Test Plan, Design and Cases

- A software project test plan is a document that describes the objectives, scope, approach, and focus of a software testing effort.
- The process of preparing a test plan is a useful way to think through the efforts needed to validate the acceptability of a software product.
- The completed document will help people outside the test group understand the 'why' and 'how' of product validation. It should be thorough /complete enough to cover all requirements specifications (functional and non-functional).
- A test plan states what the items to be tested are, at what level they will be tested, what sequence they are to be tested in, how the test strategy will be applied to the testing of each item, and describes the test environment.
- A test plan should ideally be organisation wide, being applicable to all of an organisation's software developments.

Test Plan

- The objective of each test plan is to provide a plan for verification, by testing the software, the software produced fulfils the functional or design statements of the appropriate software specification. In the case of acceptance testing and system testing, this generally means the Functional Specification.
- The first consideration when preparing the Test Plan is who the intended audience is – i.e. the audience for a Unit Test Plan would be different, and thus the content would have to be adjusted accordingly.
- You should begin the test plan as soon as possible. Generally it is desirable to begin the master test plan as the same time the Requirements documents and the Project Plan are being developed. Test planning should have an impact on the Project Plan. Even though plans that are written early will have to be changed during the course of the development and testing, but that is important, because it records the progress of the testing and helps planners become more proficient.

Contents of a Test Plan

- 1. Test Plan Identifier
  2. References
  3. Introduction
  4. Test Items
  5. Software Risk Issues
  6. Features to be Tested
  7. Features not to be Tested
  8. Approach
  9. Item Pass/Fail Criteria
  10. Suspension Criteria and Resumption Requirements
  11. Test Deliverables
  12. Remaining Test Tasks
  13. Environmental Needs
  14. Staffing and Training Needs
  15. Responsibilities
  16. Schedule
  17. Planning Risks and Contingencies
  18. Approvals
  19. Glossary
- **IEEE standards:** 829-1983 IEEE Standard for Software Test Documentation; 1008-1987 IEEE Standard for Software Unit Testing; 1012-1986 IEEE Standard for Software Verification & Validation Plans; 1059-1993 IEEE Guide for Software Verification & Validation Plans; The IEEE website **http://www.ieee.org**

Test Process

- **Organise Project** involves creating a System Test Plan, Schedule & Test Approach, and requesting/assigning resources.
- **Design/Build System Test** involves identifying Test Cycles, Test Cases, Entrance & Exit Criteria, Expected Results, etc. In general, test conditions/expected results will be identified by the Test Team in conjunction with the Project Business Analyst or Business Expert. The Test Team will then identify Test Cases and the Data required.
- **Design/Build Test Procedures** includes setting up procedures such as Error Management systems and Status reporting, and setting up the data tables for the Automated Testing Tool.
- **Build Test Environment** includes requesting/building hardware, software and data set-ups.
- **Execute Project Integration Test** –
- **Execute Acceptance Test and Signoff** .

Test Scope

- Outlined below are the main test types that will be performed for this release.
- All system test plans and conditions will be developed from the functional specification and the requirements catalogue.
- EXAMPLE:
- **Functional Testing**
- The objective of this test is to ensure that each element of the application meets the functional requirements of the business as outlined in the : Requirements Catalogue
- **Integration Testing**
- This test proves that all areas of the system interface with each other correctly and that there are no gaps in the data flow. Final Integration Test proves that system works as integrated unit when all the fixes are complete.
- **Business (User) Acceptance Test**
- This test, which is planned and executed by the Business Representative(s), ensures that the system operates in the manner expected, and any supporting material such as procedures, forms etc. are accurate and suitable for the purpose intended. It is high level testing, ensuring that there are no gaps in functionality.
- **Performance Testing**
- These tests ensure that the system provides acceptable response times (which should not exceed 4 seconds).
- **Other Non-Functional Testing …..**

Test Schedule and Resources

- Use Gantt chart to show the sequence of testing activities, dependencies between tests and milestones to be reached during the testing process.
- Resources:
  - Human – who is needed during the testing process: For example, business analyst, testers, technical support, programmers, business representative and test controller.
  - Hardware needed to do the testing: For example, One separate, controlled system will be required for the initial phase
    of testing, setup as per one standard, complete office environment.
    In order to maintain the integrity of the test environment his network will not be accessible to anybody outside this project. The printers are also exclusively for use by the test network.
  - Software needed to do the testing: For example, test environments and software such as X operating system, Y run time files, Z Netware.

