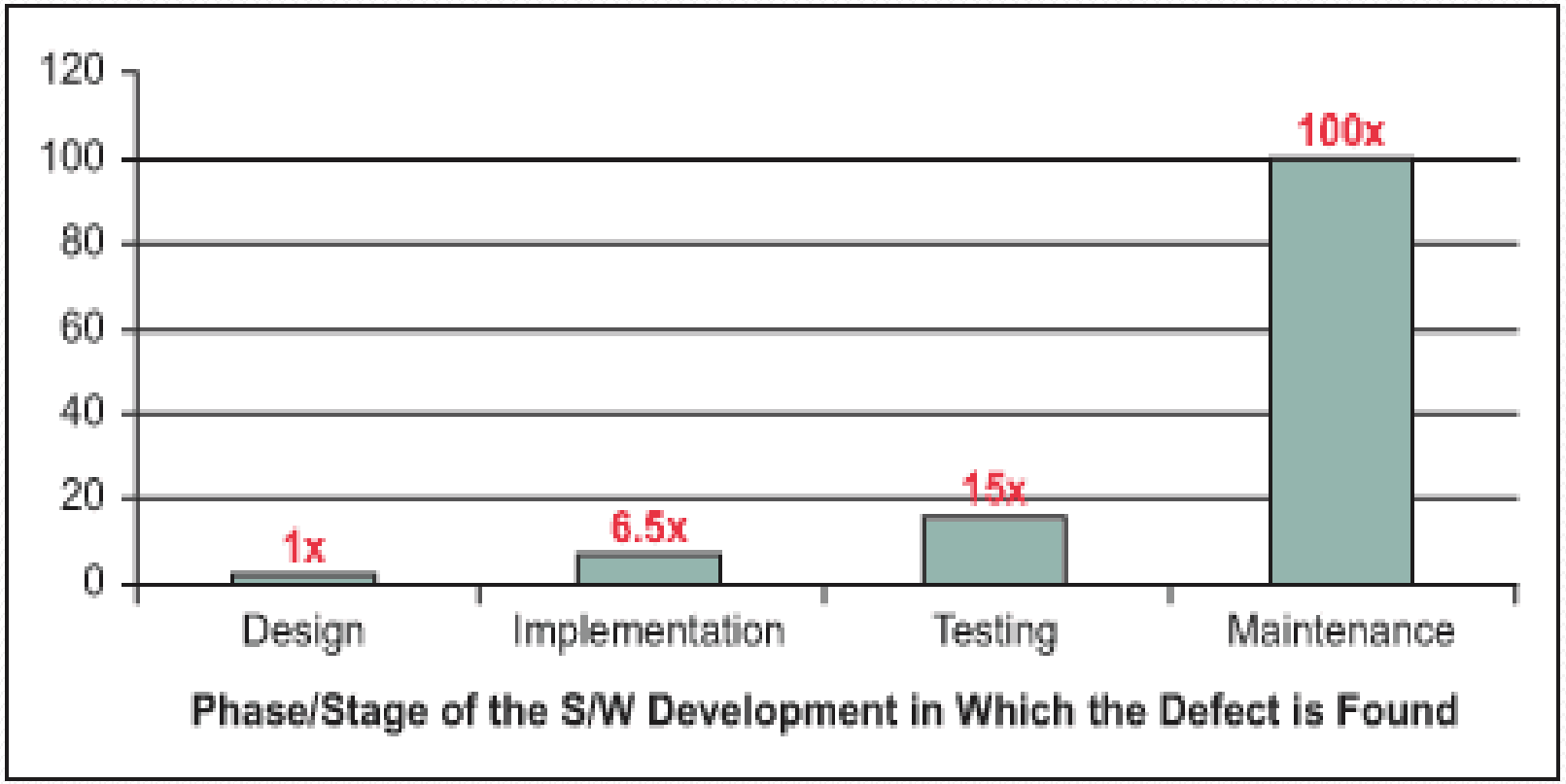
Quality Assurance in Software Projects

Validation and Verification

Costs to fix defects



Source: Implementing Software Inspections,
IBM Systems Sciences Institute, IBM, 1981

Quality

- Product to be assured to achieve some quality level by the whole team.
- QA team verifies if that quality level is achieved

Testing

- Tests can reveal a defects, but tests cannot create bug-free products.
- Need to assure testability
- Report issues
- Track issues until resolution

Validation & Verification (V&V)

- Quality Assurance (QA) of software
- **Validation** ensures "you built the right thing”
  - Meets expected use
- **Verification** ensures that "you built it right”
  - Meets requirements & design

Program testing

- Purpose: Prior to releasing
  - assure that program behaves as **intended**
  - discover program defects
- Program is executed with artificial data.
- Check for
  - errors
  - information about non-functional behavior
- Impossible to detect all errors
  - can not prove there are no errors.

Program testing goals

- To demonstrate that the software meets its requirements.
- For developers and customers
- All of the **system features** should be tested
- Meaningful **combinations of features** should be tested
- Detect when software is not behaving correctly
  - system crashes
  - unacceptable interactions with other systems
  - incorrect computations
  - data corruption

Validation and defect testing

- Validation testing
  - System performs as expected
  - Use test cases that reflect the system’s expected use
- Defect testing
  - Expose defects
  - Tests can be deliberately obscure
  - Test may not reflect expected use.

