
1	S1	CM/D	S2
2	S2	CM/T	S1
3	S1	R/GT	S3
4	S3	S/T	S1
5	S2	R/GD	S4
6	S4	S/D	S2

Note! There are only 3 possible events

Test case for Transition Coverage (0-switch)



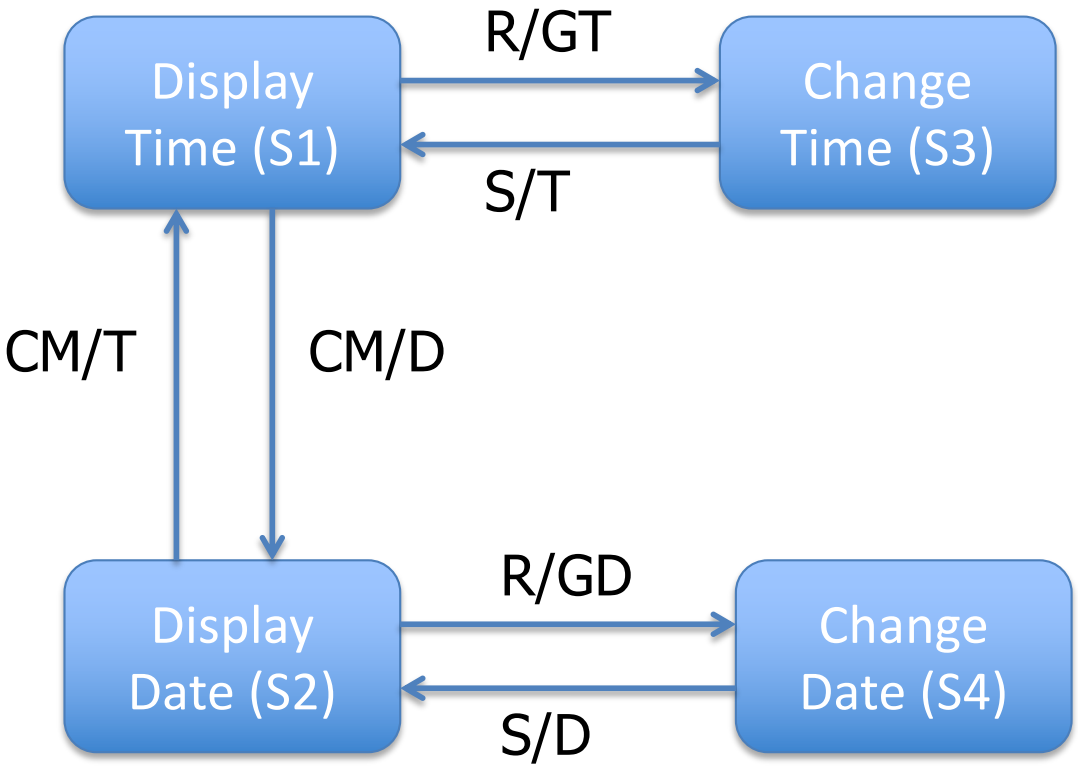
…and expect this to happen

Test Case	Start state	Event	Action/ New state
1	S1	Ready	Chg T/ (S3)
2	S3	Set	Displ T/ (S1)
3	S1	CM	Displ D/ (S2)
4	S2	Ready	Chg D/ (S4)
5	S4	Set	Displ D/ (S2)
6	S2	CM	Displ T/ (S1)

Transitions: 1 2 3 4 5 6

(Assume start state is S1)

Test case for Transition Pairs (1-switch)



Test Case	Start State	Next State	End State
1	S1	S2	S1
2	S1	S2	S4
3	S1	S3	S1
4	S3	S1	S2
5	S3	S1	S3
6	S2	S1	S2
7	S2	S1	S3
8	S2	S4	S2
9	S4	S2	S4
10	S4	S2	S1

= covered by first test case

(order: in a certain order, or make then independent)

State table for valid/invalid transitions

	CM	R	S
S1	S2/D	S3/GT	**S1/N**
S2	S1/T	S4/GD	**S2/N**
S3	**S3/N**	**S3/N**	S1/T
S4	**S4/N**	**S4/N**	S2/D

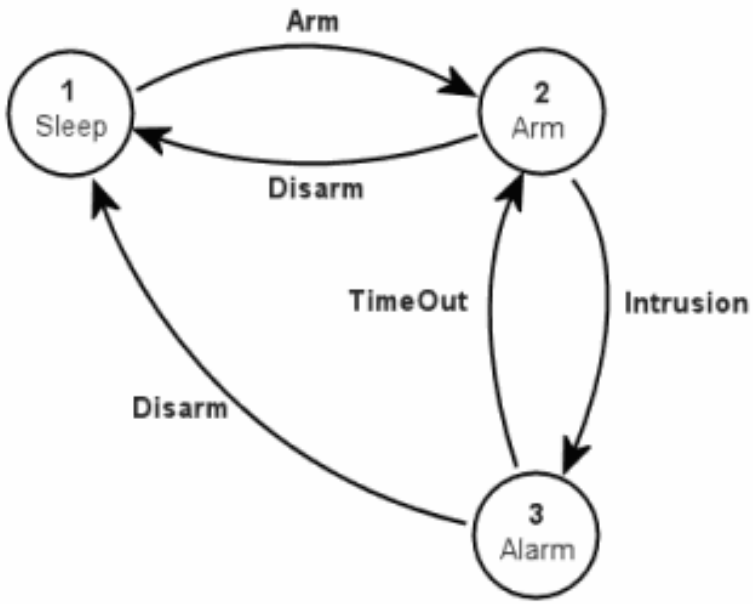
State tables are common in industry! (easier to create than drawings…)
Note: invalid input does not change state. Invalid combinations are easier to see in this representation!

Strategy when testing state transitions:

- Confirm correct results (particularly where transitions use same action)
- Start by testing most likely transitions
- Add more complex tests to cover exceptional conditions (including testing invalid transitions)

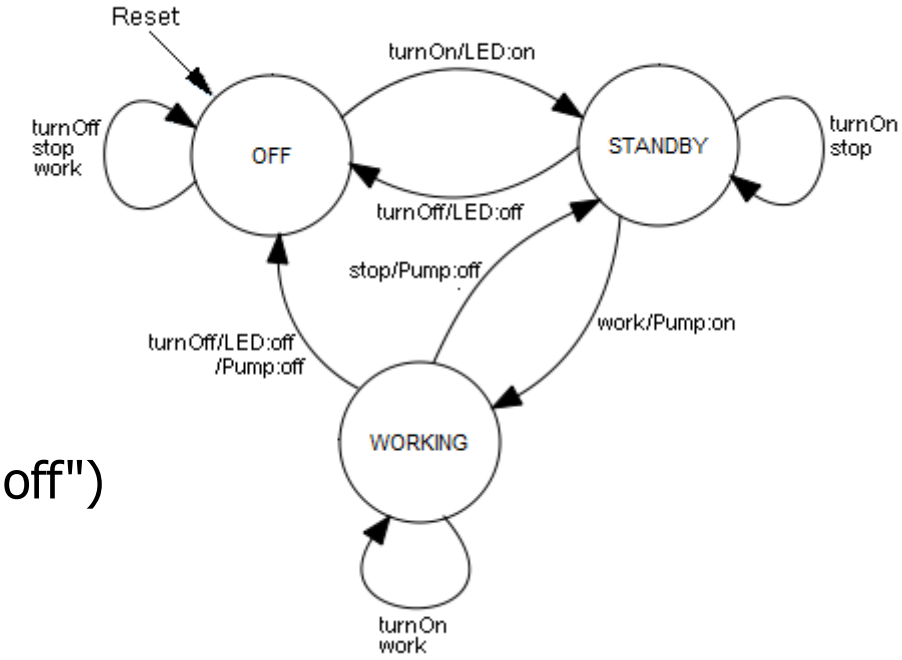
Example: State machine in C (Moore)

```
typedef enum {SLEEP, ARM, ALARM} states;
while (1) {
          Switch (current_state) {
          Case SLEEP:
                    siren (OFF);
                    If (arm_system()) {
                    Next_state = ARM;
                    }
                    arm_indicator (OFF);
                    break;
          Case ARM:
                    arm_indicator (ON);
                    siren (OFF);
                    If (!arm_system()) {
                              Next_state = SLEEP;
                    } else if (intrusion ()) {
                              next_state = ALARM;
                              StartTimer();
                    }
                    break;
          Case ALARM:
                    siren (ON);
                    arm_indicator (OFF);
                    If (!arm_system()) {
                              Next_state = SLEEP;
                    } else if (timeOut()) {
                              next_state = ARM;
                    }
                    break;
          }
          current_state = next_state;
}
```



