SOFTWARE ENGINEERING

Chapter 9 – Software Testing

TUẦN 14

Topics covered

- Development testing
- Test-driven development
- Release testing
- User testing

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.
- When you test software, you execute a program using artificial data.
- You check the results of the test run for errors, anomalies or information about the program's non-functional attributes.
- Can reveal the presence of errors NOT their absence.
- Testing is part of a more general verification and validation process, which also includes static validation techniques.

Program testing goals

- To demonstrate to the developer and the customer that the software meets its requirements.
  - For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
- To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
  - Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.

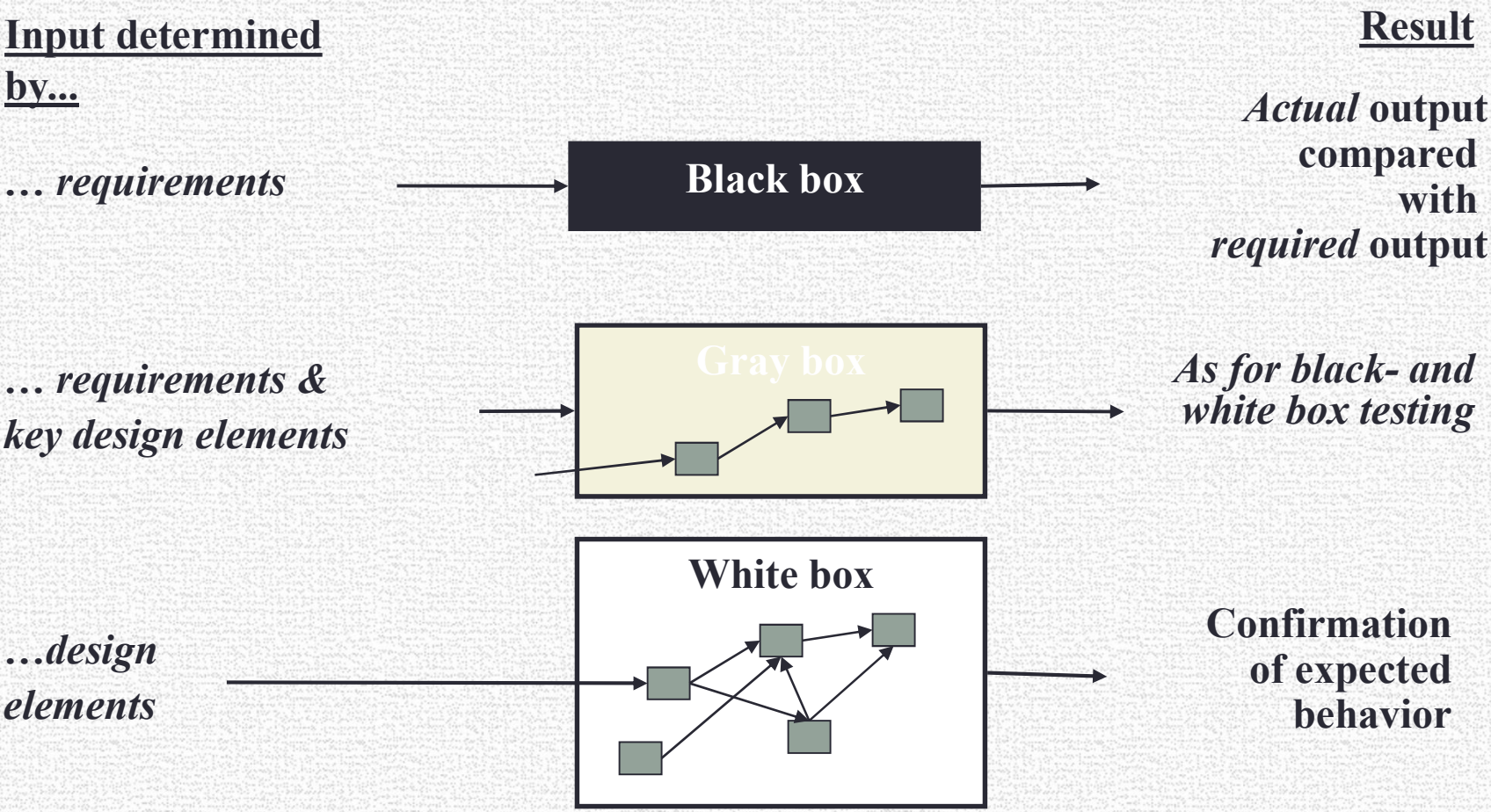
Validation and defect testing

- The first goal leads to validation testing
  - You expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
- The second goal leads to defect testing
  - The test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.

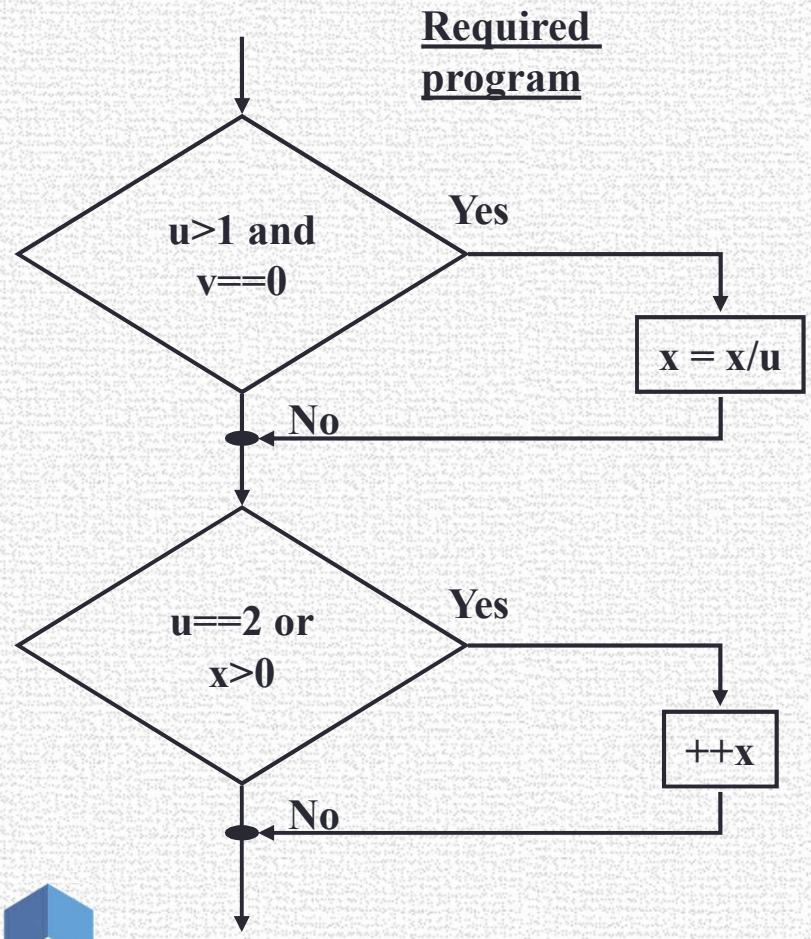
Testing process goals

- Validation testing
  - To demonstrate to the developer and the system customer that the software meets its requirements
  - A successful test shows that the system operates as intended.
- Defect testing
  - To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

Black-, Gray-, & White-box Testing



Statement Coverage for White Box Testing: Covering Every Statement is Not Sufficient