Testing process goals

- Validation testing
  - To show system works as expected
  - A successful test shows that the system operates as intended
- Defect testing
  - To discover defects
  - A successful test makes the system perform incorrectly exposes a defect in the system.

Software Testing

- Test engineer teams should test
- Use software to build and execute test scripts
- eXtreme Programming (XP) programmers write test harnesses for classes before they write the code
- Test Driven Development (TDD), all tests are written first
- User acceptance – users of the system test according the requirements

Testing approaches

- **Black box** testing
  - Does it do **what** it's meant to do? (functional)
  - Does it do it as **fast** as it should? (non-functional)
  - No information on how it does it

- **White box** testing
  - Is the solution a **good** solution?
  - Tester can see **how** the solution is provided

Purpose of Testing

- Aim: Identify errors
- Report errors
- Fix errors
- When testing if no error is found
  - Does not mean it is error free
  - No software is perfect

Testing criteria

- Data ranges
  - **extreme** values (very large numbers, long strings)
  - **borderline** values (0, negative, 9.999999999)
  - **invalid combinations** of values
    - age = 3, educational degree = graduate level
  - nonsensical values (negative order line quantities)
- Stress test
  - Does it perform well under heavy loads?
  - i.e. many customers at the same time

Levels of Testing

- **unit** testing
  - tests methods and classes
- **integration** testing
  - classes work correctly together
- **subsystem** testing
  - subsystem meet required functionality
- **system** testing
  - system works as a whole
- **acceptance** testing
  - system works to meet users specification

Levels of Testing

- Level 1 Test:
  1. modules (classes)
  2. programs (use cases)
  3. suites (application)
- Level 2 (Alpha Testing or **Verification**)
  - Execute programs in a **simulated environment**
  - Test inputs and outputs
- Level 3 (Beta Testing or **Validation**)
  - **live user environment**
  - test for response times
  - performance under load
  - recovery from failure

Test Documentation -Test plans

- Written before
  - tests are carried out
  - code is written

- Test Cases
  - description of test
  - test environment and configuration
  - test data
  - **expected outcomes**

Documentation - Test results

- Should be well documented
  - Spreadsheet
  - Database
- Record outcome of test
  - Failed – What was output as opposed to expected
  - Passed
- Report percentage passed
- Should be linked to requirements
- Error results must be reported
  - sufficient details for developers to **reproduce** error