Example: State machine (Mealy)

```
current_state ="OFF"; // init
While(1): switch (event):
Case "turnOff":
        if (current_state =="STANDBY")
                new_state ="OFF";
                action ("LED off");
        if (current_state =="WORKING")
                new_state ="OFF"
                 action ("LED and pump off")
Case "turnOn":
        if (current_state =="OFF")
                new_state ="STANDBY"
                 action ("LED enabled")
Case "stop":
        if (current_state =="WORKING")
                new_state ="STANDBY"
                 action ("Pump off")
Case "work":
        if (current_state =="STANDBY")
                new_state ="WORKING"
                 action ("Pump enabled")
```



Code is useful for the lab, but note that this FSM is not completely specified....

Conformance testing

- Verifying that an implementation fulfils requirements (e.g. of a standard)
- The generic model* for conformance tests follows a 3 step model:

1. The FSM is put in state S_i
2. Input A_k is applied and the output is checked to verify that it is O_l
3. The new state of the FSP implementation is checked to verify that it is S_j, as expected.

*Aho et. Al. “An Optimization Technique for Protocol Conformance Test Generation Based on UIO Sequences and Rural Chinese Postman Tours”

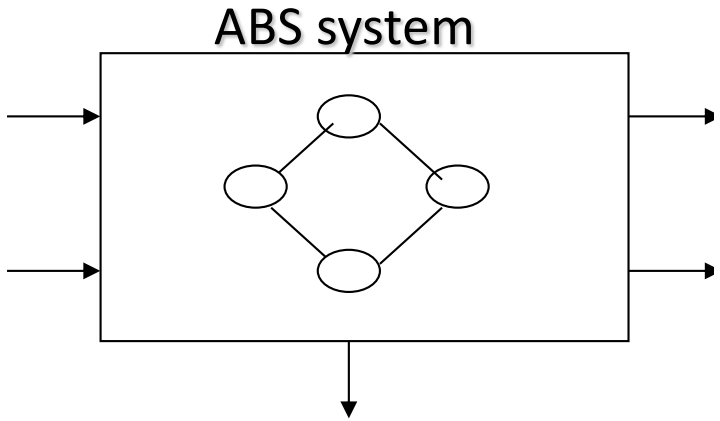
Conformance testing

- However, due to the *Controllability* of the FSM, it may not be possible to put the implementation in a specific state
- Due to the *Observability* of the FSM, it may not be possible to directly verify that the FSM implementation is in state S_j.

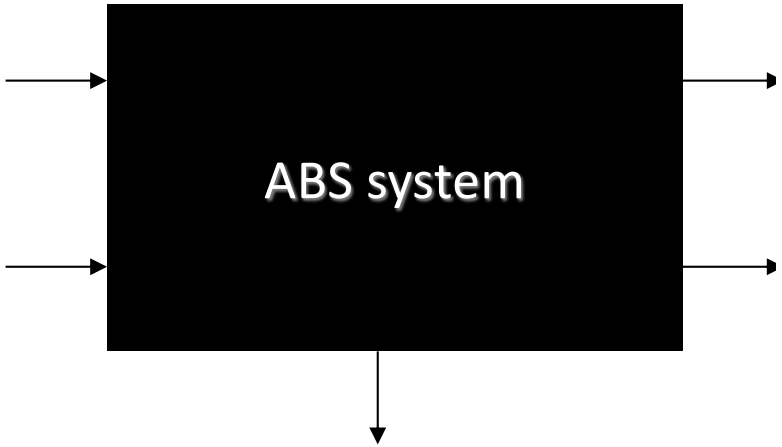
- In the Lab, where you shall test the clock described in previous slides, this can be done by adding a few methods to improve Controllability and Observability. Some students try this, but it is not allowed!
- The limitations in the assignment forces you to ”work around” this problem, by “stepping” through the SFM by giving suitable input and watching the (hopefully unique) output.

Recall - Accessibility

- Structural testing (white-box, glass-box)
  - Uses code/detailed design to develop test cases
  - Typically used in unit testing



- Functional testing (black-box)
  - Uses function specifications to develop test cases
  - Typically used in system testing

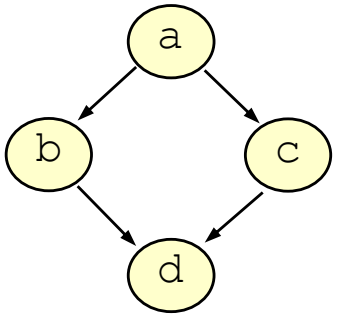


White-box Testing

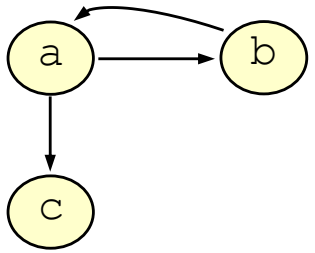
Methods based on internal structure of code

- Approaches:
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Condition coverage (NEW!)