Roles and Responsibilities

- Management Team: Project leader, Software Quality Assurance Leader, …
- Testing Team: Test Planner and Controllers, Testers.
- Business Team : Business Analyst and Representatives.
- Testing Support Team: Support Programmers
- External Support Team: Technical and Access Support
- Project Leader regularly review Testing progress with Test Controller.
- Software Quality Assurance Leader ensures testing process and deliverables are delivered to schedule, budget & quality. They also regularly review Testing progress , Manage issues/risks relating to System Test Team and Provide resources necessary for completing system test.
- Test Planner and Controllers produce High Level and Detailed Test Conditions, report progress at regular status reporting meetings, Co-ordinate review & signoff of Test Conditions and Manage individual test cycles & resolve tester queries/problems.

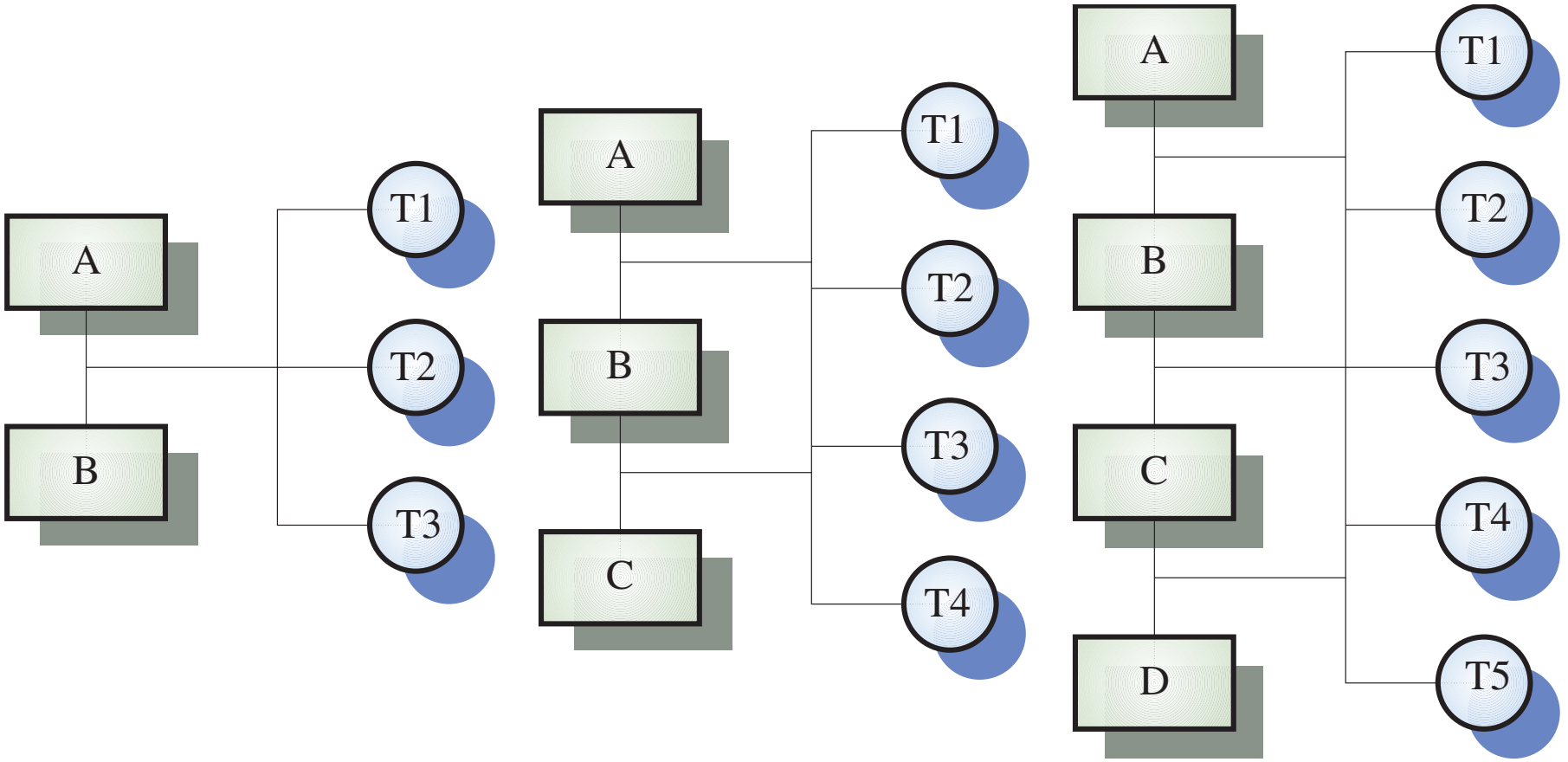
Test case

- **unique-test-case-id: Test Case Title**
- **Type of Test: Unit, Integration, Black, White, Path, Scenario, Stress , …**
- **Purpose:** Short sentence or two about the aspect of the system is being tested.
- **Prerequisite** Assumptions that must be met before the test case can be run. E.g., "logged in", "guest login allowed".
- **Test Data / Input Data / Entry Criteria**: List of variables and their possible values used in the test case. You can list specific values or describe value ranges. The test case should be performed once for each *combination* of values. These values are written in set notation, one per line. E.g.: loginID = {Valid loginID, invalid loginID, valid email, invalid email, empty} password = {valid, invalid, empty}
- **Steps**: Steps to carry out the test.
  - visit LoginPage and enter userID
  - enter password and click login
  - see the terms of use page
  - click agree radio button at page bottom
  - click submit button and see Personal Page
  - verify that welcome message is correct username
- **Output (Expected and Actual):**
- **Exit Criteria: PASS/FAIL**
- **Recommendations:**
- **Notes, Issue and Questions**:

Integration testing

- Tests complete systems or subsystems composed of integrated components
- Integration testing should be black-box testing with tests derived from the specification
- Main difficulty is localising errors
- Incremental integration testing reduces this problem