Code attempt to implement flowchart

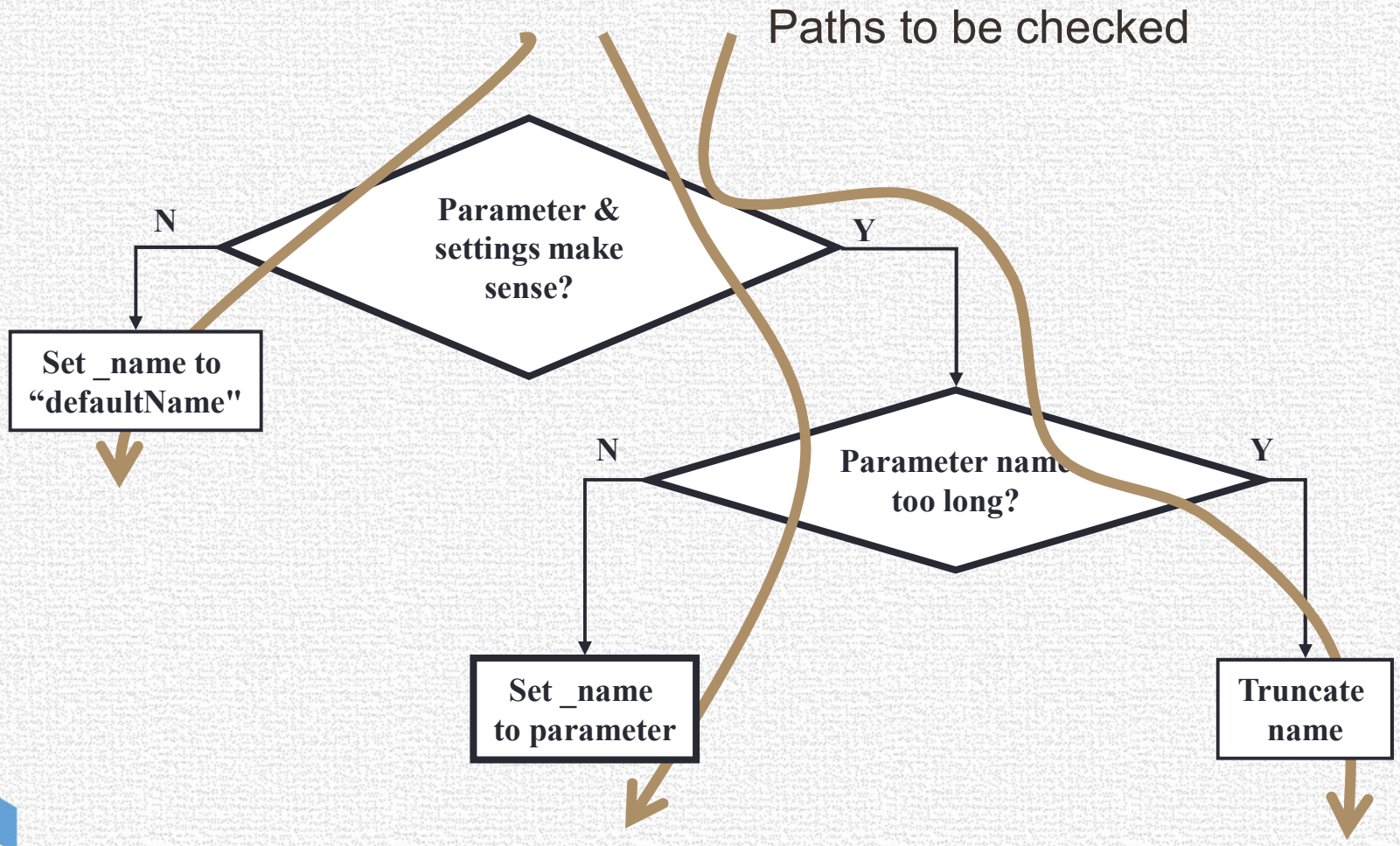
```
if( (u>1) && (v==0) )        (1)
      x = x/u;               (2)
if( (u==2) || (x>3) )        (3)
      ++x;                   (4)
```

u=2, v=0 and x=3

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

However, line (3) is wrong

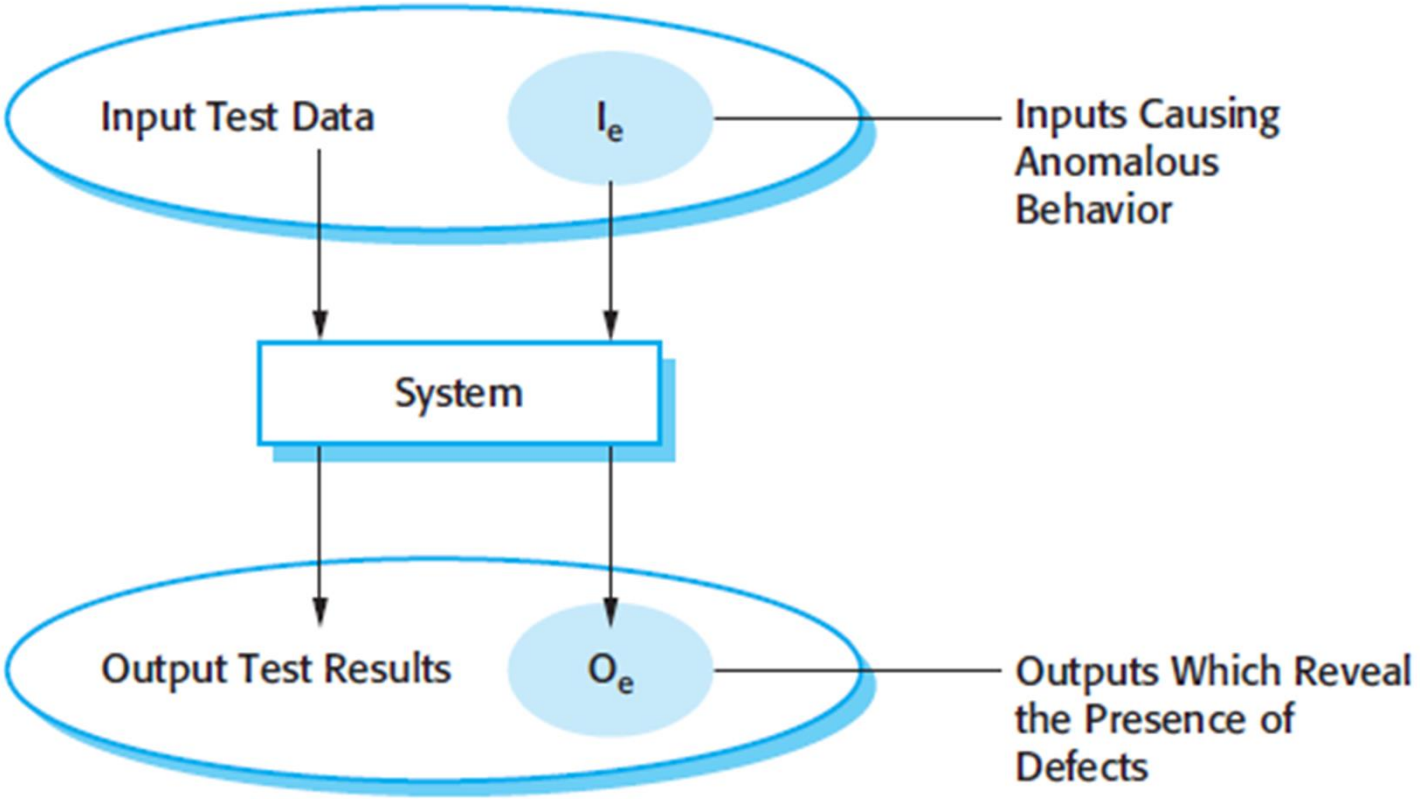
Decision Coverage for White Box Testing



Assertion-based Testing: A White Box Testing

- In many cases, assertion = invariant
- Insert assertions into the source code
  - Define assertion
  - Place assertion (into the source code)

An input-output model of program testing



Verification vs validation

- Verification:

  "Are we building the product right".
  - The software should conform to its specification.
- Validation:

  "Are we building the right product".
  - The software should do what the user really requires.