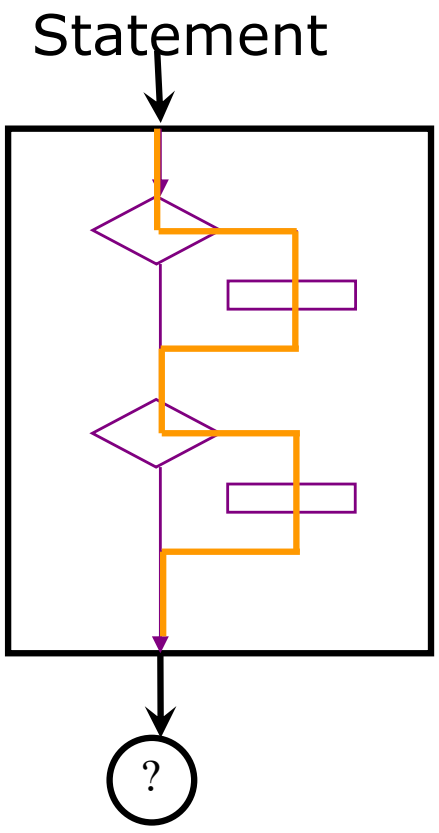
```
if a then
    b
else
    c;
d
```



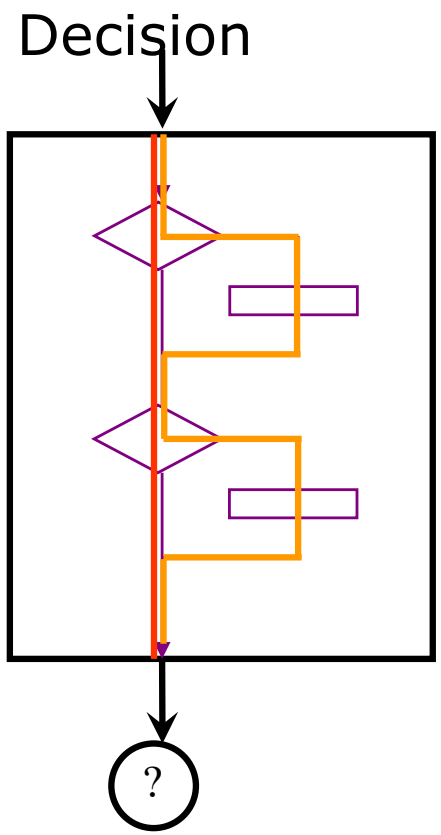

```
while a do
    b;
c
```



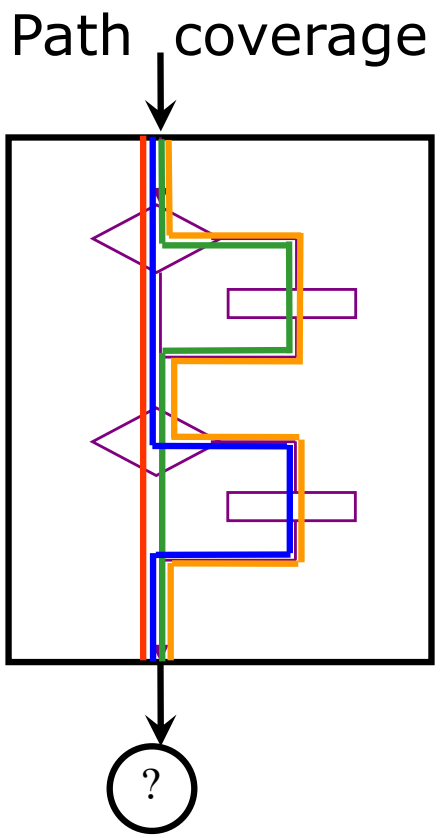
Test/Code coverage – alternative presentation



1

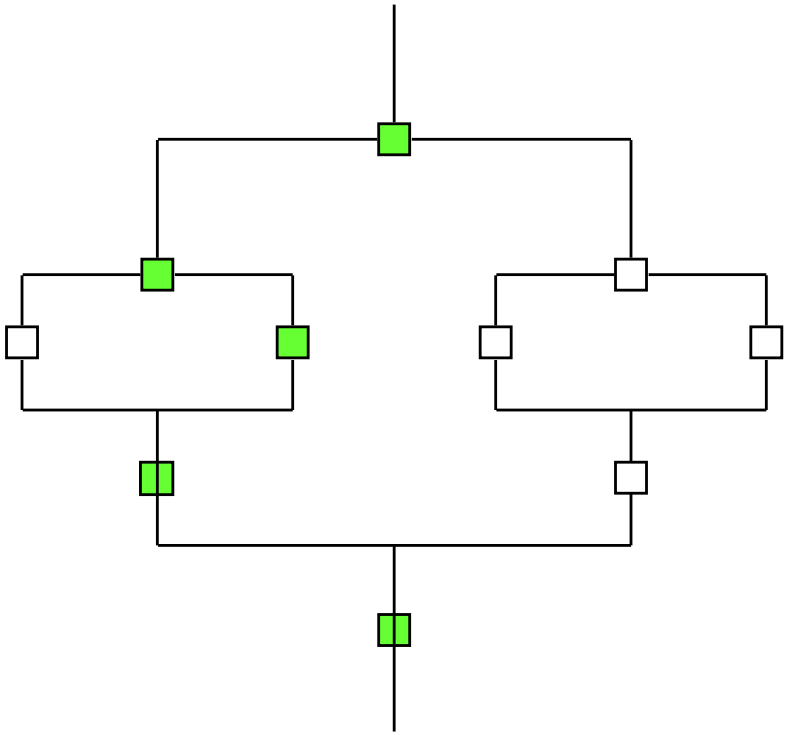


2



4 test cases

Test Coverage–Example



Statement Coverage: 5/10 = 50%

Branch Coverage: 2/6 = 33%

Path Coverage: 1/4 = 25%

We would expect statements which are associated with machine code to be regarded as executable. E.g.:

- Assignments;
- loops and selections;
- procedure and function calls;
- variable declarations with explicit initializations;
- dynamic allocation of variable storage on a heap.

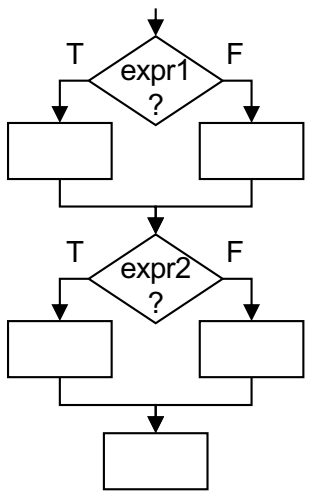
Cyclomatic Complexity – Coverage Testing

Cyclomatic Complexity can be used as an indication of the needed number of test cases to fully cover a piece of code.

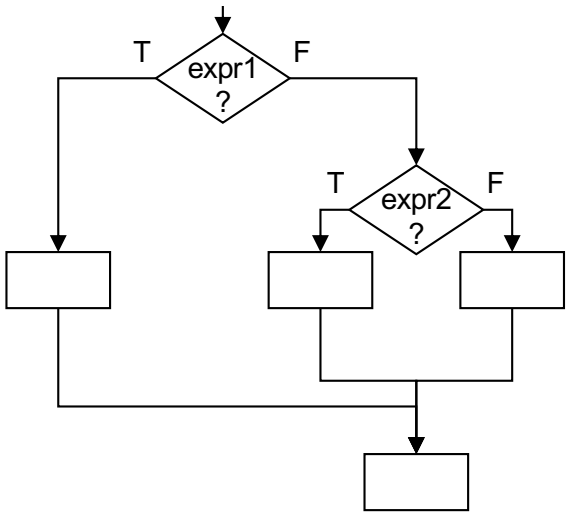
As can be seen from the figure below (CC = 3 in both cases), the number cannot always be used directly to determine the number of test cases. Instead, the following applies:

Branch Coverage <= Cyclomatic Complexity <= Number of Paths

CC is an “Upper Bound” for Branch Coverage (BC) and a “Lower Bound” for Path Coverage (PC)