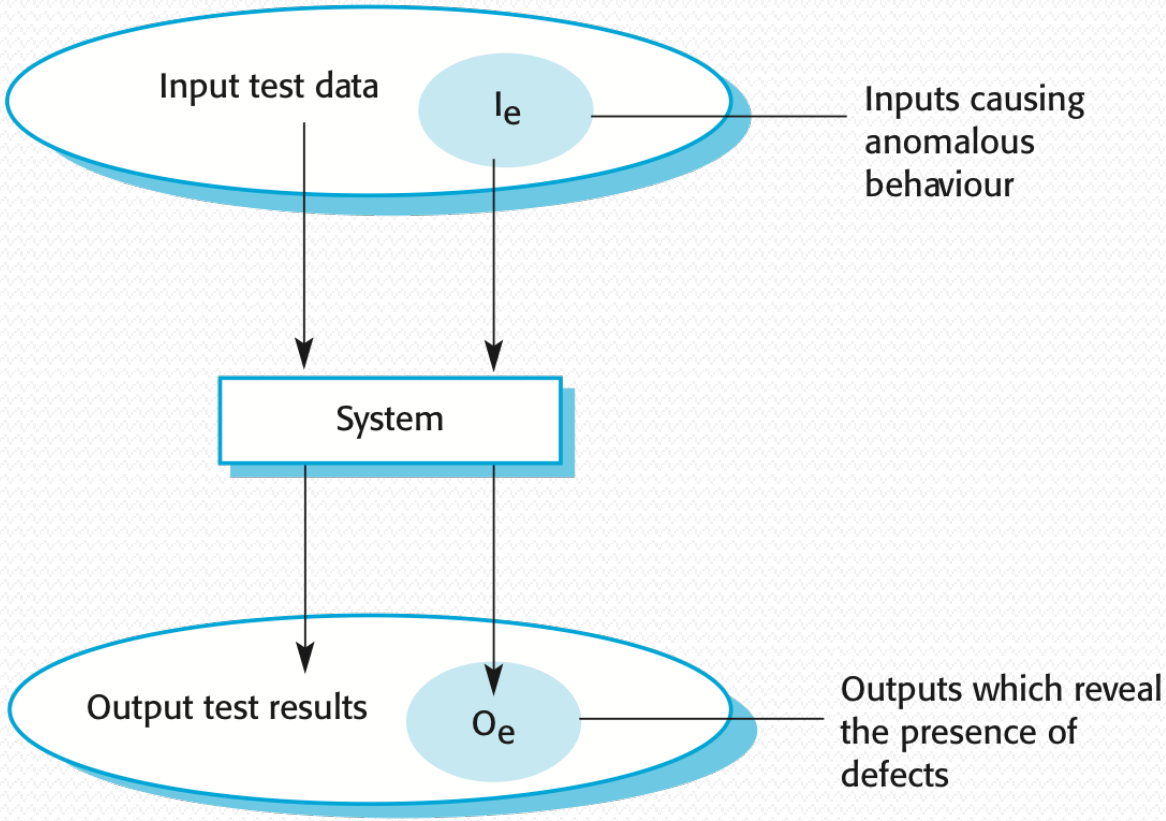
2.1.6 Test Phase: Default Encoding Type

Step No.	Step Description	Expected Results	P/F	Comments
1.	Open XYZ IDE. Open Tools->Recording Options-> Default Encoding Type tab. Check default encoding type.	Encoding type should be "Auto-Detect" by default.		
2.	Select an option as an default encoding type.	Any of the types can be selected.		
3.	Select an option as an default encoding type. Click "OK" Button. Record a website including selected encoding type.	Characters should be recorded properly.		
4.	Click "Help" button.	XYZ IDE User's Guide should be opened. Displayed User's Guide content should be "Configuring the Default Encoding Type"		

Test Phase Passed / Failed: _____

Date: _____ Tester Name: _____

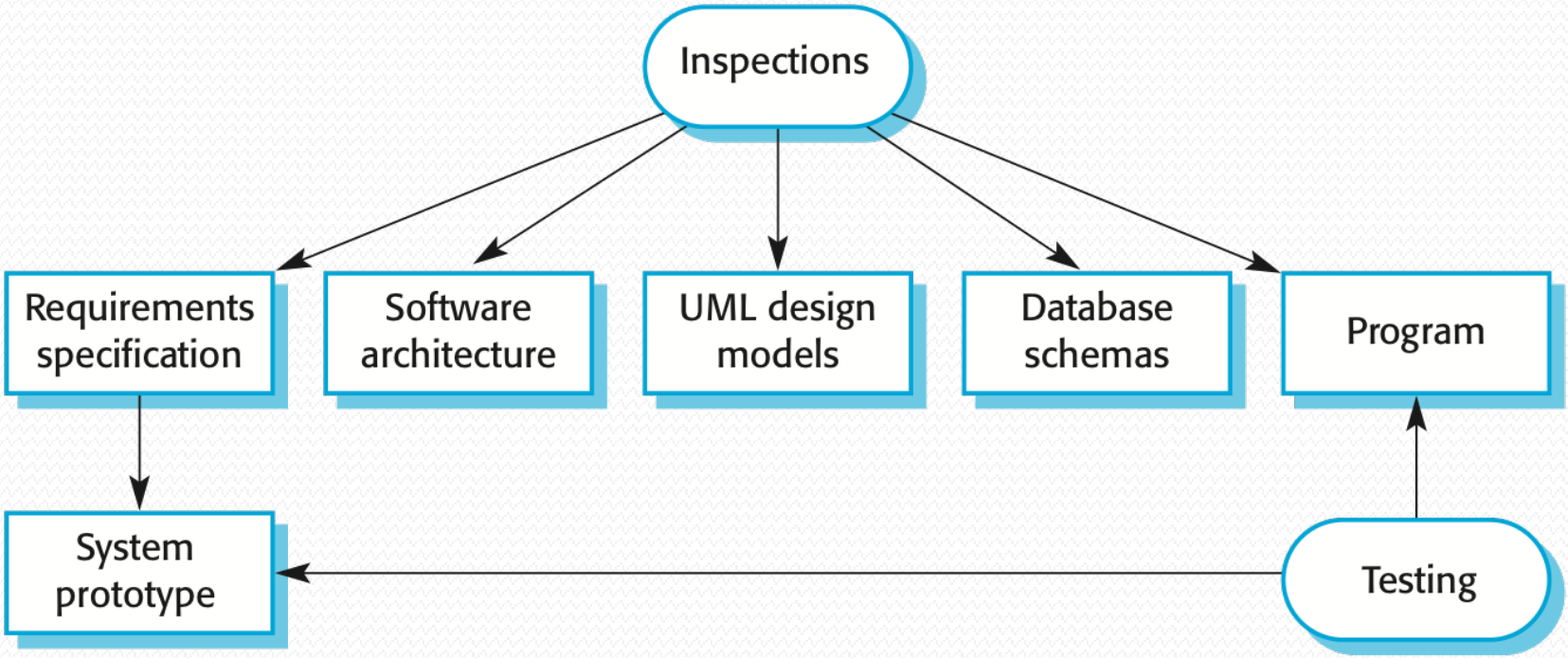
Input-output program testing



Inspections and testing

- Software inspections (static verification)
  - Inspection of code
  - Conventions, good practices, measures, …
  - Tools can be used
- Software testing (dynamic verification)
  - Observe product behaviour
  - System is executed with test data

Inspections and testing



Software inspections

- Source representation are inspected
  - To discover defects
- Any source can be inspected
  - Requirements
  - Design
  - Configuration data
  - Test data
  - Code
- Effective for discovering program errors. Why?

