CSCD306-DCIT208: SOFTWARE ENGINEERING

Mark Atta Mensah | Department of Computer Science | 2020-2021

UNIVERSITY OF GHANA

INTEGRI PROCEDAMUS

Session 4

Design and implementation | Software testing

Reference: Sommerville, Software Engineering, 10 ed., Chapters 7 & 8

The big picture

- Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- ***Software design*** is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
- ***Implementation*** is the process of realizing the design as a program. These two activities are invariably inter-leaved.

The big picture

In a wide range of domains, it is now possible to buy ***commercial off-the-shelf systems*** (COTS) that can be adapted and tailored to the users' requirements.

When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

Object-oriented design using the UML

- Structured object-oriented design processes involve developing a number of different ***system models***. They require a lot of effort for development and maintenance and, for small systems, this may not be cost-effective.
- However, for ***large systems*** developed by different groups design models are an important communication mechanism.

Object-oriented design using the UML

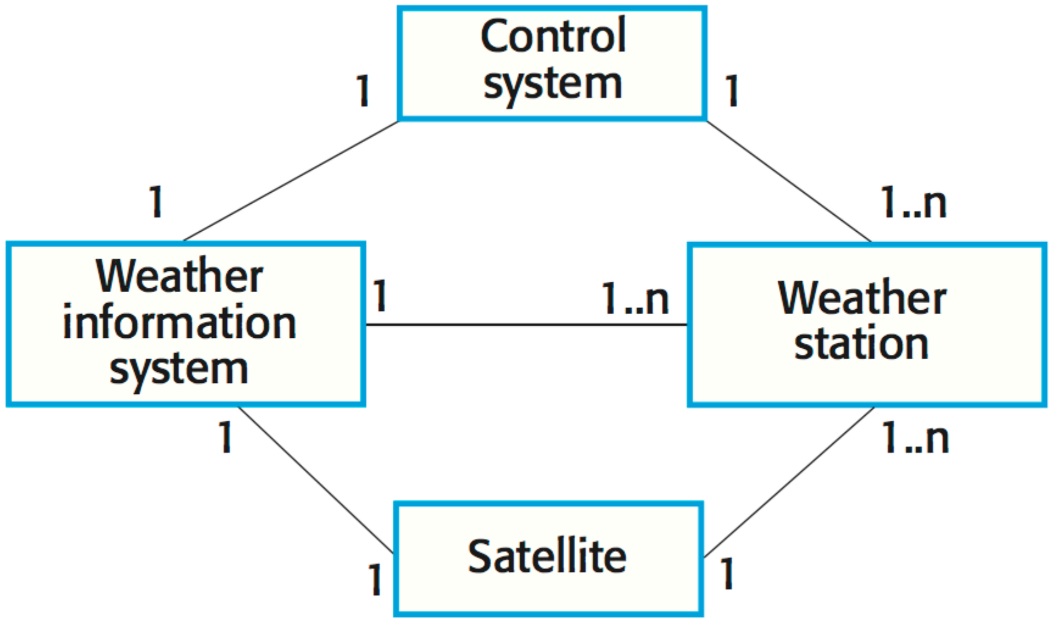
Common activities in these processes include:

- Define the context and modes of use of the system;
- Design the system architecture;
- Identify the principal system objects;
- Develop design models;
- Specify object interfaces.