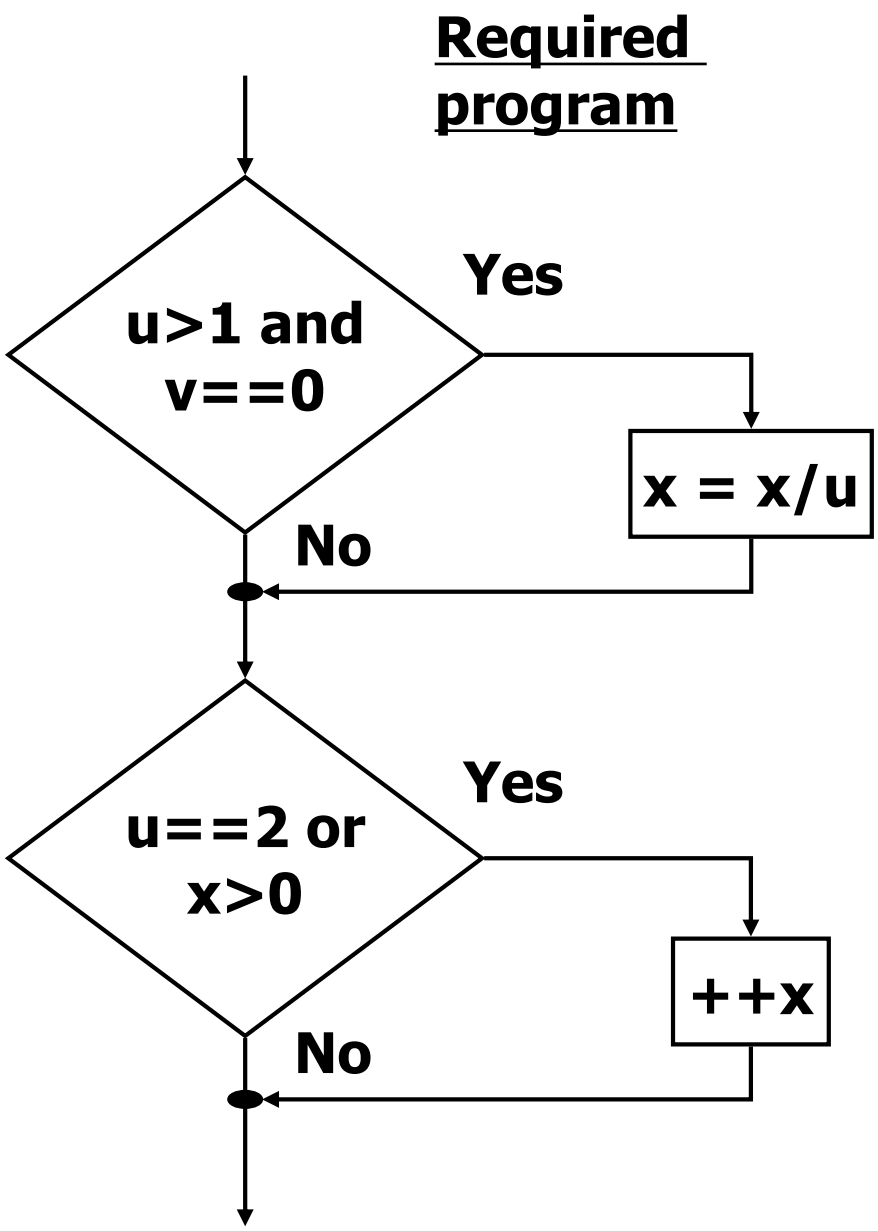
= CC =



BC = 2, PC = 4

BC = 3, PC = 3

Covering Every Statement is Not Sufficient (Myers)



Code attempt to implement flowchart

```
if( (u>1) && (v==0) )          (1)
        x = x/u;               (2)
if( (u==2) || (x>3) )          (3)
        ++x;                   (4)
```

Test data: u=2, v=0 and x=3

- executes every line (1) - (4)
- gives the correct output x= 2.5

However, line (3) is wrong
We thus need "Condition coverage"!

Test Coverage - Condition

- Apart from the three coverage types mentioned above (Statement, Branch and Path) – a fourth is sometimes mentioned.
- Condition coverage looks at complex conditions:
- "If (A>0)" is a simple condition that can be tested with 2 test cases (e.g. A=0 and A=1).
- How about: "if ((A>2) && (B<0))"?
- To cover all conditions, we need to make sure that "each Boolean sub-expression has been evaluated both to true and false at least once".
- Now we need 4 tests: (e.g. False False{A=2, B=1}, TF{A=3, B=1}, FT{A=2, B=-1}, TT{A=3, B=-1}) Note that only in the last case, the statements in the IF are executed…

Test Coverage – MC/DC
Modified condition/decision coverage

- Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, and each condition has been shown to affect that decision outcome independently.
  - A condition is shown to affect a decision's outcome independently by varying just that condition while holding fixed all other possible conditions.
  - The condition/decision criterion does not guarantee the coverage of all conditions in the module because in many tests, some conditions of a decision are masked by the other conditions.
  - Using the modified condition/decision criterion, each condition must be shown to be able to act on the decision outcome by itself, everything else being held fixed.
- The MC/DC criterion is thus much stronger than the condition/decision coverage

E.g. required for avionic software covered by the DO-178B standard of the Federal Aviation Administration.

Coverage-based Testing

- Advantages
  - Systematic way to develop test cases
  - Measurable results (the coverage)
  - Extensive tool support
    - Flow graph generators
    - Test data generators
    - Bookkeeping
    - Documentation support
- Disadvantages
  - Code must be available

Work process – coverage-based testing

1. Start by identifying what should be tested (and how/which test data to use)
2. Consider applying TDD för the first stage, creating the test cases BEFORE writing the production code.
3. Run code-coverage measurements and iterate:
   - Measure coverage
   - If not enough coverage*, add test cases that test previously uncovered parts of the code

*Industry requirement may be 85%

What Can't White Box Tests Find?

As essential as it is to have well-tested units and builds, white box testing can't find every kind of bug

- Bugs of omission
  - Missing requirements, design elements, code
    - Use requirements, design and code reviews
- Unmaintainable code
  - Inscrutable variable names or constructs, etc.
    - Use code reviews and static analysis tools
- Systemic problems
  - Performance, security, usability, interoperability, etc.
    - Use code reviews, black-box system testing, beta/pilot tests, usability studies, prototyping, etc.

Automated testing

- Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- With automated tests, the test suite can be rerun whenever there has been a change to the SW. One can then see if the new code has made any of the previously passing test cases to now fail.
- In automated unit testing, you make use of a test automation framework (such as Junit/NUnit) to write and run your program tests.
- 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 of otherwise of the tests.

In the lab, you will create test cases in JUnit for a given piece of code. You will also measure code coverage, based on your tests.