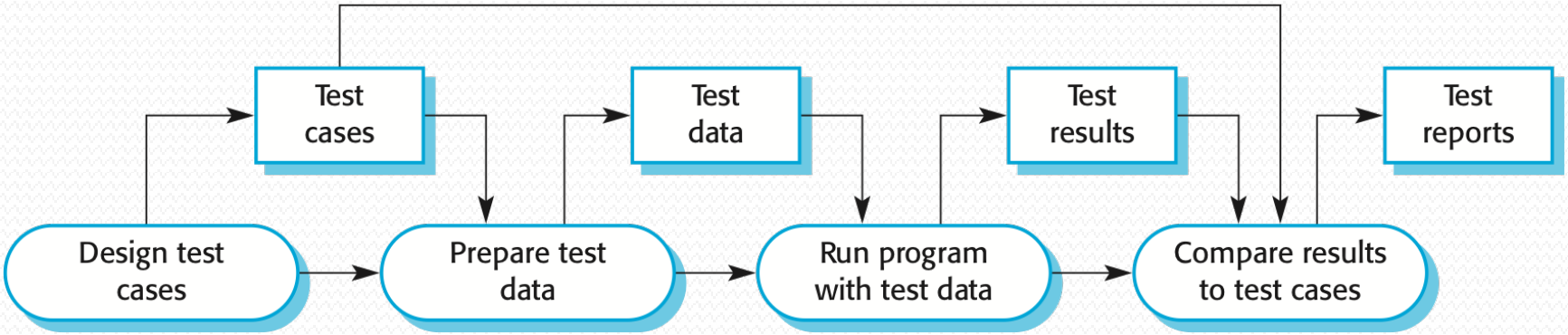
Advantages of inspections

- Errors can hide other errors, inspection avoids interactions between errors.
- Incomplete versions of a system can be inspected.
- Check for higher quality issues
  - compliance with standards
  - portability
  - maintainability
  - …

Inspections and testing

- Inspections and testing are complementary
- Both should be used during the V & V
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

A model of the software testing process



Stages of testing

- **Development** testing
  - System is tested during development to discover bugs and defects.
- **Release** testing
  - **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.

Reporting Bugs

- Before we dive into testing processes, let us consider what to do when we have bugs.
- Remember we are always focused on tracking aspects of software

Bad Bug reports

- It doesn't work!
- I just tried to save a new customer and it didn't work properly
- system is really slow

ill bad bug example

Araç : Bug Tool
Konu: sorunlu kayitlar – I
Priority: Urgent!!
Açıklama:

comment posted by mehmeta on 2007-02-12

- Admin menu'deki bunu sayfayi kullanacak kullanicilara yetki vermek icin, Yetki Tanimlari menusunde, bu sayfayi "kaynak" olarak tanimlamam gerekiyor. Kaynak adresi nedir?

- Temsilci numarasi alani zorunlu degil. Ama bos biraktigimda hata veriyor su anda. Zorunlu olmamali ama bu alana bir numara yazilmissa, bu numaranin gecerli olup olmadigi kontrol edilmeli.

- Sayfanin altindaki "not" alanini 2 kat daha genis yapabilir misin? 250pixel falan olabilir.

- Normal bolge numaralarimiz 100-999 arasinda oluyor. 100'den kucuk olan bolge numaralari ozel amacli olarak kullaniliyor. Hem formu kaydetmeden once, hem de SMS gonder butonu tiklandiginda, yazilan bolgenin uygun (100 - 999 arasinda) olup olmadigini kontrol edelim lutfen. (100 ve 999 dahil.). Bolge olarak 10 yazdigimda, SMS gonderilecek bölgeyi hatali tespit etti. Ekteki ornekte Ahmet B., 10 numarali bolgenin sorumlusu degil.

- SMS gonderilen pencerede, sag ust kosede "ana sayfa" ve "cikis" linkleri olmasin lutfen.

- SMS metninin sonunda GTM yaziyor. GTM kelimesinden once 1 karakter bosluk birakalim lutfen.