Incremental integration testing



Test sequence 1

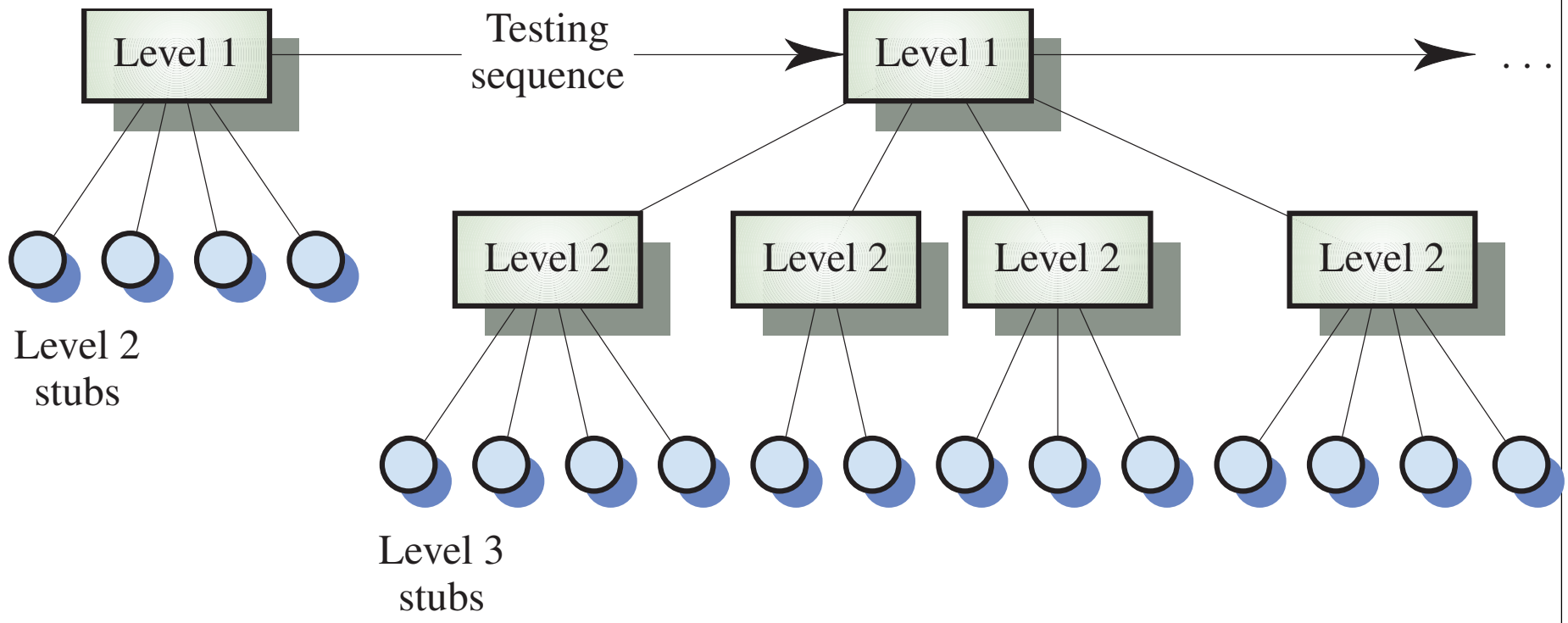
Test sequence 2

Test sequence 3

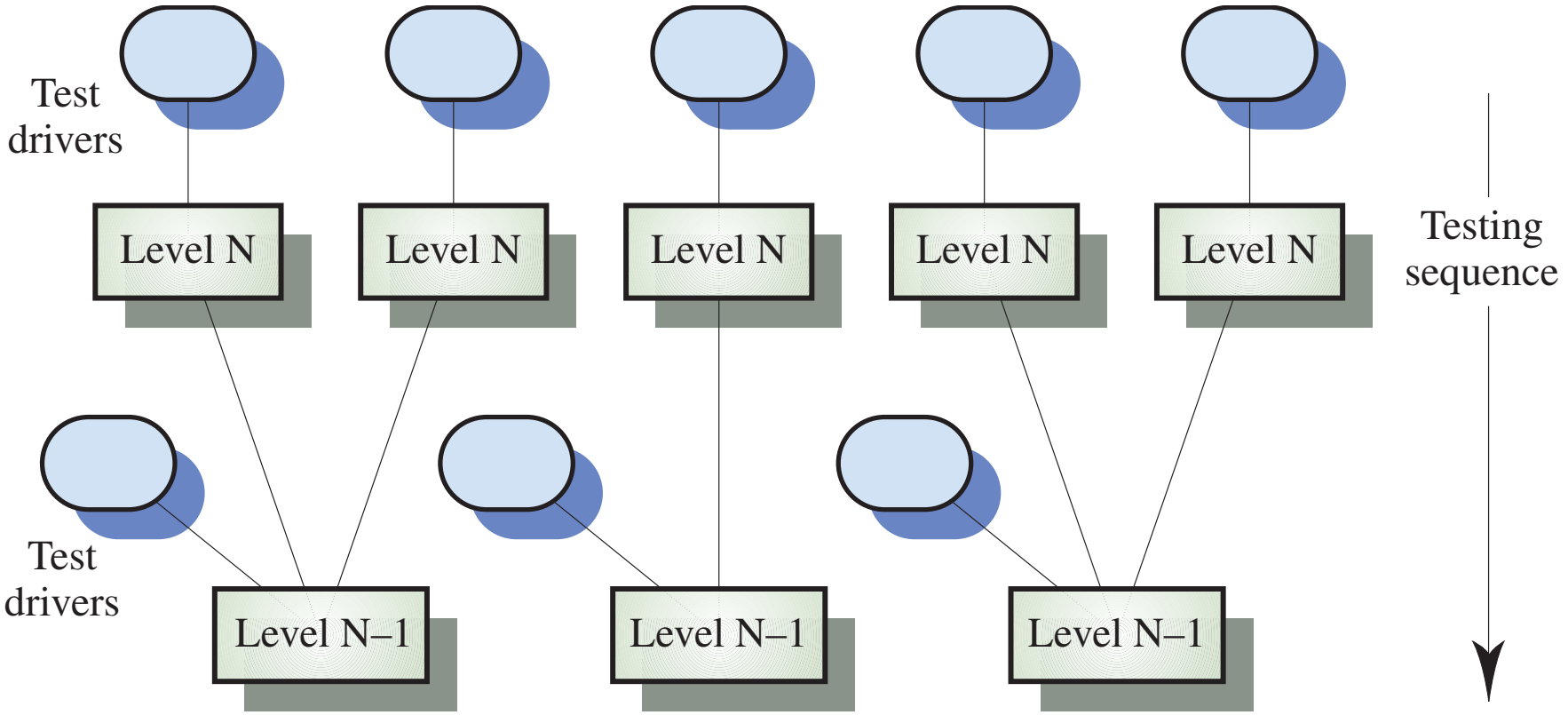
Approaches to integration testing

- Top-down testing
  - Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- Bottom-up testing
  - Integrate individual components in levels until the complete system is created
- In practice, most integration involves a combination of these strategies