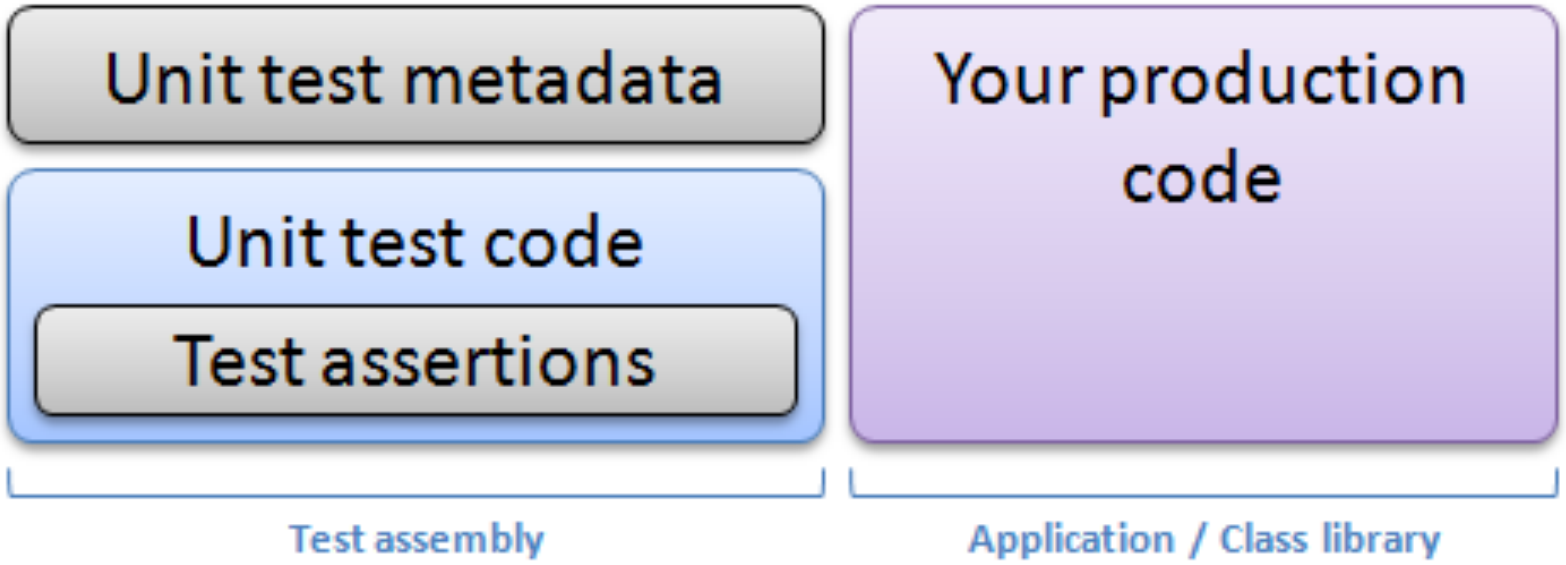
Test Framework – for automated testing

JUnit is a **Regression Testing Framework.** It contains:

1. **Fixtures** - a fixed state of a set of objects used as a baseline for running tests. The purpose of a test fixture is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable. Examples:
   - setUp() method which runs before every test invocation.
   - tearDown() method which runs after every test method.
2. **Test suites** – written by the developer….
3. **Test runners** - used for executing the test cases.
4. **JUnit classes** - used in writing and testing JUnits. Examples:
   - **Assert** which contain a set of assert methods.

http://www.tutorialspoint.com/junit/junit_test_framework.htm

Test framework – relation to production code



- Note! The test code must be separated from the production code, so that this can be delivered without the test code.
- All dependencies must thus be from test code to production code – never the opposite!

Automated test components

The test cases contains the following 4 steps “The Four-Phase Test Pattern”*:

1. A setup part, where you initialize the tested system and establish the preconditions to the test. This may include creating instances of tested classes and data.
2. A call part, where you do something to the system, e.g. call a method to be tested.
3. An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful. If false, then it has failed.
4. Cleanup, where you return the system under test to its initial state.

** from: Grenning: Test-Driven Development for Embedded C (originates in Meszaro: xUnit Testing Patterns)*

xUnit / JUnit

- Verification is usually done using the `assert_*_()` methods that define the expected state and raise errors if the actual state differs
- http://www.junit.org/
- Examples:
  - `assertTrue(4 == add(2,2));`
  - `assertEquals(expected, actual);`
  - `assertNull(Object object);`
  - etc.

Example Test Case 1

Listing 2-1: Example test case for the Key Checker class.

```
public class CheckerTest {
  // test case to check that invalid key is rejected
  @Test
      public void checkKey_anyState_invalidKeyRejected() {

    // 1. set up
    Checker checker = new Checker( /* constructor params */ );

    // 2. act
    Key invalidTestKey = new Key( /* setup with invalid code */ );
    boolean result = checker.checkKey(invalidTestKey);

    // 3. verify
    assertEqual(result, false);
  }
}
```

Example Test Case 2

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class MyTests {
   @Test
   public void multiplicationOfZeroIntegersShouldReturnZero() {
   MyClass tester = new MyClass(); // MyClass is tested
                                   // assert statements
   assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
   }
}
```

```
// Only ONE assert per test case, since execution is aborted when an
assert fails!
```

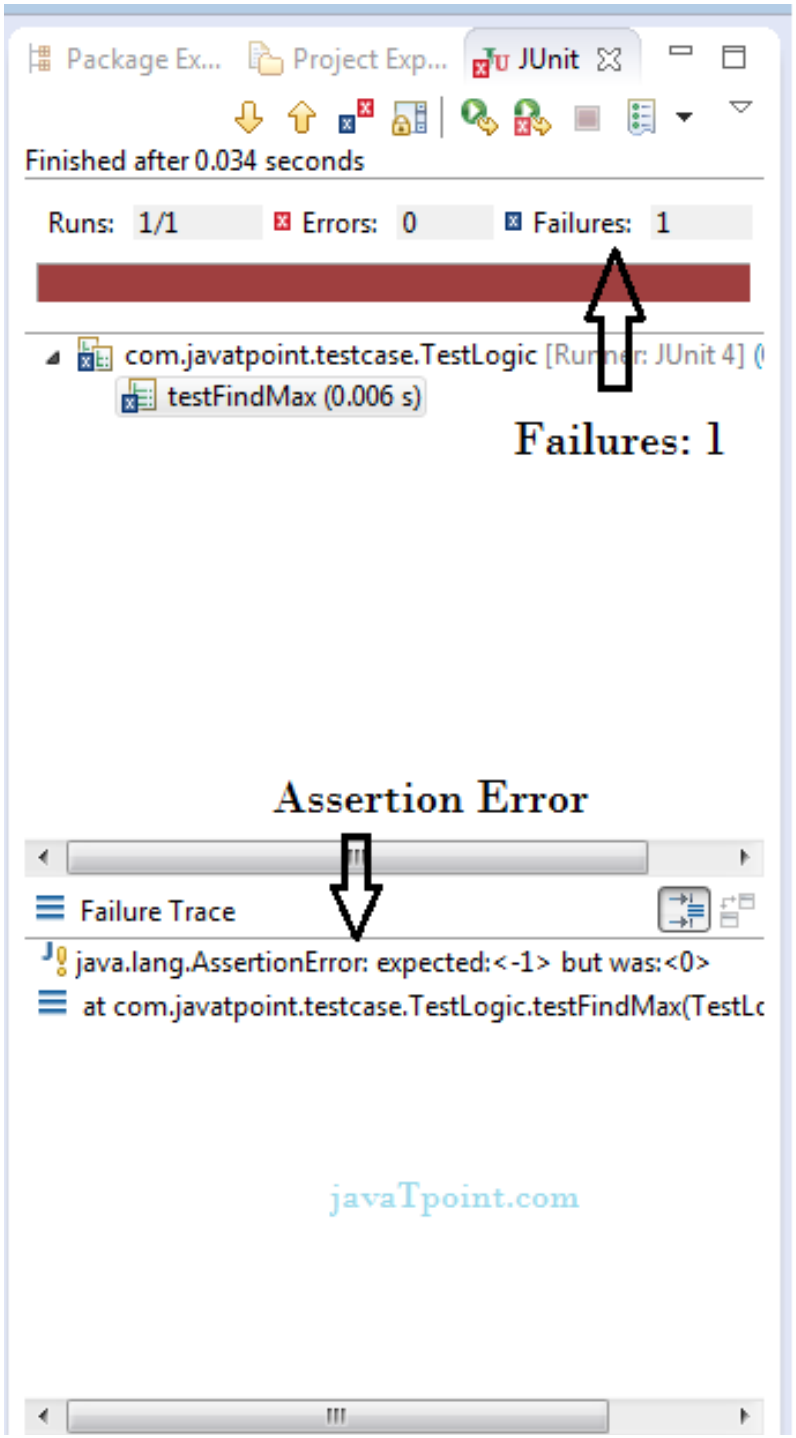
“production code – to be tested”