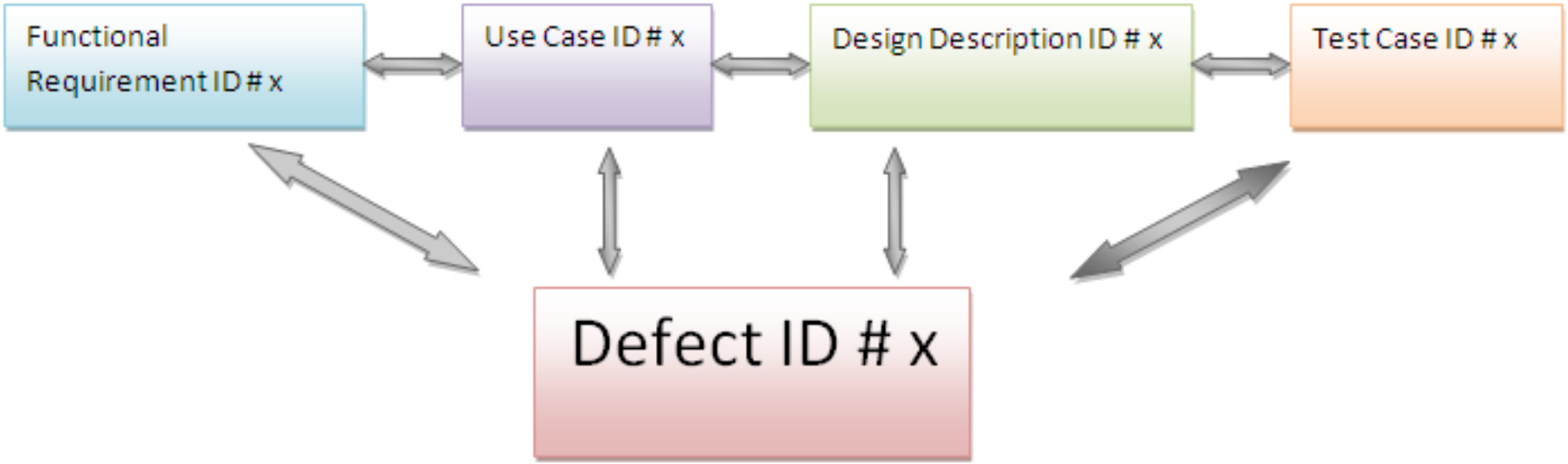
Useful Fields

- Bug **Title**: A good name that describes the bug
- Bug **ID**: automatically created by BUG Tracking tool
- Application **area**: Which part of the application it occurs in
- **Build Number**: Version Number
- **Severity**: HIGH (High/Medium/Low) or 1
- **Priority**: HIGH (High/Medium/Low) or 1
- **Assigned to**: who should fix the bug
- **Reported By**: name of person reporting the bug
- **Reported On**: Date
- **Reason**: Defect/Enhancement/New Feature
- **Status**: New/Open/Active/ …
- **Environment**: Versions of related apps and OS

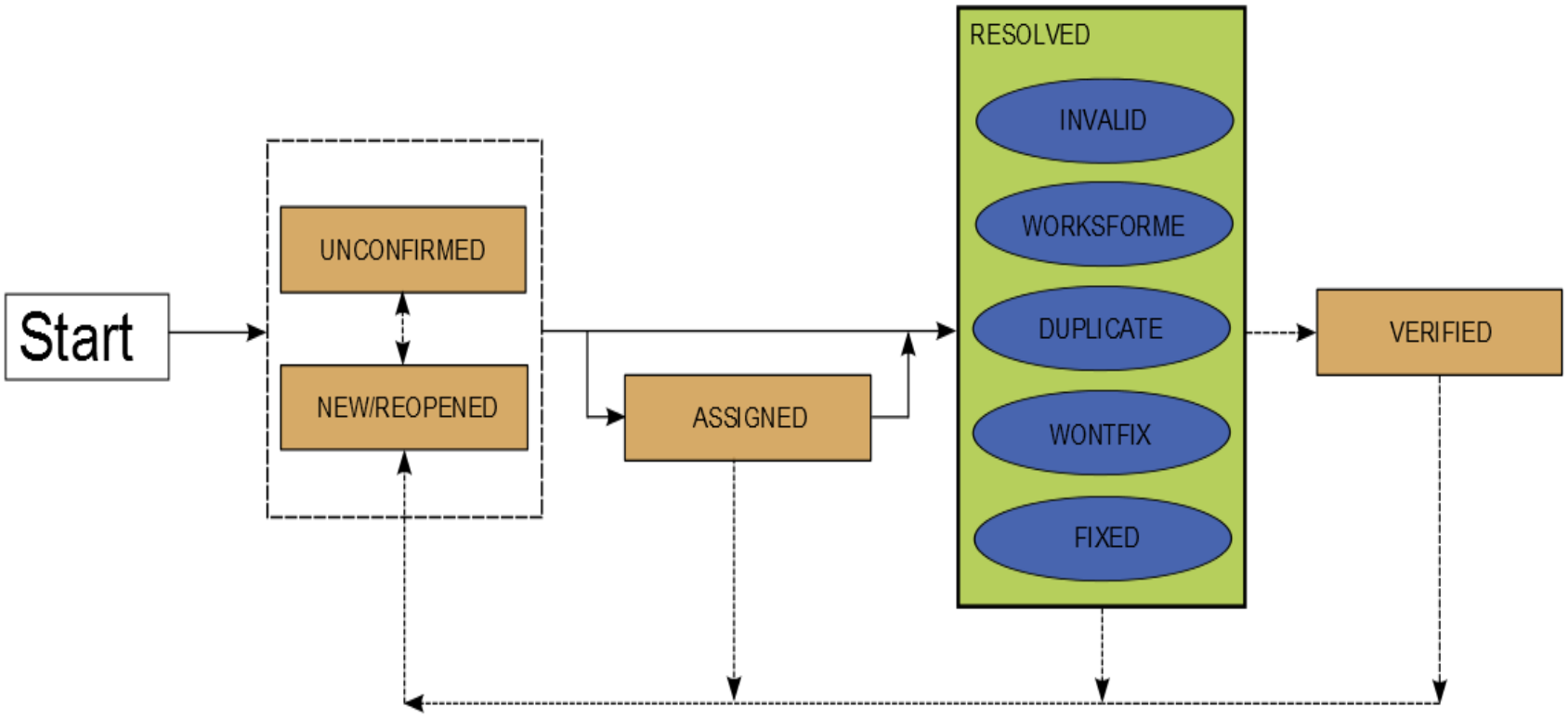
- **Description**: Clear description of issue with how to reproduce

Let's Examine a Real Bug report

- From the popular D3 application on Github
- See:
  - https://github.com/mbostock/d3/issues/1883
  - How do you evaluate this report?