Top-down testing



Bottom-up testing

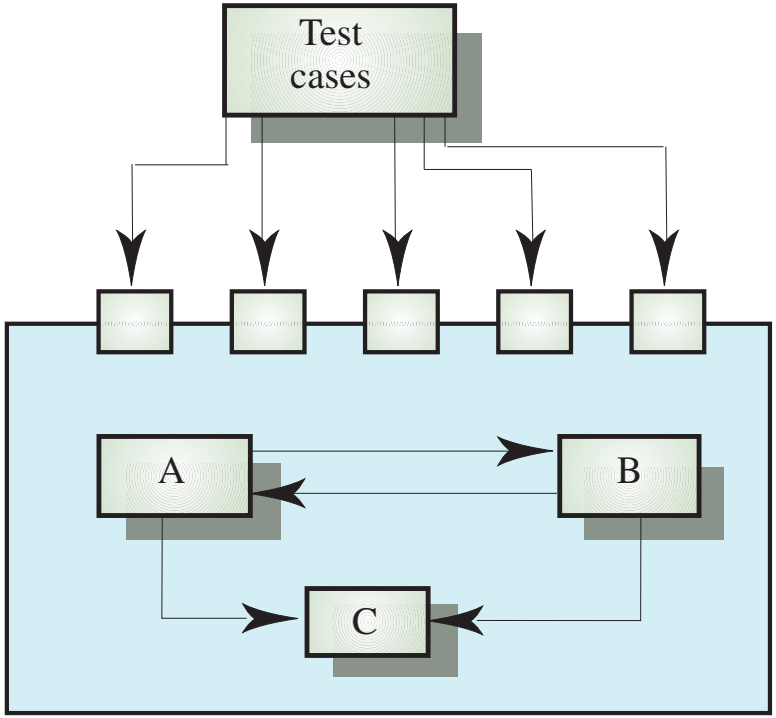


Testing approaches

- Architectural validation
  - Top-down integration testing is better at discovering errors in the system architecture
- System demonstration
  - Top-down integration testing allows a limited demonstration at an early stage in the development
- Test implementation
  - Often easier with bottom-up integration testing
- Test observation
  - Problems with both approaches. Extra code may be required to observe tests

Interface testing

- Takes place when modules or sub-systems are integrated to create larger systems
- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces
- Particularly important for object-oriented development as objects are defined by their interfaces



Interfaces types

- Parameter interfaces
  - Data passed from one procedure to another
- Shared memory interfaces
  - Block of memory is shared between procedures
- 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

Testing, Stress testing

- Exercises the system beyond its maximum design load. Stressing the system often causes defects to
  come to light
- Stressing the system test failure behaviour.. Systems should not fail catastrophically. Stress testing checks for unacceptable loss of service or data
- Particularly relevant to distributed systems
  which can exhibit severe degradation as a
  network becomes overloaded

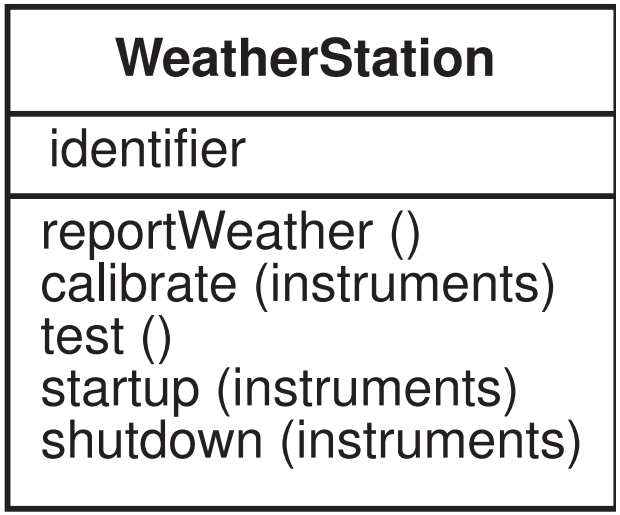
Object-oriented testing

- The components to be tested are object classes that are instantiated as objects
- Larger grain than individual functions so approaches to white-box testing have to be extended
- No obvious ‘top’ to the system for top-down integration and testing
- Testing operations associated with objects
- Testing object classes
- Testing clusters of cooperating objects
- Testing the complete OO system