System context and interactions

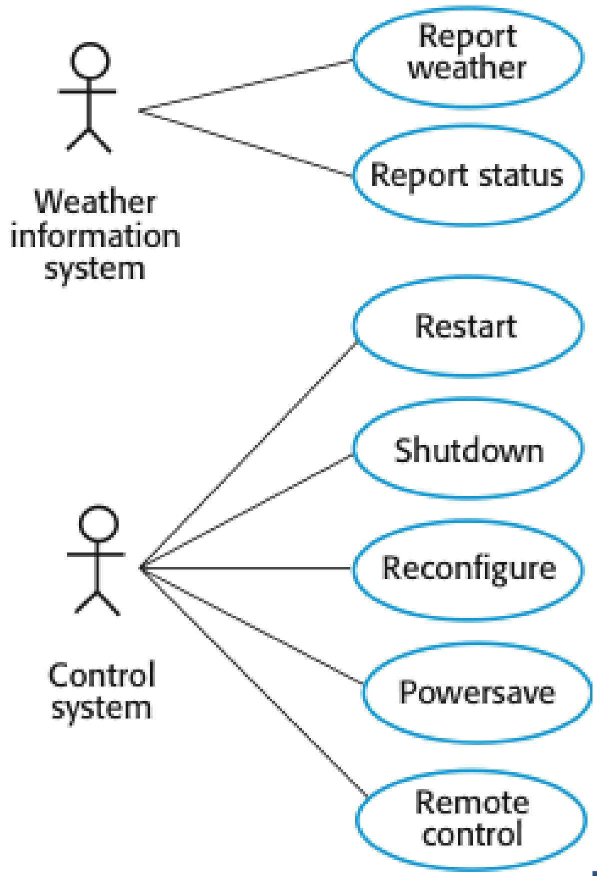
- Understanding the relationships between the software that is being designed and its ***external environment*** is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment.
- Understanding of the context also lets you establish the ***boundaries*** of the system. Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

System context and interactions



A system ***context*** is a ***structural model*** (e.g., a class diagram) that demonstrates the other systems in the environment of the system being developed.

An ***interaction*** model is a ***dynamic mode***l (e.g., a use case diagram + structured natural language description) that shows how the system interacts with its environment as it is used.