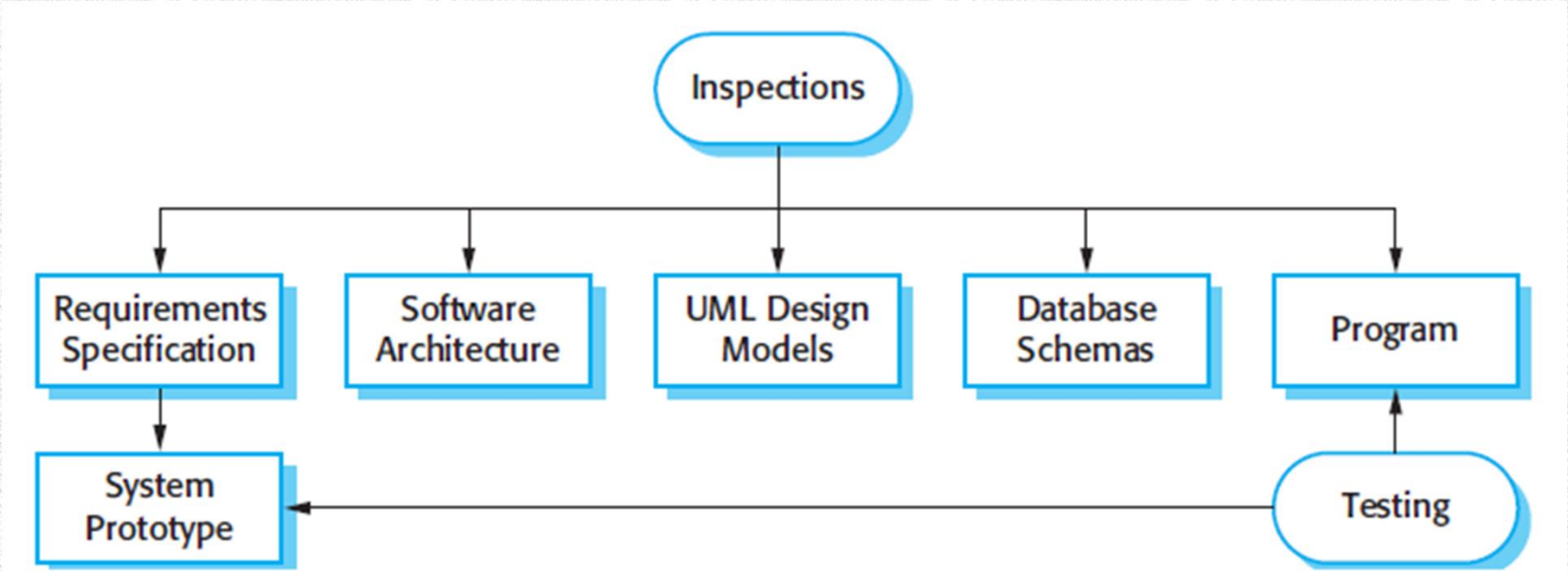
V & V confidence

- Aim of V & V is to establish confidence that the system is 'fit for purpose'.
- Depends on system's purpose, user expectations and marketing environment
  - Software purpose
    - The level of confidence depends on how critical the software is to an organisation.
  - User expectations
    - Users may have low expectations of certain kinds of software.
  - Marketing environment
    - Getting a product to market early may be more important than finding defects in the program.

Inspections and testing

- Software inspections concerned with analysis of the static system representation to discover problems (static verification)
  - May be supplement by tool-based document and code analysis.
  - Discussed in Chapter 15.
- Software testing concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed.

Inspections and testing



Software inspections

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design,configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

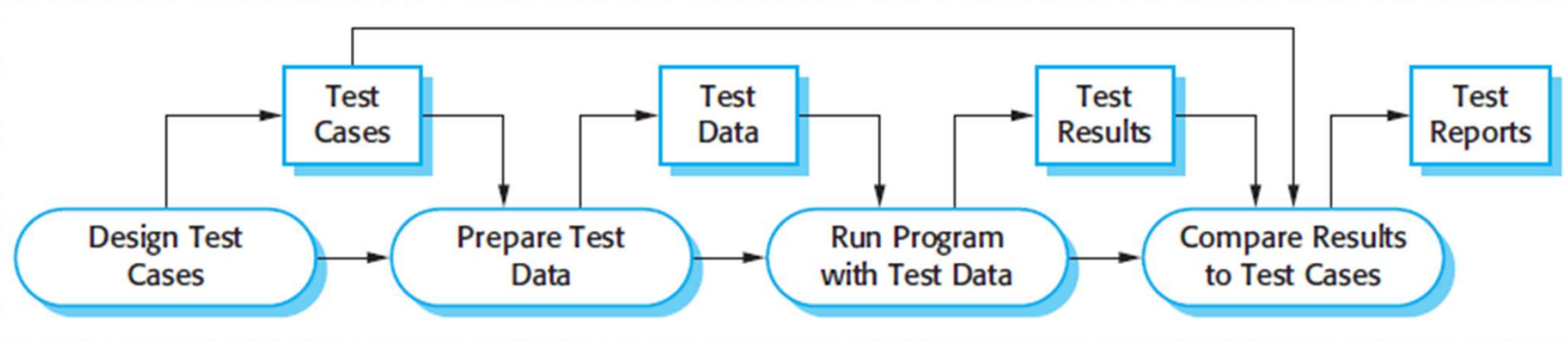
Advantages of inspections

- During testing, errors can mask (hide) other errors. Because inspection is a static process, you don't have to be concerned with interactions between errors.
- Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

Inspections and testing

- Inspections and testing are complementary and not opposing verification techniques.
- Both should be used during the V & V process.
- Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

A model of the software testing process



Stages of testing

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

Development testing

- Development testing includes all testing activities that are carried out by the team developing the system.
  - Unit testing, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
  - Component testing, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
  - System testing, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

Unit testing

- Unit testing is the process of 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.

Perform Method Testing 1/2

One way to ...

- 1. Verify operation at normal parameter values
  - (a black box test based on the unit's requirements)
- 2. Verify operation at limit parameter values
  - (black box)
- 3. Verify operation outside parameter values
  - (black box)
- 4. Ensure that all instructions execute
  - (statement coverage)
- 5. Check all paths, including both sides of all branches
  - (decision coverage)
- 6. Check the use of all called objects
- 7. Verify the handling of all data structures
- 8. Verify the handling of all files

Perform Method Testing 2/2

One way to ...

- 9. Check normal termination of all loops
  - (part of a correctness proof)
- 10. Check abnormal termination of all loops
- 11. Check normal termination of all recursions
- 12. Check abnormal termination of all recursions
- 13. Verify the handling of all error conditions
- 14. Check timing and synchronization
- 15. Verify all hardware dependencies

Perform Class Unit Tests

One way to ...