```
package com.javatpoint.logic;
public class Calc {

    public static int findMax(int arr[]){
        int max=0;
        for(int i=1;i<arr.length;i++){
            if(max<arr[i])
                max=arr[i];
        }
        return max;
    }
}
```

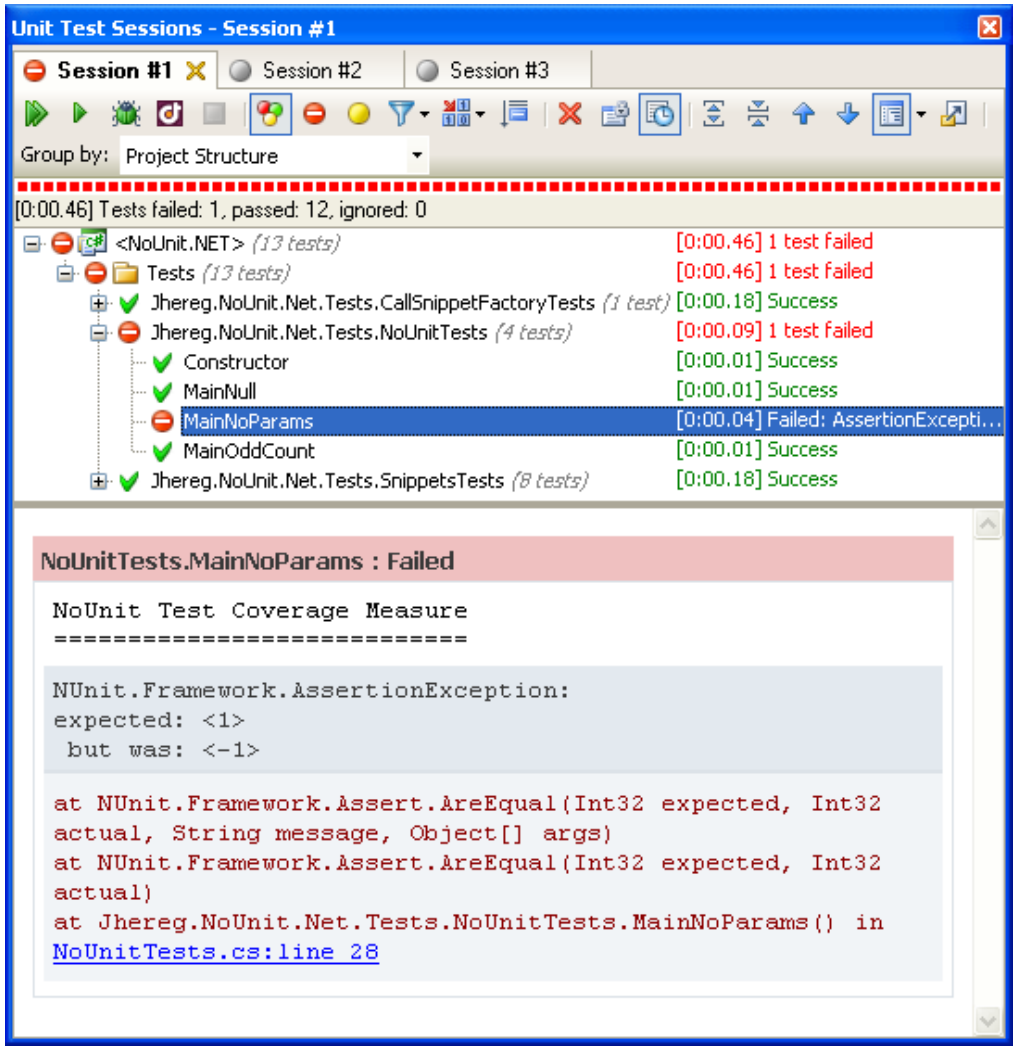
Example Test Case 3

```
package com.javatpoint.testcase;

import static org.junit.Assert.*;
import com.javatpoint.logic.*;
import org.junit.Test;

public class TestLogic {

   @Test
   public void testFindMax1(){
      assertEquals(4,Calc.findMax(new int[]{1,3,4,2}));
   }
   @Test
   public void testFindMax2(){
      assertEquals(-1,Calc.findMax(new int[]{-12,-1,-3,-4}));
   }