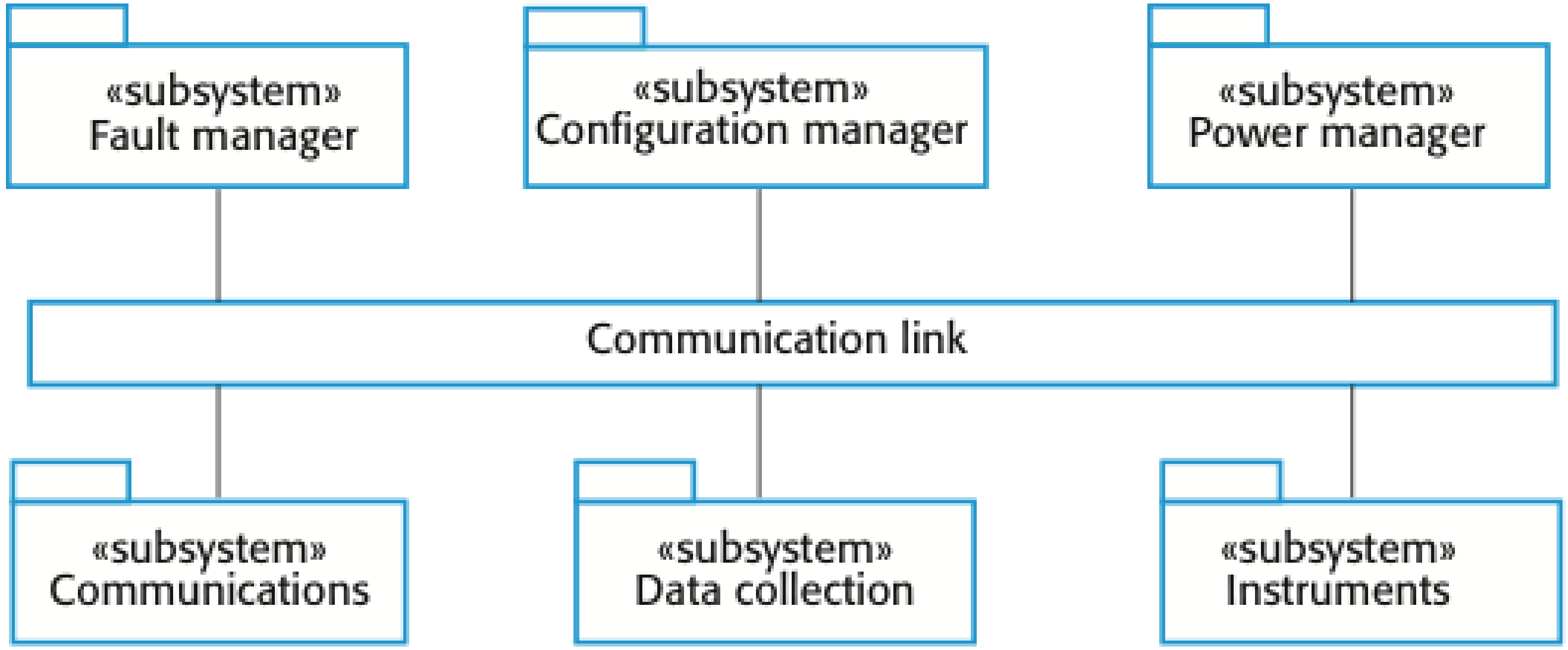


System context and interactions

System	**Weather station**
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.

Architectural design

Once interactions between the system and its environment have been understood, you use this information for designing the system architecture. You identify the ***major components*** that make up the system and ***their interactions***, and then may organize the components using an architectural pattern (e.g. a layered or client-server model).


«subsystem» Fault manager
«subsystem» Configuration manager
«subsystem» Power manager
Communication link
«subsystem» Communications
«subsystem» Data collection
«subsystem» Instruments

Architectural design

Identifying object classes is often a difficult part of object oriented design. There is no 'magic formula' for object identification. It relies on the skill, experience and domain knowledge of system designers. Object identification is an iterative process. You are unlikely to get it right first time.

Architectural design

Approaches to object identification include:

- Use a ***grammatical approach*** based on a natural language description of the system.
- Base the identification ***on tangible things*** in the application domain.
- Use a ***behavioral approach*** and identify objects based on what participates in what behavior.
- Use a ***scenario-based analysis***. The objects, attributes and methods in each scenario are identified.

Design models

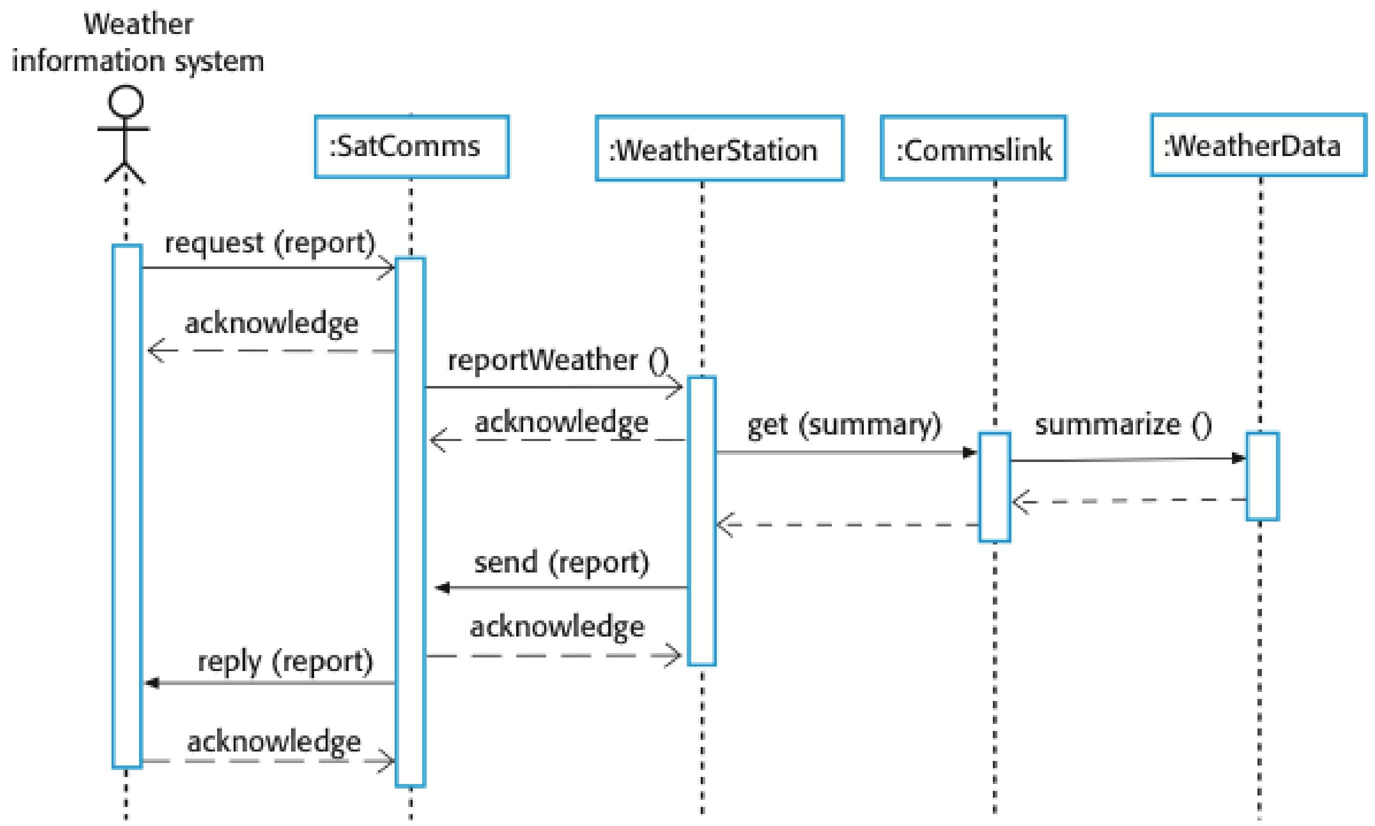
Design models show the objects and object classes and relationships between these entities.