- 1. Exercise methods in combination
  - 2-5, usually
  - choose most common sequences first
  - include sequences likely to cause defects
  - requires hand-computing the resulting attribute values
- 2. Focus unit tests on each attribute
  - initialize, then execute method sequences that affect it
- 3. Verify that each class invariant is unchanged
  - verify that the invariant is true with initial values
  - execute a sequence (e.g., the same as in 1.)
  - verify that the invariant still true
- 4. Verify that objects transition among expected states
  - plan the state / transition event sequence
  - set up the object in the initial state by setting variables
  - provide first event & check that transition occurred . etc.

Method Combination Test Example

Abbr.	Method prototype
aq	adjustQunlity(…)
d	deleteFromEncounterCharacters(…)
ge	getEncounterCharacter(…)
gq	getQualityValue(…)
gs	getSumOfQualities(…)
gt	getTolerance(…)
io	indexOf(…)
ii	insertIntoEncounterCharacters(…)
m	maxNumCharsInName()
sq	setQuality(…)

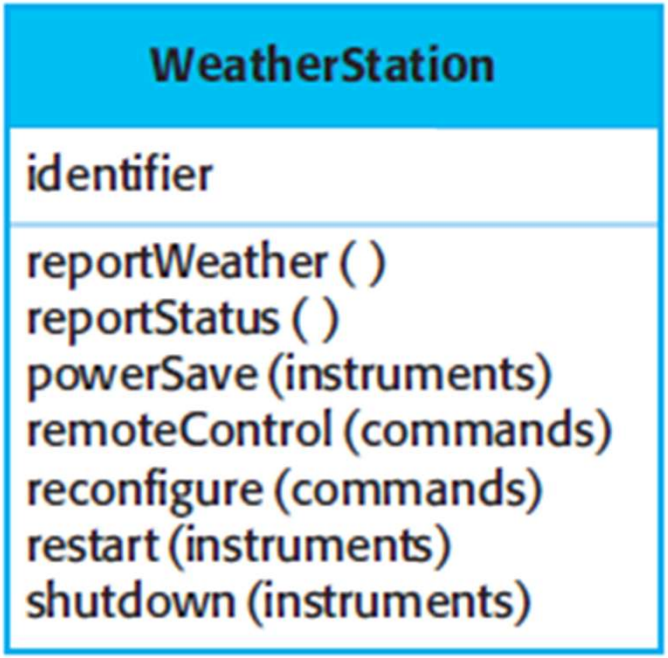
- Concentrate sequence in two wa
  - likely to be commonly used
  - likely to have defects
- Ex: common used
  - ge-aq-gs :
    - get the character—adjust the
    - qualities—get the sum of qualities
  - ge-sq-aq-gq:
    - Get the character—set a quality—adjust the qualities—get the quality
- Ex: causes defects
  - ge-sq-aq-sq-aq-gp:
    - Get the character—set a quality—adjust the qualities—set a quality—adjust the qualities—get the quality

Complicated process => likely causes defects

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.
- Using a state model, identify sequences of state transitions to be tested and 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

- Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- In automated unit testing, you make use of a test automation framework (such as JUnit) 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.

Automated test components

- A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.
- A call part, where you call the object or method to be tested.
- 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.

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.

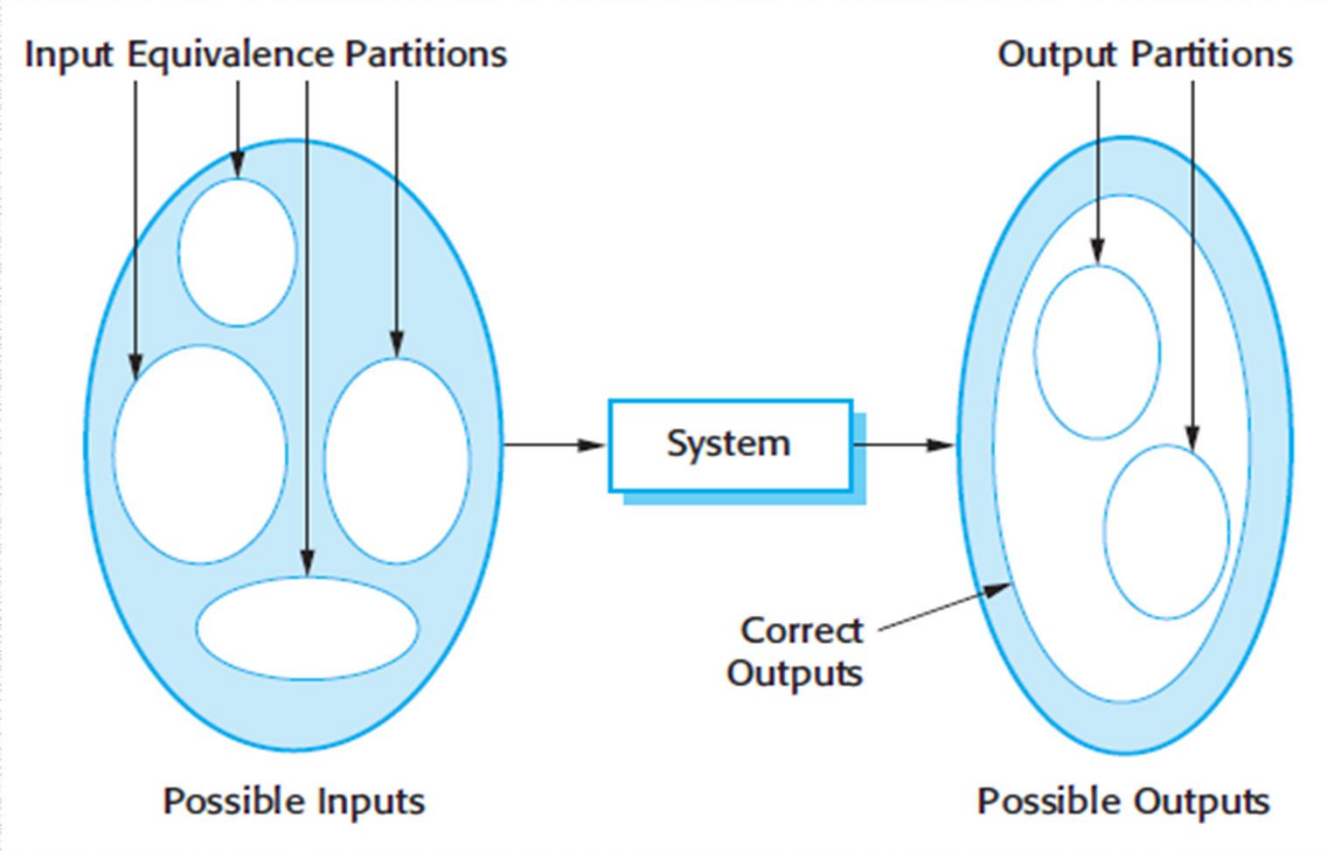
Testing strategies

- Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way.
  - You should choose tests from within each of these groups.
- Guideline-based testing, where you use testing guidelines to choose test cases.
  - These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