Functional Requirement ID # x
Use Case ID # x
Design Description ID # x
Test Case ID # x
Defect ID # x

Bug Lifecycle



Development testing

- All tests that by the development team
  - **Unit** testing
    - individual program units or object classes are tested
    - tests the functionality of objects or methods
  - **Component** testing
    - several units composed into components.
    - tests component interfaces
  - **System** testing
    - system is tested as a whole
    - tests component interactions.

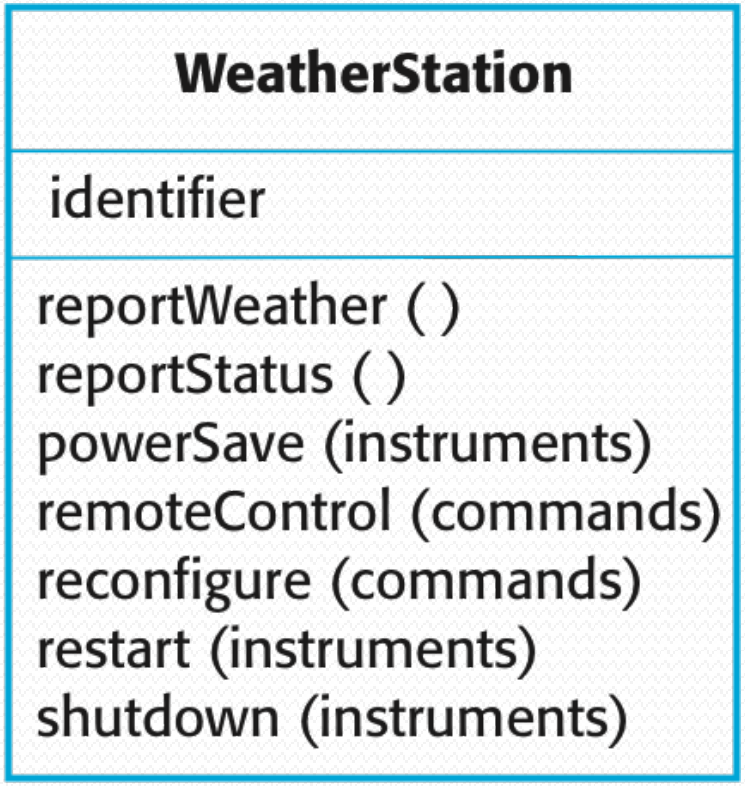
Unit testing

- Testing individual components in isolation.
- It is a **defect** testing process.
- 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
  - Setting and interrogating all object attributes
  - Exercising the object in all possible states.
- Inheritance makes it more difficult to design object class tests as the information to be tested is not localised.

The weather station object interface



Weather station testing

- Need to define test cases for reportWeather, calibrate, test, startup and shutdown.

Weather Station Testing

- Use a state model to identify
  - sequences of state transitions to be tested
  - the event sequences to cause these transitions

- For example:
  - Shutdown -> Running-> Shutdown
  - Configuring-> Running-> Testing -> Transmitting -> Running
  - Running-> Collecting-> Running-> Summarizing -> Transmitting -> Running

Automated testing

- Automated **unit testing** should be used

- In automated unit testing
  - test automation framework (i.e **JUnit**)
  - **write** and **run** tests
  - generic test classes extended to create specific test cases
  - all tests are run automatically
  - success and failures are reported

Automated test components

- Test case:
  - inputs
  - expected outputs
- Call part: where the unit to be tested is called
- Assertion part:
  - compare result of the call to expected result.
  - If the assertion evaluates to
    - true → success
    - Otherwise → fail

Unit test effectiveness

- Test normal operation
  - show that the component works as **expected**
- Test defects
  - use abnormal inputs
  - used to verify if poor input is properly processed
  - verify it does not crash the component

Unittest / Python

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())
```

Unittest – cont.

```
    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello',
'world'])
        # check that s.split fails when the
separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

The test runner inspects code

- Test methods start with “test”
- To check for a condition call
  - assertEqual() check for an expected result
  - assertTrue() check if true
  - assertFalse() check if false
  - assertRaises() check if raises condition
- unittest.main() provides a command-line
- Ran 3 tests
  in 0.000s

  OK

Running tests

- Sometimes we need to set up conditions to test
- SetUp() – set up environment
- TearDown() – reset environment
- You can group tests with suites

```
def suite():
    suite = unittest.TestSuite()
    suite.addTest(WidgetTestCase('test_default_size'))
    suite.addTest(WidgetTestCase('test_resize'))
    return suite
```