Static models describe the static structure of the system in terms of object classes and relationships.

Dynamic models describe the dynamic interactions between objects.

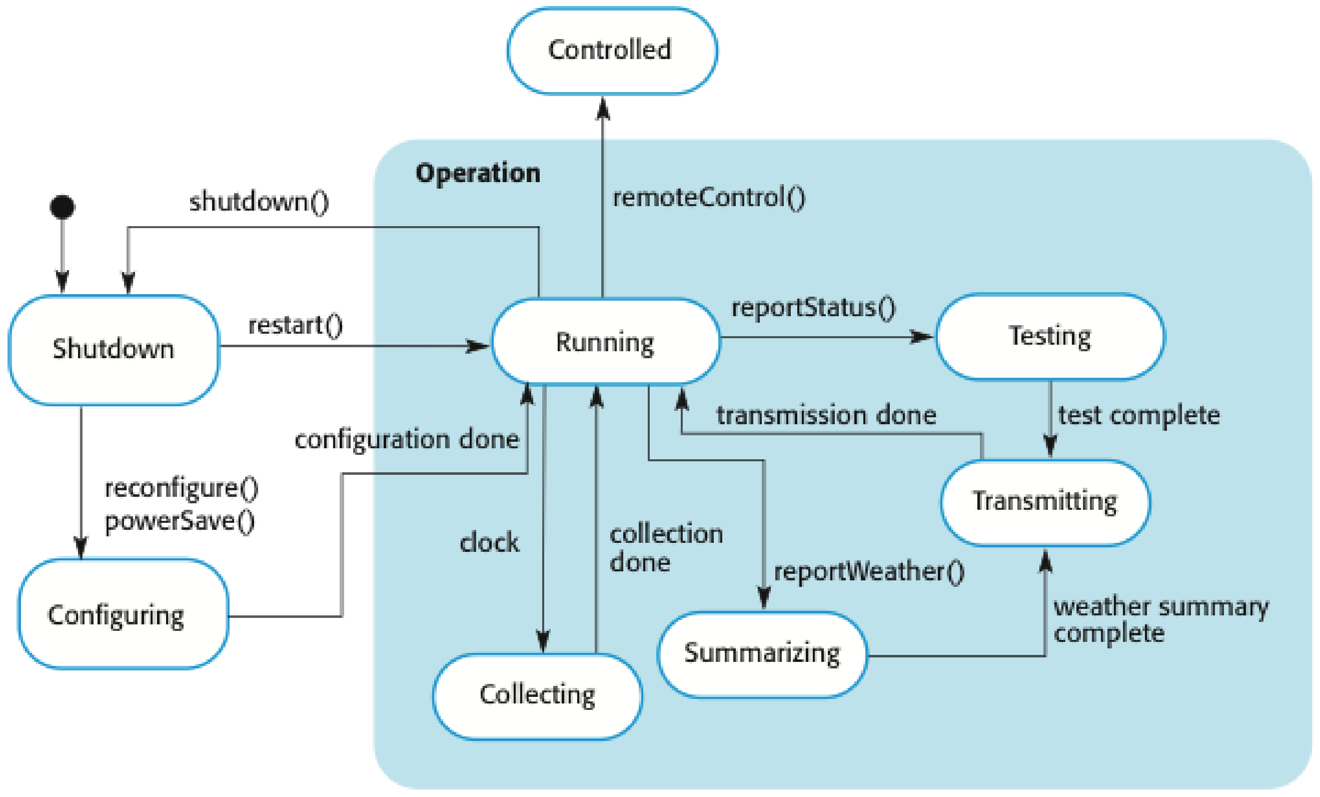
Subsystem models show logical groupings of objects into coherent subsystems. These are represented using a form of class diagram with each subsystem shown as a package with enclosed objects. Subsystem models are static (structural) models.