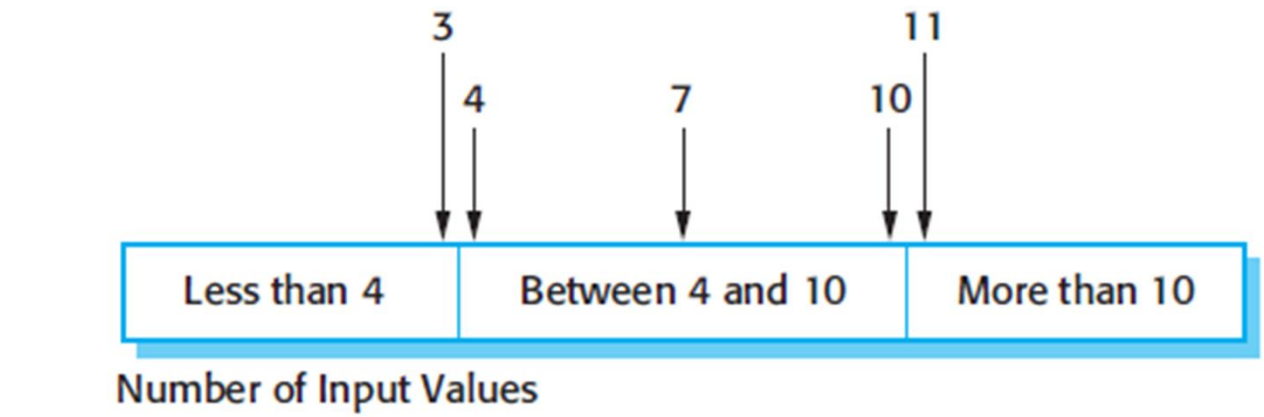
Partition testing

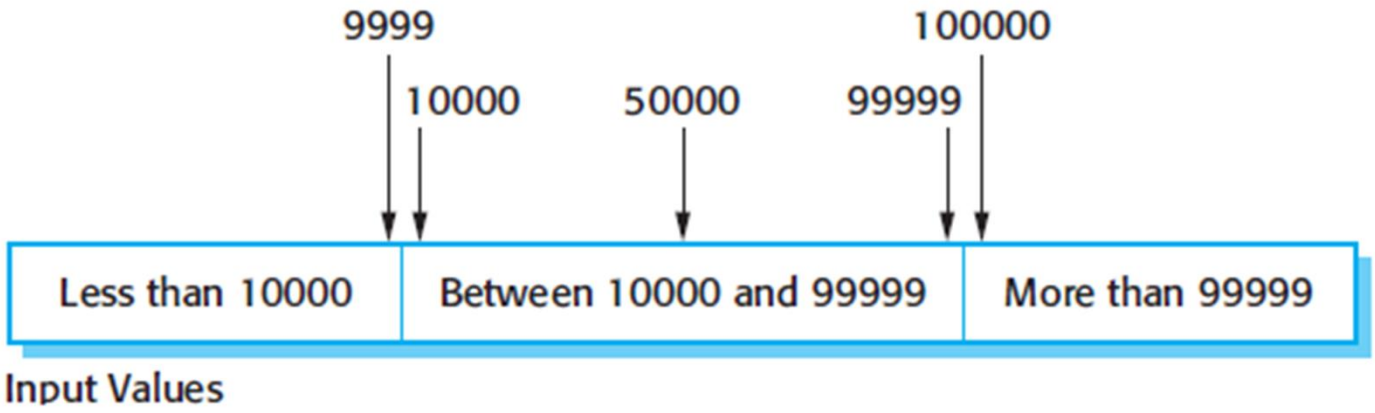
- Input data and output results often fall into different classes where all members of a class are related.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.

Equivalence partitioning



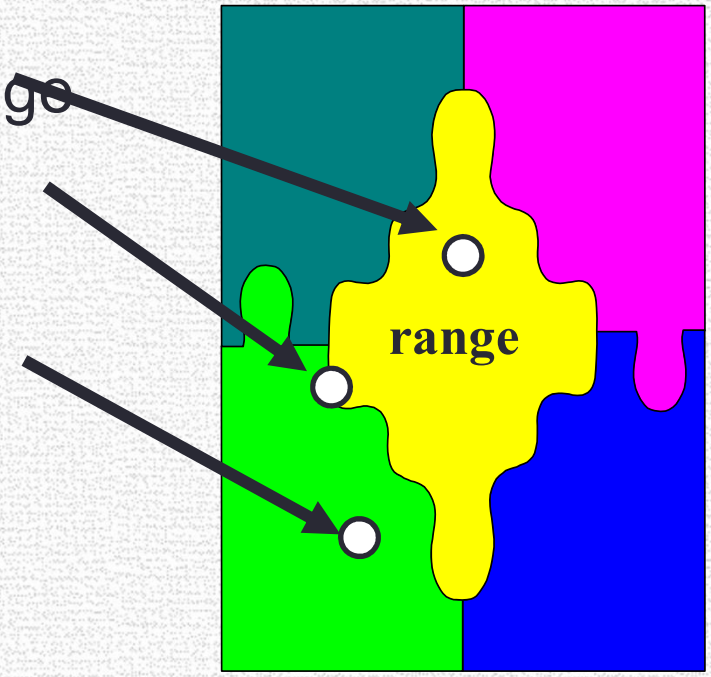
Equivalence partitions





Testing Ranges: Elementary Cases

- 1. within range
- 2. at the boundaries of the range
- 3. outside the range
  - ("illegal")



Testing guidelines (sequences)

- Test software with sequences which have only a single value.
- Use sequences of different sizes in different tests.
- Derive tests so that the first, middle and last elements of the sequence are accessed.
- Test with sequences of zero length.

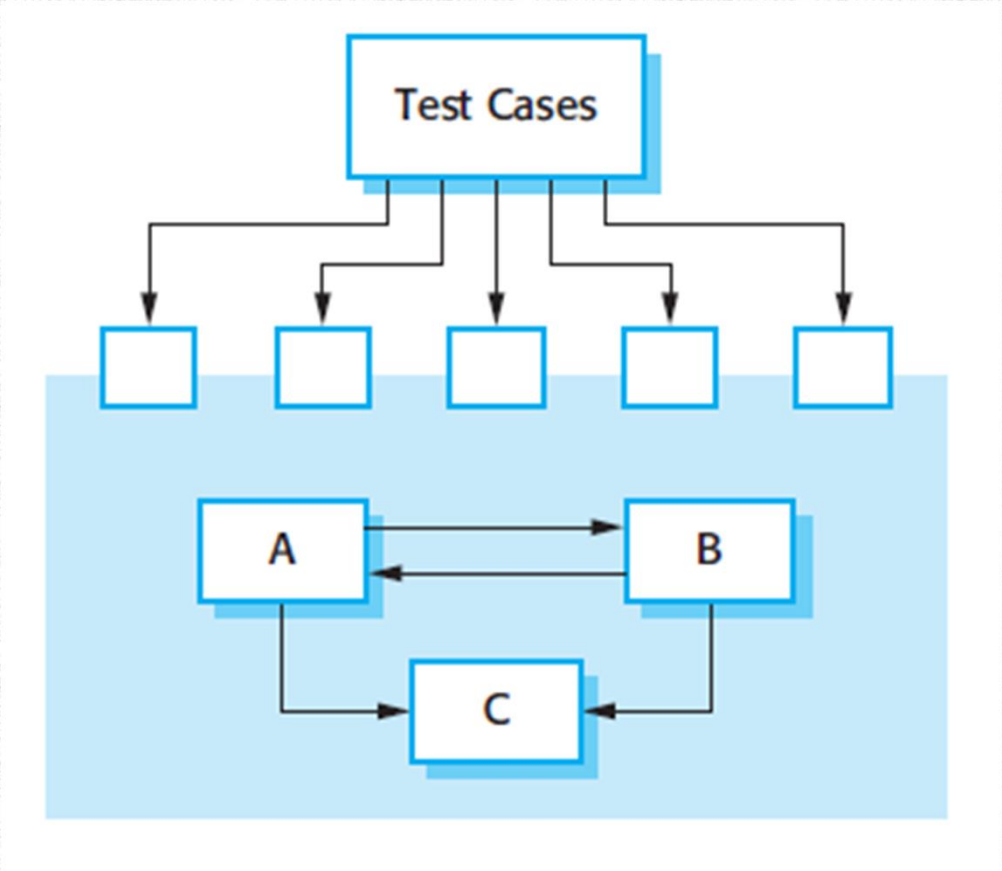
General testing guidelines

- Choose inputs that force the system to generate all error messages
- Design inputs that cause input buffers to overflow
- Repeat the same input or series of inputs numerous times
- Force invalid outputs to be generated
- Force computation results to be too large or too small.