Interface testing

- Detect defects due to interface errors or wrong assumptions
- Interface types
  - **Parameter** interfaces:
    - Data passed from one method to another
  - **Shared memory** interfaces:
    - Block of memory is shared between procedures or functions
  - **Procedural** interfaces:
    - Sub-system encapsulates a set of procedures to be called by other sub-systems.
  - **Message passing** interfaces:
    - Sub-systems request services from other sub-systems

Interface errors

- Incorrect use
  - parameters called in the wrong order.
- Misunderstanding
  - Incorrect assumptions about the behaviour of called component
- Timing errors
  - the called and the calling component operate at different speeds
  - information is out of date

Interface testing guidelines

- Test parameters with **extreme values**
- Test pointer parameters with **null pointers**
- Design tests to **cause the component to fail**
- Use stress testing in message passing systems

System testing

- During development
  - integrate components to create a system
  - test it!!
- Test interactions between components
- Check if components are
  - compatible
  - interact correctly
  - transfer the right data at the right time across interfaces

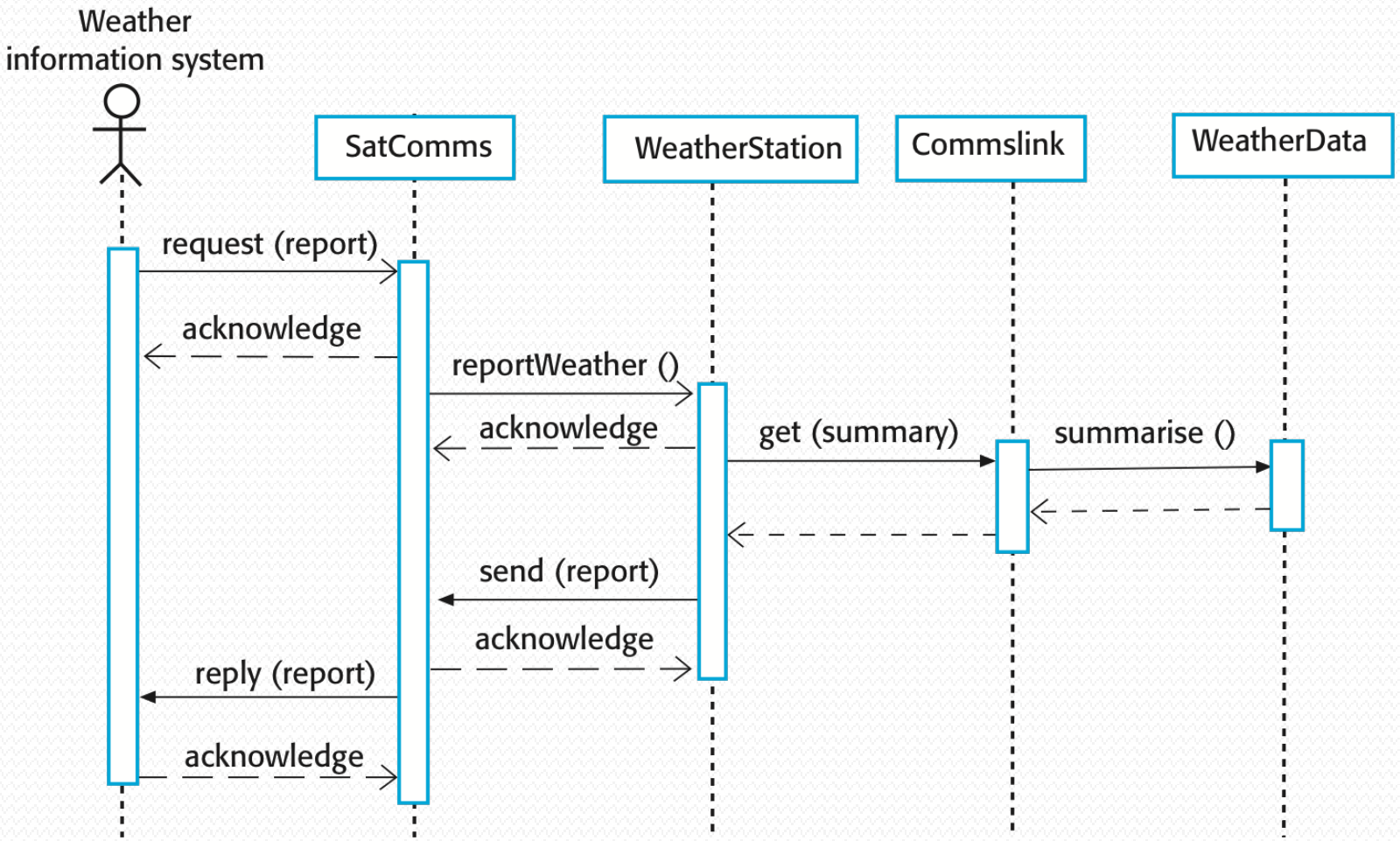
System and component testing

- Collective process
- Components developed by different team members
- Testing teams
  - Separate testing people
  - Designers and programmers are not involved
  - Q: what is the advantage of having testers?