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

Weather station object interface



- Test cases are needed for all operations
- Use a state model to identify state transitions for testing
- Examples of testing sequences
  - Shutdown → Waiting → Shutdown
  - Waiting → Calibrating → Testing → Transmitting → Waiting
  - Waiting → Collecting → Waiting → Summarising → Transmitting → Waiting

Object integration

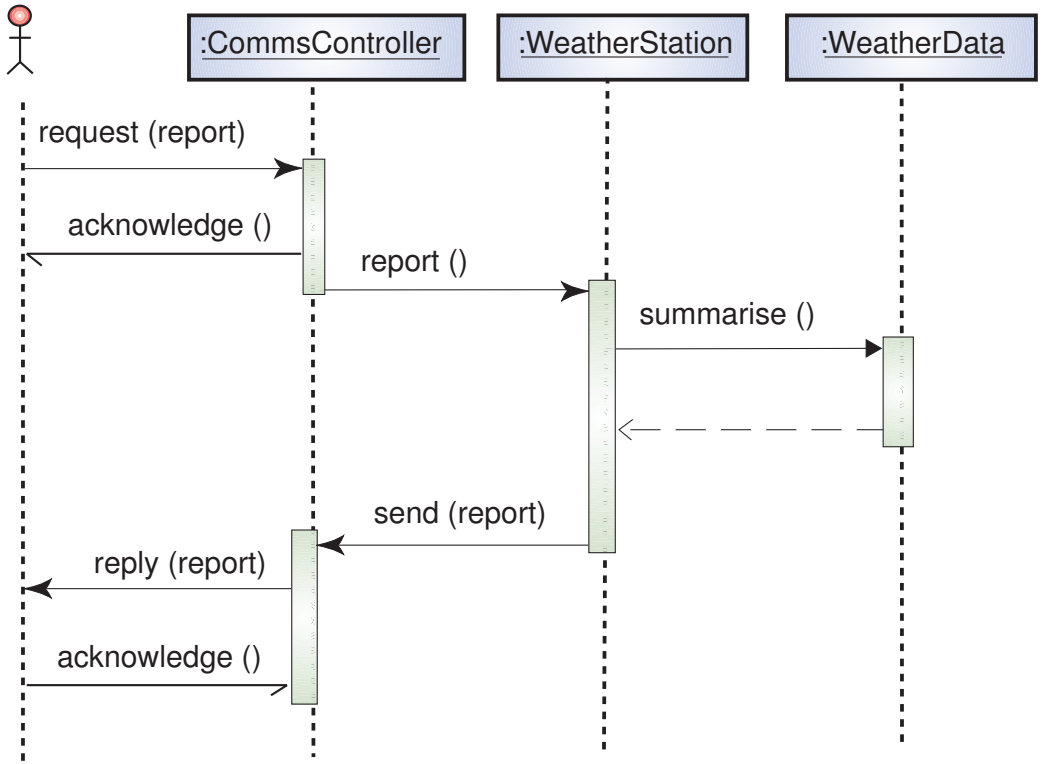
- Levels of integration are less distinct in object-oriented systems
- Cluster testing is concerned with integrating and testing clusters of cooperating objects
- Identify clusters using knowledge of the operation of objects and the system features that are implemented by these clusters
- PLAN:
- To test object classes, test all operations, attributes and states
- Integrate object-oriented systems around clusters of objects and test clusters.

Approaches to cluster testing

- Use-case or scenario testing
  - Testing is based on a user interactions with the system
  - Has the advantage that it tests system features as experienced by users
- Thread testing
  - Tests the systems response to events as processing threads through the system
- Object interaction testing
  - Tests sequences of object interactions that stop when an object operation does not call on services from another object

Scenario-based testing

- Identify scenarios from use-cases and supplement these with interaction diagrams that show the objects involved in the scenario
- Consider the scenario in the weather station system where a report is generated