Component testing

- Software components are often composite components that are made up of several interacting objects.
  - For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- You access the functionality of these objects through the defined component interface.
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
  - You can assume that unit tests on the individual objects within the component have been completed.

Interface testing



Interface testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- Interface types
  - Parameter interfaces Data passed from one method or procedure 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

- Interface misuse
  - A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.
- Interface misunderstanding
  - A calling component embeds assumptions about the behaviour of the called component which are incorrect.
- Timing errors
  - The called and the calling component operate at different speeds and out-of-date information is accessed.

Interface testing guidelines

- Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
- Always test pointer parameters with null pointers.
- Design tests which cause the component to fail.
- Use stress testing in message passing systems.
- In shared memory systems, vary the order in which components are activated.

System testing

- System testing during development involves integrating components to create a version of the system and then testing the integrated system.
- The focus in system testing is testing the interactions between components.
- System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- System testing tests the emergent behaviour of a system.

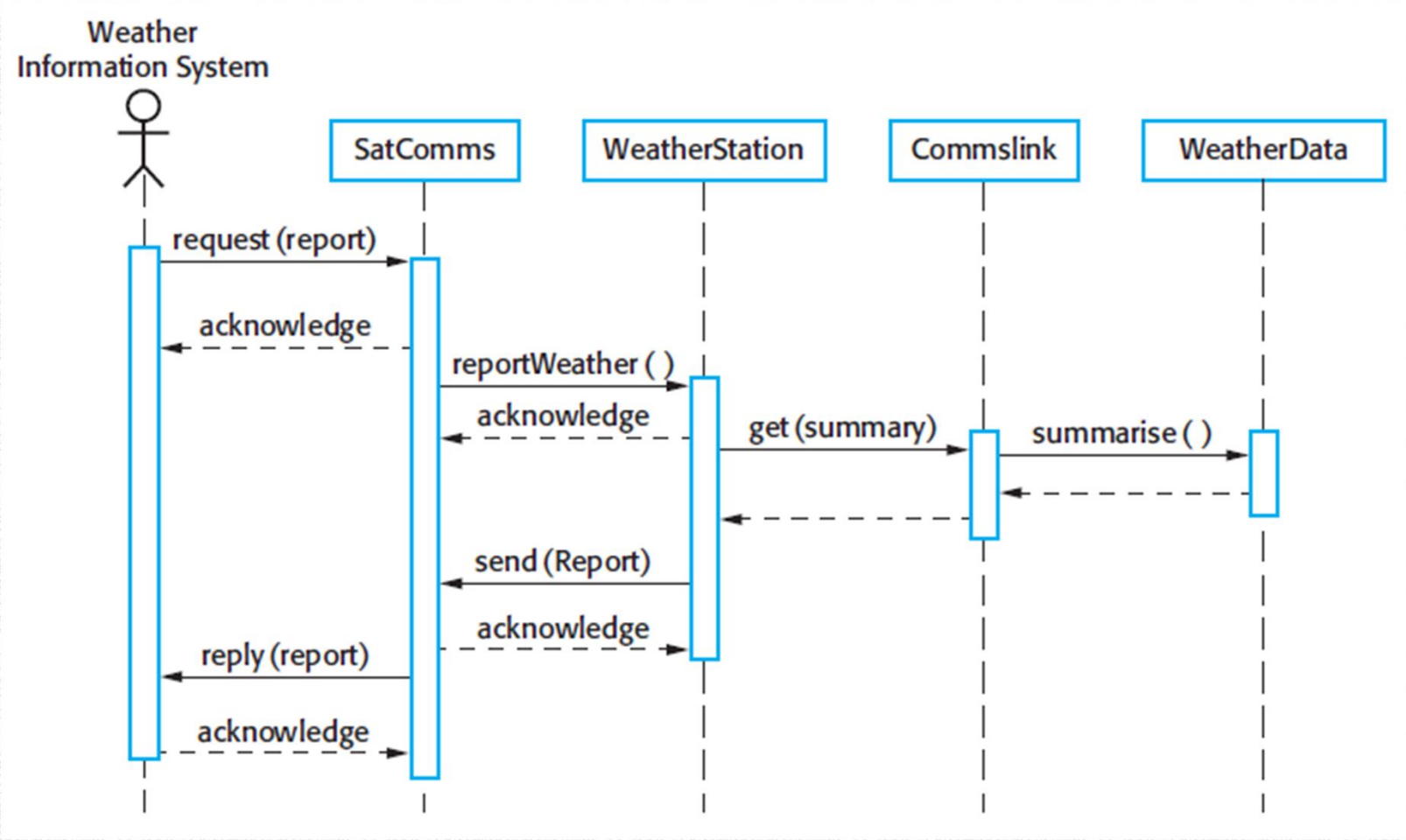
System and component testing

- During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
- Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.
  - In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

Use-case testing

- The use-cases developed to identify system interactions can be used as a basis for system testing.
- Each use case usually involves several system components so testing the use case forces these interactions to occur.
- The sequence diagrams associated with the use case documents the components and interactions that are being tested.