Use-case testing

- **Use-cases** can be used to test system
- Use case involve several system components
- Interactions are required to test
- The sequence diagrams of the use case specifies the **components** and **interactions**

Collect weather data sequence chart



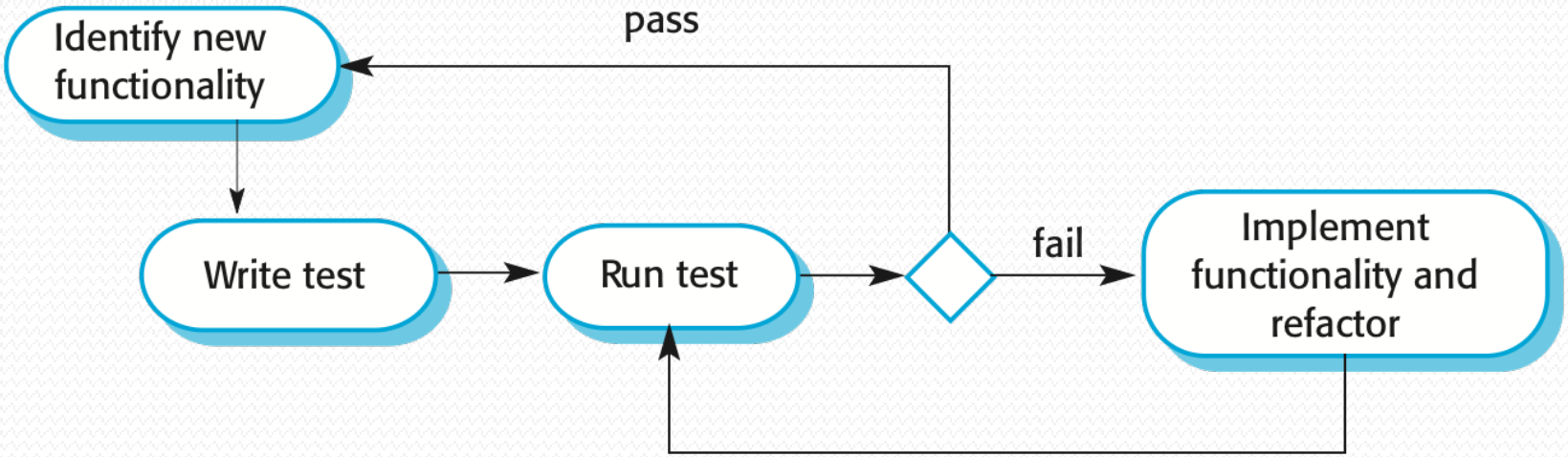
Testing policies

- Testing policies should be determined
- Exhaustive system testing is impossible
- A testing policy describes what should be tested
  - functions accessed from menus
  - combinations of functions accessed from same menu
  - functions that require user input with
    - correct input
    - Incorrect input

Test-driven development

- Test-driven development (TDD) is an approach to software development
- Tests are written **before** code
- **Passing** the tests drives the development
- Code is developed incrementally with tests
- A new increment can not be developed until the previous increment succeeds (passes its test)

Test-driven development



TDD process

- Identifying functionality of the increment
- Functionality should be implementable in a few lines
- Write a test for this functionality (an automated test)
- Run the test
  - First call will fail (not implemented, yet)
  - Implement the functionality
  - re-run the test until succeeds
- Run all remaining tests
- When all tests run successfully increment is done!

Benefits of test-driven development

- Code coverage
  - all code has at least one test
- Regression testing
  - A regression test suite is naturally created
- Easier to debug
  - the reason for fail is easier to identify (Q: Why?)
- System documentation
  - The tests document and describe the

Regression testing

- checks that changes to code have not '**broken**' previously working code
- Regression testing is expensive if done manually
- Simple and clear if done automatically:
  - **All tests** are rerun **every time** a change is made to the program.
- All tests must run 'successfully' before the change is committed/pushed to project repo.

Release testing

- Tests a particular release intended for users
- Goal: Is the system good enough to use?
- Must demonstrate that the system
  - delivers specified functionality
  - performance
  - dependability
  - does not fail during normal use

Requirements based testing

- Requirements-based testing involves examining each requirement and developing a test or tests for it.
- Example:
  - For a patient who is allergic to any medication, a prescription for that medication will cause a warning
  - A prescriber who wants to ignore an allergy must provide a reason for this decision

Test Environment

- Test environment requirements
  - Hardware – CPU, RAM, Disk,..
  - Software – OS, Application Server, Web Server, DB Server, Browser,...

- Test setup

User testing

- User or customer provides input on testing
- User testing is essential!
- User's working environment has impact on
  - reliability
  - performance,
  - usability
  - robustness
- difficult be replicate in a testing environment

Types of user testing

- **Alpha** testing
  - Users and development team test at the **developer site**
- **Beta** testing
  - A release of the software is created
  - Users report problems to developers
- **Acceptance** testing
  - Deployed in the **customer environment**
  - Customers test to decide if system is ready to be used

Stages in acceptance testing

- Define acceptance criteria
- Plan acceptance testing
- Derive acceptance tests
- Run acceptance tests
- Negotiate test results
- Reject/accept system

Fun Reading

- How to write annoying bug reports:
  - http://maurits.wordpress.com/2012/03/14/bug-reporting-8-ways-to-annoy-your-software-development-team/