Design models



Sequence models show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic models.

Design models



State machine models show how individual objects change their state in response to events. These are represented in the UML using state diagrams. State machine models are dynamic models. State diagrams are useful high-level models of a system or an object's run-time behavior.

Interface specification

Object interfaces have to be specified so that the objects and other components can be ***designed in parallel***. Designers should avoid designing the interface representation but should hide this in the object itself. Objects may have several interfaces which are viewpoints on the methods provided. The UML uses class diagrams for interface specification but Java may also be used.

Design patterns

A design pattern is a way of ***reusing abstract knowledge*** about a problem and its solution. A pattern is a description of the problem and the essence of its solution. It should be sufficiently abstract to be reused in different settings. Pattern descriptions usually make use of object-oriented characteristics such as inheritance and polymorphism.

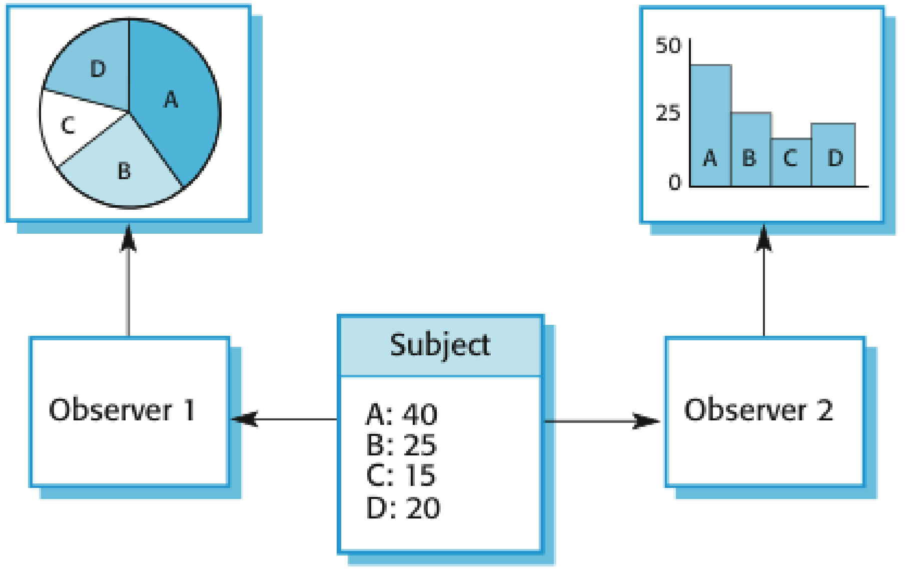
Design patterns

Design pattern elements:

- ***Name -*** A meaningful pattern identifier
- ***Problem description -*** A common situation where this pattern is applicable
- ***Solution description -*** Not a concrete design but a template for a design solution that can be instantiated in different ways
- ***Consequences -*** The results and trade-offs of applying the pattern

Design patterns: *(Example: the Observer pattern)*

Pattern name	**Observer**
Description	Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.
Problem description	In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated. This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.
Solution description	This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed. The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
Consequences	The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.