Collect weather data sequence chart



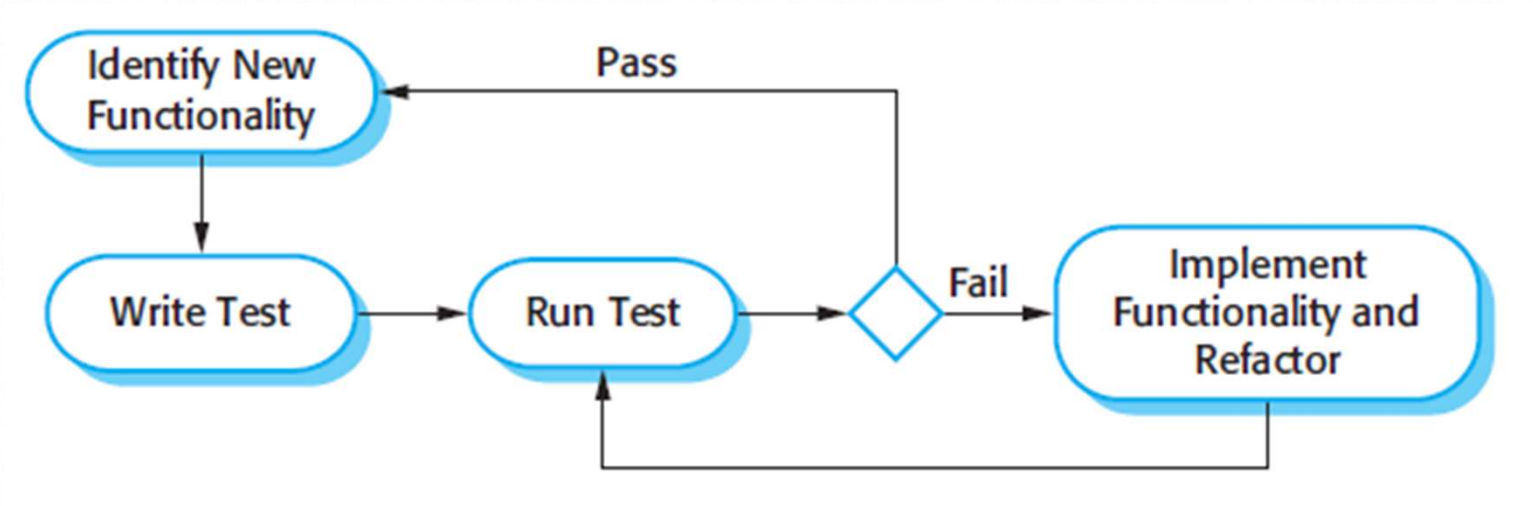
Testing policies

- Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.
- Examples of testing policies:
  - All system functions that are accessed through menus should be tested.
  - Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
  - Where user input is provided, all functions must be tested with both correct and incorrect input.

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.
- TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

Test-driven development



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.

Benefits of test-driven development

- 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.

Regression testing

- Regression testing is testing the system to check that changes have not 'broken' previously working code.
- In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward. All tests are rerun every time a change is made to the program.
- Tests must run 'successfully' before the change is committed.

Release testing

- Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.
- The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.
  - Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- Release testing is usually a black-box testing process where tests are only derived from the system specification.

Release testing and system testing

- Release testing is a form of system testing.
- 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 testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

Types of System Tests

- Volume
  - Subject product to large amounts of input.
- Usability
  - Measure user reaction (e.g., score 1-10).
- Performance
  - Measure speed under various circumstances.
- Configuration
  - Configure to various hardware / software
    - e.g., measure set-up time.
- Compatibility
  - with other designated applications
    - e.g., measure adaptation time.
- Reliability / Availability
  - Measure up-time over extended period.
- Security
  - Subject to compromise attempts.
    - e.g., measure average time to break in.
- Resource usage
  - Measure usage of RAM and disk space etc.
- Install-ability
  - Install under various circumstances.
    - measure time to install.
- Recoverability
  - Force activities that take the application down.
    - measure time to recover
- Serviceability
  - Service application under various situations.
    - measure time to service
- Load / Stress
  - Subject to extreme data & event traffic

Requirements based testing

- Requirements-based testing involves examining each requirement and developing a test or tests for it.
- MHC-PMS 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.

Requirements tests

- 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.
- 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.
- 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.
- Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
- 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.

Features tested by scenario

- Authentication by logging on to the system.
- Downloading and uploading of specified patient records to a laptop.
- Home visit scheduling.
- Encryption and decryption of patient records on a mobile device.
- Record retrieval and modification.
- Links with the drugs database that maintains side-effect information.
- The system for call prompting.

Performance testing

- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Tests should reflect the profile of use of the system.
- Performance tests usually involve planning 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 behaviour.

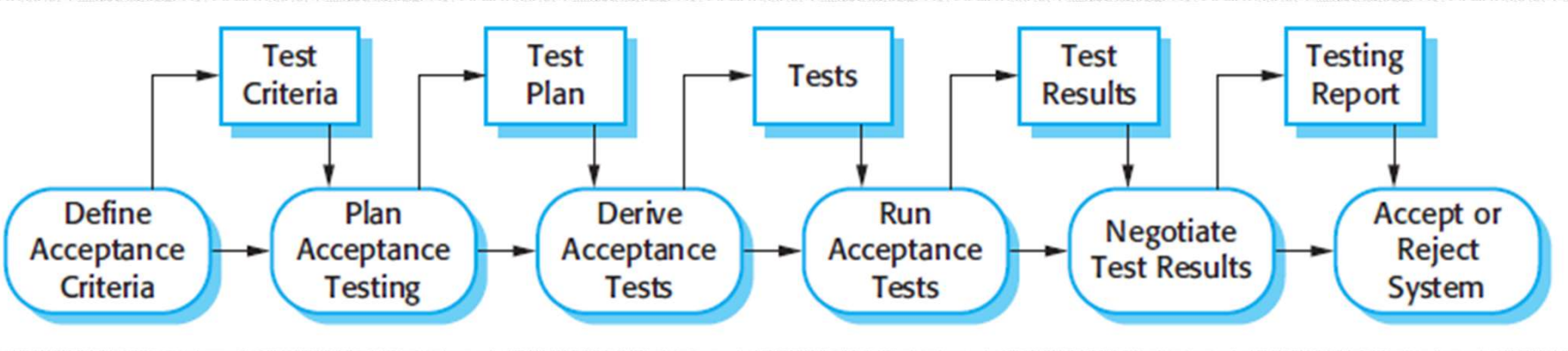
User testing

- User or customer testing is a stage in the testing process in which 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.
  - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

Types of user testing

- Alpha testing
  - Users of the software work with the development team to test the software at the developer's site.
- Beta testing
  - A release of the software is made available to 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.

The acceptance testing process



Stages in the acceptance testing process

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

Alpha- and Beta- Releases

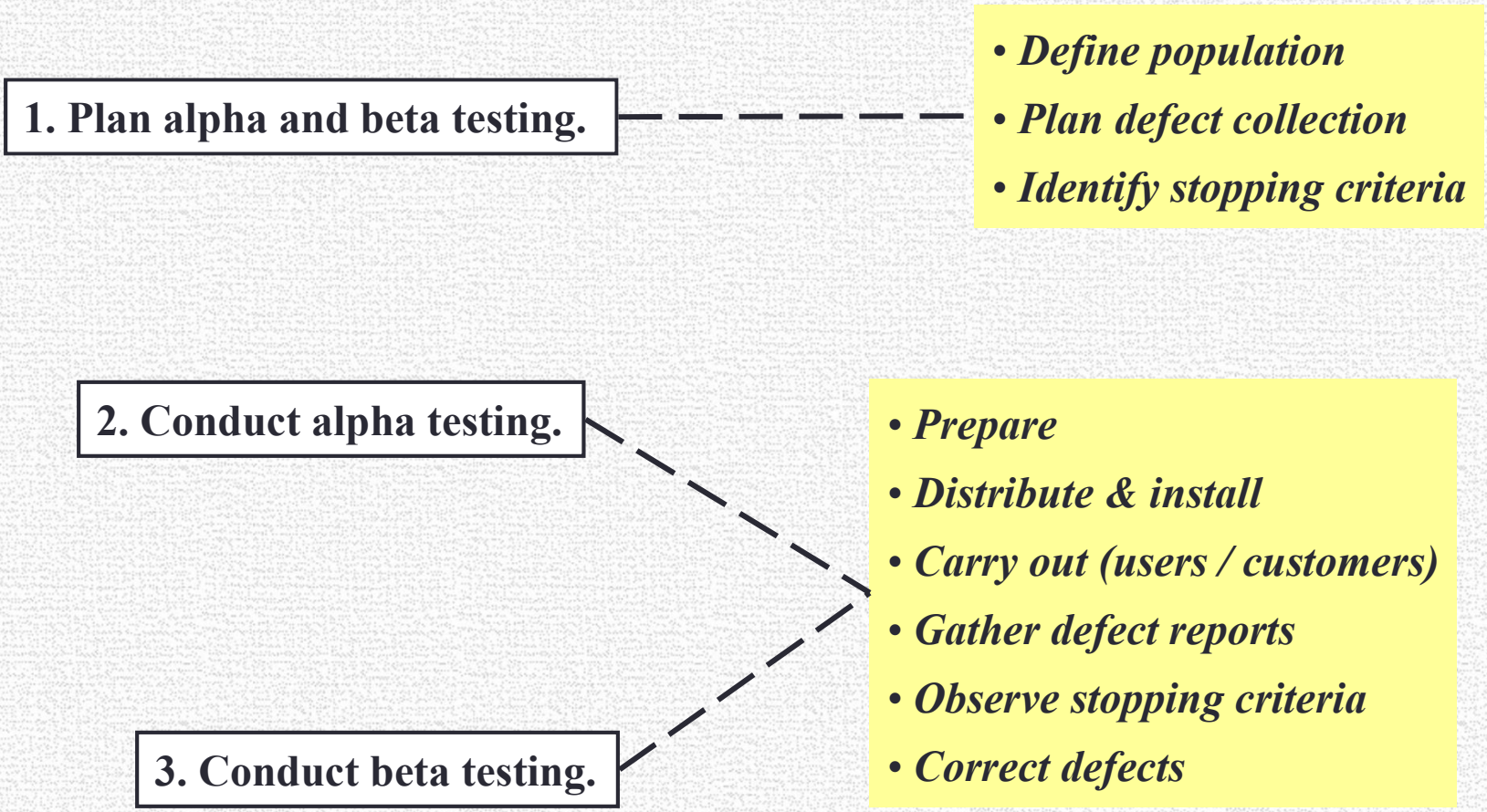
- In-house and highly trusted users
  - Multiplies testing
  - Previews customer reaction
  - Benefits third-party developers
  - Forestalls competition