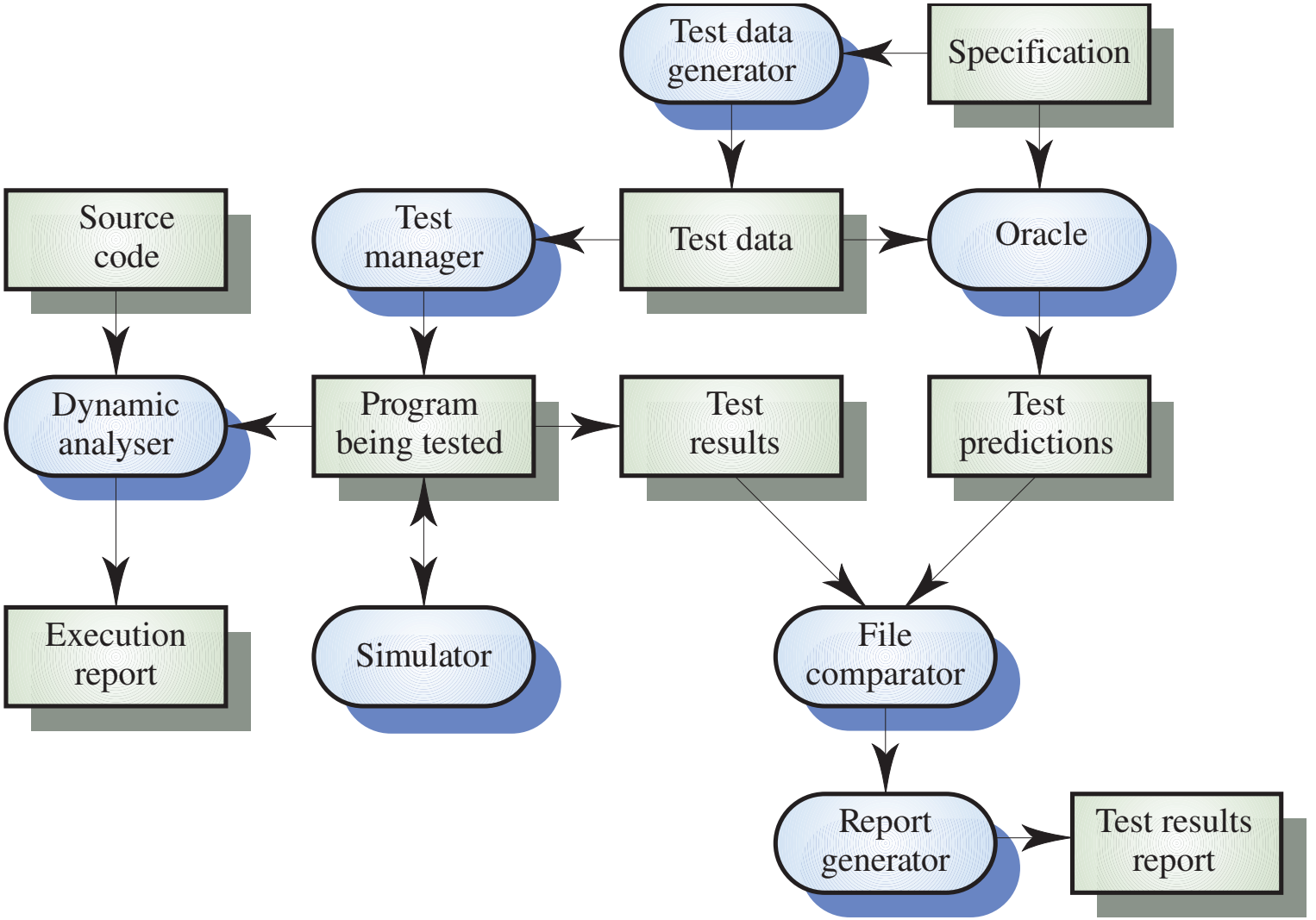
Weather station testing

- Thread of methods executed
  - CommsController:request → WeatherStation:report → WeatherData:summarise
- Inputs and outputs
  - Input of report request with associated acknowledge and a final output of a report
  - Can be tested by creating raw data and ensuring that it is summarised properly
  - Use the same raw data to test the WeatherData object

Testing workbenches

- Testing is an expensive process phase. Testing workbenches provide a range of tools to reduce the time required and total testing costs
- Most testing workbenches are open systems because testing needs are organisation-specific
- Difficult to integrate with closed design and analysis workbenches
- ADAPTATION:
- Scripts may be developed for user interface simulators and patterns for test data generators
- Test outputs may have to be prepared manually for comparison
- Special-purpose file comparators may be developed