Reuse

From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language. The only significant reuse or software was the reuse of functions and objects in programming language libraries. Costs and schedule pressure mean that this approach became increasingly unviable, especially for commercial and Internet-based systems. An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software.

Reuse: *Levels*

- The ***abstraction*** level: don't reuse software directly but use knowledge of successful abstractions in the software design.
- The ***object*** level: directly reuse objects from a library rather than writing the code yourself.
- The ***component*** level: components (collections of objects and object classes) are reused in application systems.
- The ***system*** level: entire application systems are reused.

Reuse: *Costs*

- The costs of the ***time*** spent in looking for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the costs of ***buying*** the reusable software. For large off-the-shelf systems, these costs can be very high.
- The costs of ***adapting and configuring*** the reusable software components or systems to reflect the requirements of the system that you are developing.
- The costs of ***integrating*** reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed.

Configuration management

Configuration management is the name given to the general process of ***managing a changing software system***. The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system.

Configuration management

Configuration management activities include:

- ***Version management***, where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers.
- ***System integration***, where support is provided to help developers define what versions of components are used to create each version of a system. This description is then used to build a system automatically by compiling and linking the required components.
- ***Problem tracking***, where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

Host-target development

Most software is developed on one computer (the host, ***development platform***), but runs on a separate machine (the target, ***execution platform***). A platform is more than just hardware; it includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment (IDE). Development platform usually has different installed software than execution platform; these platforms may have different architectures. Mobile app development (e.g. for Android) is a good example.

Host-target development

Typical development platform tools include:

- An integrated compiler and syntax-directed editing system that allows you to create, edit and compile code.
- A language debugging system.
- Graphical editing tools, such as tools to edit UML models.
- Testing tools, such as JUnit that can automatically run a set of tests on a new version of a program.
- Project support tools that help you organize the code for different development projects.

Open source development

Open source development is an approach to software development in which the ***source code*** of a software system is ***published*** and ***volunteers*** are invited to ***participate in the development process***.

Its roots are in the *Free Software Foundation*, which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish. Open source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code.

Open source development

The best-known open source product is, of course, the Linux operating system which is widely used as a server system and, increasingly, as a desktop environment. Other important open source products are Java, the Apache web server and the mySQL database management system.

Open source development

A ***fundamental principle*** of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code. Typical licensing models include:

- The GNU ***General Public License*** (GPL). This is a so-called 'reciprocal' license that means that if you use open source software that is licensed under the GPL license, then you must make that software open source.
- The GNU ***Lesser General Public License*** (LGPL) is a variant of the GPL license where you can write components that link to open source code without having to publish the source of these components.
- The ***Berkley Standard Distribution*** (BSD) License. This is a non-reciprocal license, which means you are not obliged to re-publish any changes or modifications made to open source code. You can include the code in proprietary systems that are sold.