Alpha

- Selected customers
  - Multiplies testing activity
  - Gets customer reaction

Beta

Roadmap for the Transition Iterations



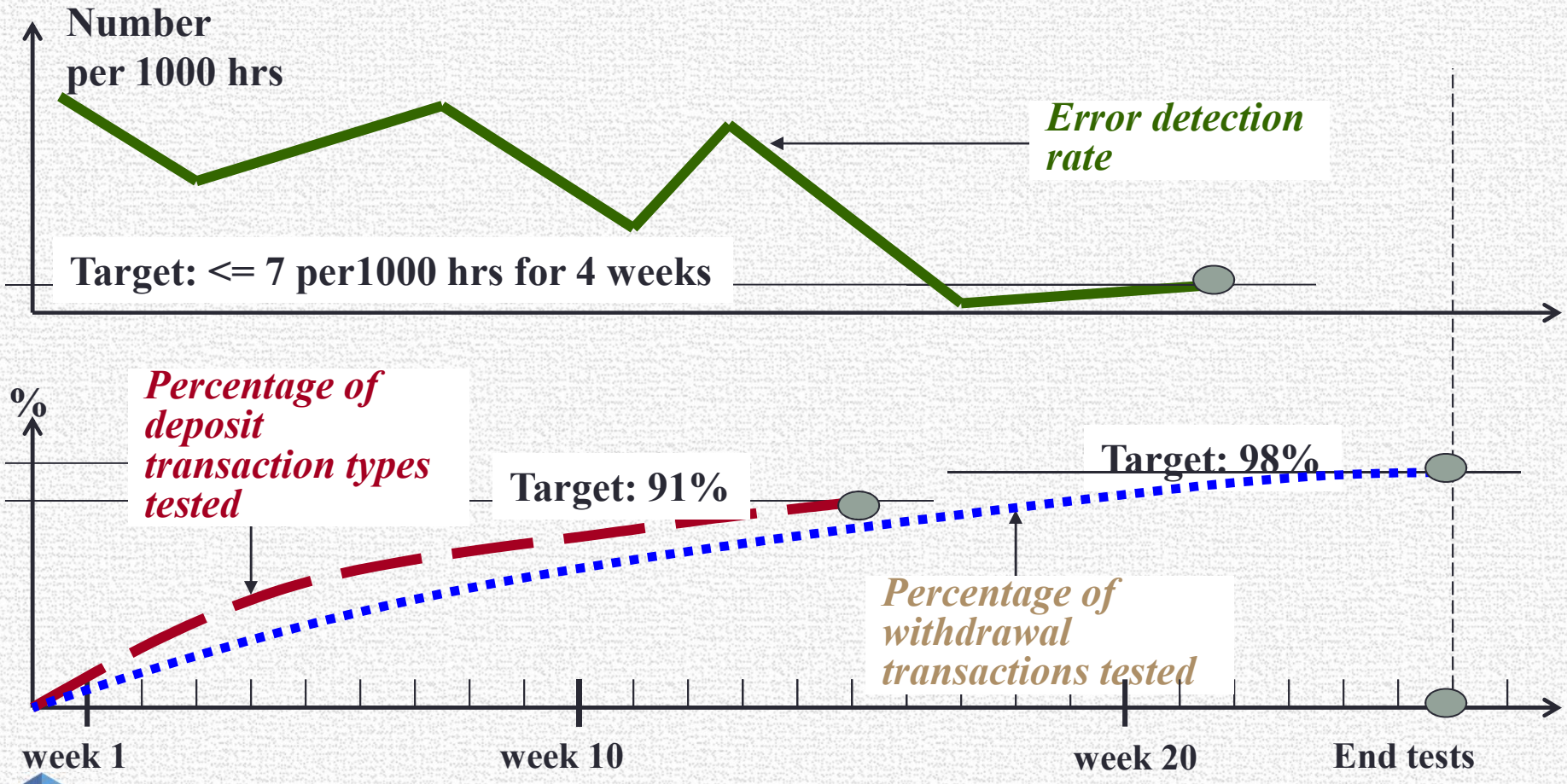
Stopping Criteria

- Completing a particular test methodology
  - Complete the procedures of a method or tool.
- Estimated percent coverage for each category
  - predetermine percent of each & how to calculate
  - e.g., “95% statement coverage”
- Error detection rate
  - predetermine rate with given severity level
  - e.g., “2 medium severity defects or less per 100 hours of operation”
- Total number of errors found
  - (if possible) computed from a percentage of remaining defects
  - predetermine percent
  - e.g., “95% of estimated existing defects found ”

Estimating Remaining Defects: The "seeding" method

- Insert a variety of defects into the application
- Determine the percentage of these which are detected by testers within a given time period
- Estimate the number of defects remaining:
  - By the un-seed defects and the above percentage
- Ex:
  - 50 seed faults are inserted
  - 3 seed faults and 100 un-seed faults are found during the test
  - The remaining undetected seed faults = 50-3 = 47
    - 47/3 = 15.67 undetected (seed) faults per detected (seed) fault
  - The estimated remaining undetected (un-seed) faults:
    - 15.67 * 100 = 1567

Stopping Criteria: Graphical Representation



Summary

- Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.
- 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.
- Development testing includes unit testing, in which you test individual objects and methods component testing in which you test related groups of objects and system testing, in which you test partial or complete systems.

Summary (cont.)

- When testing software, you should try to ‘break’ the software by using experience and guidelines to choose types of test case that have been effective in discovering defects in other systems.
- Wherever possible, you should write automated tests. The tests are embedded in a program that can be run every time a change is made to a system.
- Test-first development is an approach to development where 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 is a user testing process where the aim is to decide if the software is good enough to be deployed and used in its operational environment.