A testing workbench



List of Different Types of Testing

- Black box testing - not based on any knowledge of internal design or code. Tests are based on requirements and functionality.
- White box testing - based on knowledge of the internal logic of an application's code. Tests are based on coverage of code statements, branches, paths, conditions.
- unit testing - the most 'micro' scale of testing; to test particular functions or code modules. Typically done by the programmer and not by testers, as it requires detailed knowledge of the internal program design and code. Not always easily done unless the application has a well-designed architecture with tight code; may require developing test driver modules or test harnesses.
- incremental integration testing - continuous testing of an application as new functionality is added; requires that various aspects of an application's functionality be independent enough to work separately before all parts of the program are completed, or that test drivers be developed as needed; done by programmers or by testers.
- integration testing - testing of combined parts of an application to determine if they function together correctly. The 'parts' can be code modules, individual applications, client and server applications on a network, etc. This type of testing is especially relevant to client/server and distributed systems.
- functional testing - black-box type testing geared to functional requirements of an application; this type of testing should be done by testers. This doesn't mean that the programmers shouldn't check that their code works before releasing it (which of course applies to any stage of testing.)
- system testing - black-box type testing that is based on overall requirements specifications; covers all combined parts of a system.
- end-to-end testing - similar to system testing; the 'macro' end of the test scale; involves testing of a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.
- sanity testing or smoke testing - typically an initial testing effort to determine if a new software version is performing well enough to accept it for a major testing effort. For example, if the new software is crashing systems every 5 minutes, bogging down systems to a crawl, or corrupting databases, the software may not be in a 'sane' enough condition to warrant further testing in its current state.

Different Types of Testing

- regression testing - re-testing after fixes or modifications of the software or its environment. It can be difficult to determine how much re-testing is needed, especially near the end of the development cycle. Automated testing approaches can be especially useful for this type of testing.
- acceptance testing - final testing based on specifications of the end-user or customer, or based on use by end-users/customers over some limited period of time.
- load testing - testing an application under heavy loads, such as testing of a web site under a range of loads to determine at what point the system's response time degrades or fails.
- stress testing - term often used interchangeably with 'load' and 'performance' testing. Also used to describe such tests as system functional testing while under unusually heavy loads, heavy repetition of certain actions or inputs, input of large numerical values, large complex queries to a database system, etc.
- performance testing - term often used interchangeably with 'stress' and 'load' testing. Ideally 'performance' testing (and any other 'type' of testing) is defined in requirements documentation or QA or Test Plans.
- usability testing - testing for 'user-friendliness'. Clearly this is subjective, and will depend on the targeted end-user or customer. User interviews, surveys, video recording of user sessions, and other techniques can be used. Programmers and testers are usually not appropriate as usability testers.
- install/uninstall testing - testing of full, partial, or upgrade install/uninstall processes.
- recovery testing - testing how well a system recovers from crashes, hardware failures, or other catastrophic problems.
- failover testing - typically used interchangeably with 'recovery testing'
- security testing - testing how well the system protects against unauthorized internal or external access, willful damage, etc; may require sophisticated testing techniques.

Different Types of Testing

- compatibility testing - testing how well software performs in a particular hardware/software/operating system/network/etc. environment.
- exploratory testing - often taken to mean a creative, informal software test that is not based on formal test plans or test cases; testers may be learning the software as they test it.
- ad-hoc testing - similar to exploratory testing, but often taken to mean that the testers have significant understanding of the software before testing it.
- context-driven testing - testing driven by an understanding of the environment, culture, and intended use of software. For example, the testing approach for life-critical medical equipment software would be completely different than that for a low-cost computer game.
- user acceptance testing - determining if software is satisfactory to an end-user or customer.
- comparison testing - comparing software weaknesses and strengths to competing products.
- alpha testing - testing of an application when development is nearing completion; minor design changes may still be made as a result of such testing. Typically done by end-users or others, not by programmers or testers.
- beta testing - testing when development and testing are essentially completed and final bugs and problems need to be found before final release. Typically done by end-users or others, not by programmers or testers.
- mutation testing - a method for determining if a set of test data or test cases is useful, by deliberately introducing various code changes ('bugs') and retesting with the original test data/cases to determine if the 'bugs' are detected. Proper implementation requires large computational resources