}
```

JUnit result….
…running the test on previous slides.



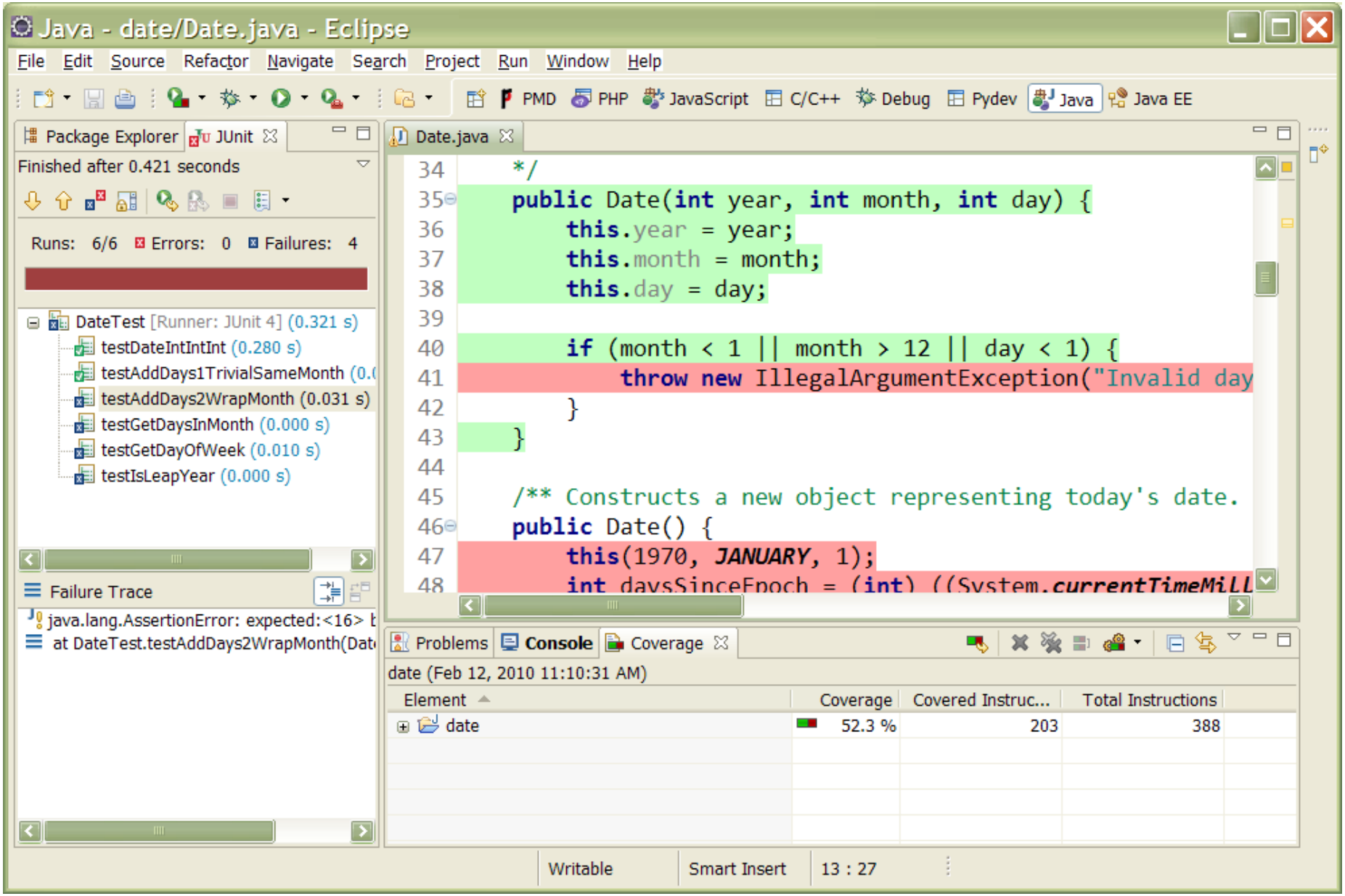
Test Runners - example



A good test runner system provides a good overview of:

- What was executed
- The status of each test
- An explanation of any problems that was found

Code coverage example – JUnit/Emma



Unit test effectiveness

- The test cases should show that, when used as expected, 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.
- This leads to 2 types of unit test case:
  - The first of these should reflect normal operation of a program and should show that the component works as expected.
  - The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

Test-Driven Development

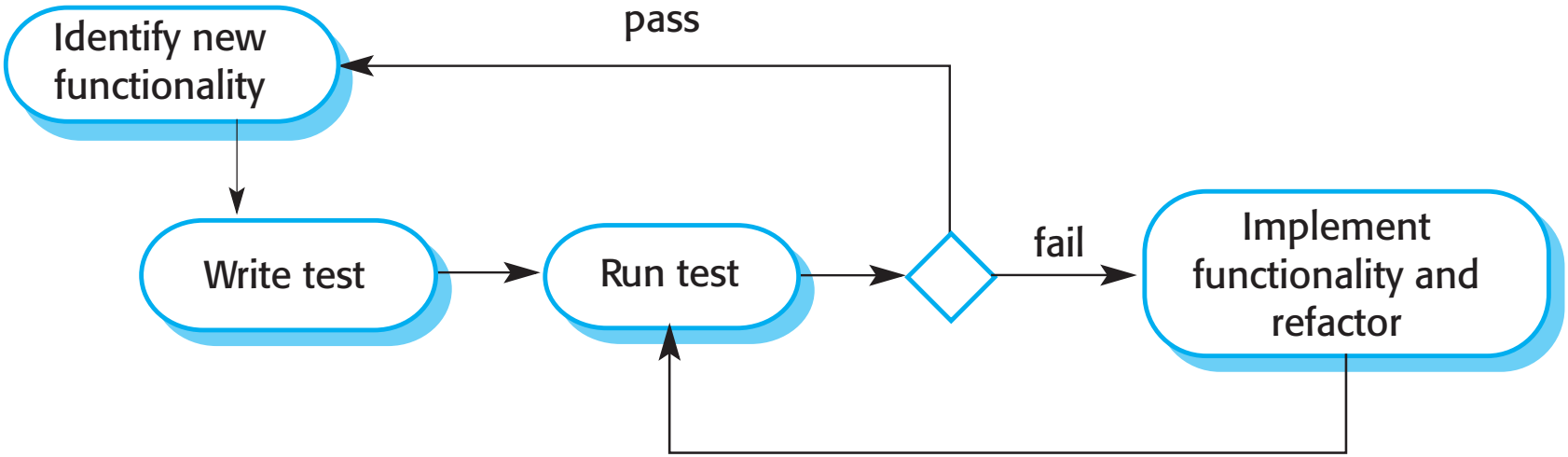
- Test-Driven Development (TDD) is an approach to program development in which you inter-leave testing and code development.
- Tests are written **before code** and 'passing' the tests is the critical driver of development.
- You develop code incrementally, along with a test for that increment. You don't move on to the next increment until the code that you have developed passes its test.

In the lab, you will try out TDD when implementing a set of methods (following a given specification)

TDD process activities

- Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- Write a test for this functionality and implement this as an automated test.
- Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- Implement the functionality and re-run the test.
- Once all tests run successfully, you move on to implementing the next chunk of functionality.

Test-driven development



Code refactoring is a "disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour", undertaken in order to improve some of the non-functional attributes of the software. Advantages include improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility.

Benefits of test-driven development

- Writing the test first, forces you to consider what the new code shall do. This often leads to better structured code (remember the "single responsibility principle")
- 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 problem lies. The newly written code needs to be checked and modified.
- System documentation
  - The tests themselves are a form of documentation that describe what the code should be doing.

Key things to remember

- Choosing good input values in the key to successful testing!
- Border Value Analysis (BVA) is a good technique to start test design with. (Often overlaps EP)
- State transition testing is good for SW that uses state.
- There are different types of Code Coverage. Different tools may measure different things. Find out what your tool measures!
- TDD is a good practice when developing code!

Extra slides

Extra: How to create a good unit test

- SOLID:
  - S: Single Responsibility principle.
  - O:Open-Closed principle.
  - L: Liskov substitution principle.
  - I: Interface Segregation principle.
  - D: Dependency Inversion principle.

  SOLID principles of object-oriented design are an important consideration for anyone looking for good software design. The problem is they can be hard to understand and implement. In my personal experience unit-testing can actually help understanding these principles better and provide a genuine use case for implementing them

Dependency Inversion

- A dependency can be anything like a database, the system clock, a web-service, or even a method in a different class. Unit-testing code presents interesting challenges and forces you to think about how you can isolate the bit you want to test from other code it interacts with-- its dependencies. In order to successfully unit-test your classes, you will need to be able to move dependencies from your code to one place, usually the constructor, and make them replaceable. The idea behind this is that then in your test harness you can substitute away those dependencies for dummy values or dummy providers and invoke code under test without hitting the real external dependency. So in order to isolate your code under test from other code that effects its behaviour, you are forced to firstly think about what other dependencies it has and isolate them and secondly think how those dependencies can be replaced during unit-testing. What you are doing is Dependency Inversion, and it is much easier to do Dependency Inversion when thinking about how you can unit test your code rather doing it just because you read it in a book.

Interface Segregation

Once you have identified and isolated dependencies, the next thing you have to do in your test code is to somehow mock/stub/fake the dependencies, so you can substitute them in your code under test in place of the real dependency. This is usually a challenging part of unit testing and ideally you want to mock as little as possible, enough so that the code under test will run correctly. If you find yourself needing to mock a large interface or a complex interaction of methods you should think about the design of those dependencies and about splitting those interfaces or interactions into smaller chunks, so that you can mock only the chunk that concerns your code under test right now. What you are indirectly forced to do is to think about the interfaces provided by the dependencies and how your code under test uses them, then split each interface so that it makes logical sense to mock it for your code under test. That is Interface Segregation for your dependencies. Not only are you thinking about improving the code under test, you are also improving interfaces provided by its external dependencies, so that they are easier to mock. It is far easier to design interfaces when you have a client using it, then to imagine a design for an interface your client may probably want to use. A badly designed interface will make mocking difficult or will need several different mocks depending on how it is being used.

Single Responsibility

Most test harnesses allow a setup and teardown operation where you setup the dependencies before actual testing begins and cleanup afterwards. If you find yourself filling up these methods with lots of unrelated dependencies, for example, you find that your class depends on the database, logger, email service, soap client interface, and everything else under the sun and you find you are spending a lot of time setting all of these up in your test and its giving you a headache, this is a clear warning that your code under test is actually violating the Single Responsibility principle. When your code violates this principle, your class loses focus and starts doing several often unrelated operations. Should your business entity class represent just the entity? should it also save and load it from the database? should it also have methods that enable the report view to display aggregated data?

- If the test setup smells, most probably the code does too. The harder it is to unit test a piece of code, the greater the chances that SOLID principles are being violated.

Your unit-test code tries to keep up with this bloat and in the process becomes increasingly complex as you try and satisfy the various dependencies (database, report view, aggregator) your code under test needs. What is more, some tests use a certain dependency while for others it is just dead-weight. In this example, the database code does not care about the aggregator, but you have to mock it nevertheless. The aggregator needs just the entity but you still find yourself needing to mock the database. Unit-testing starts to feel harder and often it is time to take a step back and think if you would do better splitting the code under tests into several classes, each handling some aspect of the larger operation. Then the unit-tests would also be split and each unit-test will have far fewer dependencies and will become easy to setup and teardown again.

Open/Closed

Once your class has been thoroughly unit-tested, you have indirectly closed it for modification. Any change to the internal logic of your class will result in a unit-test failure. This discourages people from changing your working code. Internal logic changes are of course required if fixing a bug, but I am not talking about bug-fixing here. Often someone who cleverly changes the logic in your class to make it do something extra will cause a test failure and will have to rethink your test or revert their change. So due to unit-tests extra effort is required to change behaviour and most good programmers will intuitively explore ways of avoiding that extra effort. And often the correct way to do that is to think how the class can be extended or enclosed within another class where the new functionality can live without breaking existing unit-tests. Good unit-tests enforce behaviour, and change in behaviour will break tests causing more rework for the programmer who breaks them. Unit tests encourage you to explore ways of using your class without breaking unit-tests.

Liskov Substitution Principle

This can often be one of the hardest to understand from SOLID principles, and really only applies when using inheritance. Generally, when you can successfully substitute a mock object for the real one, you are in a way following this principle, because your code under test can't tell the difference. It does not care that the real object has been substituted by a mock, it still uses the dependency in exactly the same way as before. Another way to put it is that if you find you are having to tell your code under test that it is running in "test mode" and doing something special in your code under test when in test mode you are very likely violating this principle. Specifically, when testing inheritance, if your derived class object can still pass all of the base class tests you have honoured this principle. If it cannot, you probably should not be using inheritance

SOLID – short primer

- **S - Single Responsibility Principle:** *An object should do exactly one thing, and should be the only object in the codebase that does that one thing.* For instance, take a domain class, say an Invoice. The Invoice class should represent the data structure and business rules of an invoice as used in the system. It should be the only class that represents an invoice in the codebase. This can be further broken down to say that a method should have one purpose and should be the only method in the codebase that meets this need.

By following this principle, you increase the testability of your design by decreasing the number of tests you have to write that test the same functionality on different objects, and you also typically end up with smaller pieces of functionality that are easier to test in isolation.

- **O - Open/Closed Principle:** *A class should be open to extension, but closed to change*. Once an object exists and works correctly, ideally there should be no need to go back into that object to make changes that add new functionality. Instead, the object should be extended, either by deriving it or by plugging new or different dependency implementations into it, to provide that new functionality. This avoids regression; you can introduce the new functionality when and where it is needed, without changing the behavior of the object as it is already used elsewhere.

By adhering to this principle, you generally increase the code's ability to tolerate "mocks", and you also avoid having to rewrite tests to anticipate new behavior; all existing tests for an object should still work on the un-extended implementation, while new tests for new functionality using the extended implementation should also work.

- **L - Liskov Substitution Principle:** *A class A, dependent upon class B, should be able to use any X:B without knowing the difference.* This basically means that anything you use as a dependency should have similar behavior as seen by the dependent class. As a short example, say you have an IWriter interface that exposes Write(string), which is implemented by ConsoleWriter. Now you have to write to a file instead, so you create FileWriter. In doing so, you must make sure that FileWriter can be used the same way ConsoleWriter did (meaning that the only way the dependent can interact with it is by calling Write(string)), and so additional information that FileWriter may need to do that job (like the path and file to write to) must be provided from somewhere else than the dependent.

This is huge for writing testable code, because a design that conforms to the LSP can have a "mocked" object substituted for the real thing at any point without changing expected behavior, allowing for small pieces of code to be tested in isolation with the confidence that the system will then work with the real objects plugged in.

SOLID – short primer

- **I - Interface Segregation Principle:** *An interface should have as few methods as is feasible to provide the functionality of the role defined by the interface*. Simply put, more smaller interfaces are better than fewer larger interfaces. This is because a large interface has more reasons to change, and causes more changes elsewhere in the codebase that may not be necessary.

Adherence to ISP improves testability by reducing the complexity of systems under test and of dependencies of those SUTs. If the object you are testing depends on an interface IDoThreeThings which exposes DoOne(), DoTwo() and DoThree(), you must mock an object that implements all three methods even if the object only uses the DoTwo method. But, if the object depends only on IDoTwo (which exposes only DoTwo), you can more easily mock an object that has that one method.

- **D - Dependency Inversion Principle:** *Concretions and abstractions should never depend on other concretions, but on abstractions*. This principle directly enforces the tenet of loose coupling. An object should never have to know what an object IS; it should instead care what an object DOES. So, the use of interfaces and/or abstract base classes is always to be preferred over the use of concrete implementations when defining properties and parameters of an object or method. That allows you to swap one implementation for another without having to change the usage (if you also follow LSP, which goes hand in hand with DIP).

Again, this is huge for testability, as it allows you, once again, to inject a mock implementation of a dependency instead of a "production" implementation into your object being tested, while still testing the object in the exact form it will have while in production. This is key to unit testing "in isolation".

What to test: The Right-BICEP*

Right – Are the results right?

- B: Are all the boundary conditions correct?
- I: Can you check inverse relationships?
- C: Can you cross-check results using other means?
- E: Can you force Error conditions to happen?
- P: Are performance characteristics within bounds?

**from: Hunt/Thomas: Pragmatic Unit Testing in Java with JUnit*

The CORRECT* acronym

- C: Conformance – Does the value conform to an expected format?
- O: Ordering – Is the set of values ordered or unordered as appropriate?
- R: Range – Is the value within reasonable minimum and maximum values?
- R: Reference – Dooes the code reference anything external that is not under direct control of the code itself?
- E: Existence – Does the value exist? (e.g. non-null, non-zero, present in a set, etc.)
- C: Cardinality – Are there exactly enough values?
- T: Time – (absolute and relative) Is everything happening in order? At the right time? In time?

**as before*

Properties of good tests*

A-TRIP:

- A: Automatic – can be run without user intervention.
- T: Thorough – test everything the is likely to break.
- R: Repeatable – produce the same result.
- I: Independent – test one thing at a time.
- P: Professional – written and maintained to the same standard as the production code. (expect to write as much test code as production code!)

**as before*

How to fix a bug*

When a bug is found ”in the wild” and reported back, that means that there is a ”hole in the net” – a missing test. This is your opportunity to close the hole and make sure this bug never escapes again.

1. Identify the bug.
2. Write a test that fails, to prove the bug exists.
3. Fix the code such that the test now passes.
4. Verify that ALL other tests still pass (i.e. you did not break anything else as a result of the fix).

**as before*