Software Testing: The big picture

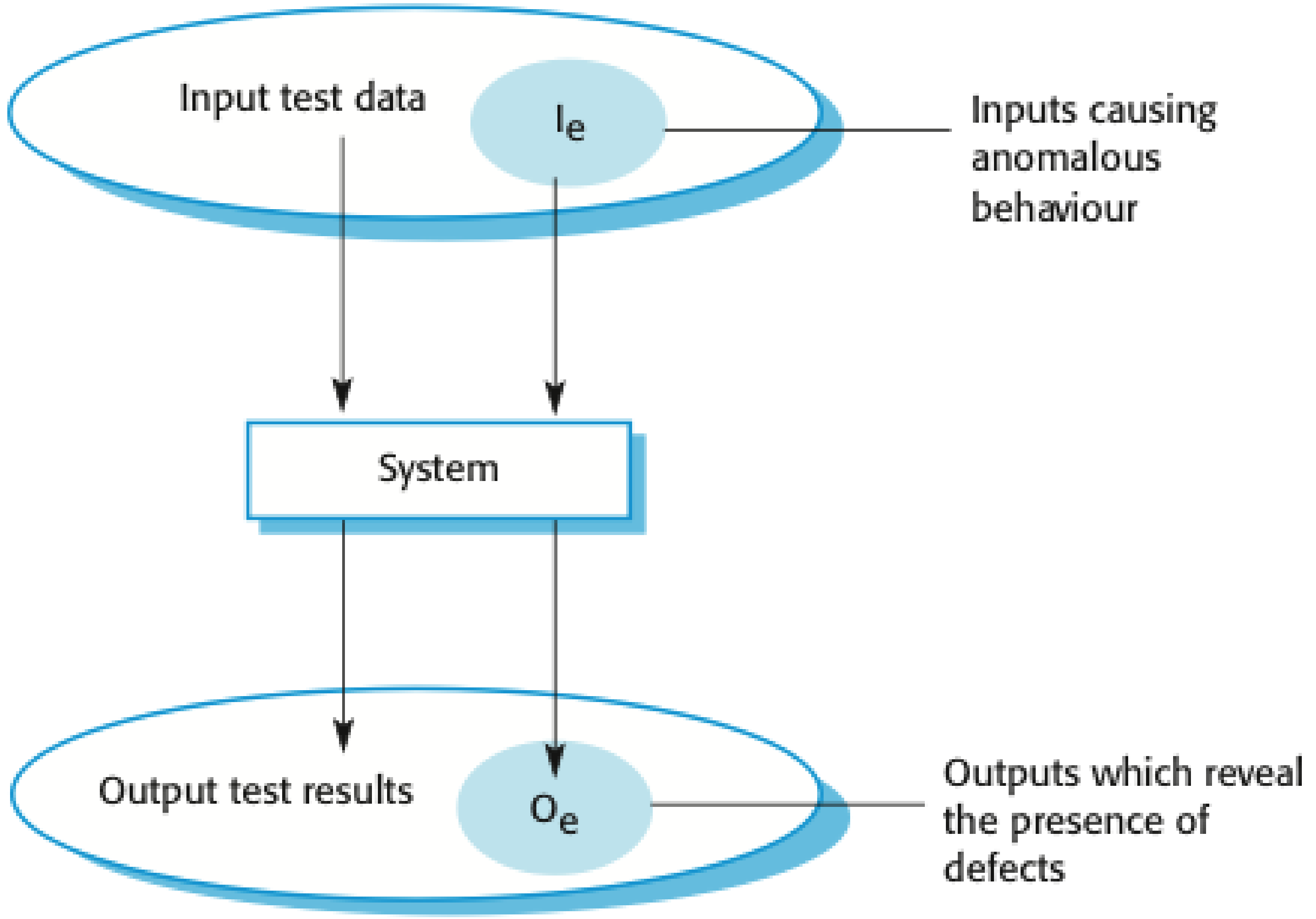
Testing is intended to show that a ***program does what it is intended to do*** and to ***discover program defects*** before it is put into use. When you test software, you execute a program using artificial data. You check the results of the test run for errors, anomalies or information about the program's non-functional attributes***. Testing can reveal the presence of errors, but NOT their absence***. Testing is part of a more general verification and validation process, which also includes static validation techniques.

Software Testing: *Goals*

- To demonstrate to the developer and the customer that the software meets its requirements.
- 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.
- A successful test shows that the system operates as intended.
- To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
- Leads to defect testing: the test cases are designed to expose defects; the test cases can be deliberately obscure and need not reflect how the system is normally used.
- A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

Software Testing:

Testing can be viewed as an input-output process:


Input test data
I_e
Inputs causing anomalous behaviour
System
Output test results
O_e
Outputs which reveal the presence of defects

Verification and validation

Testing is part of a broader process of software ***verification and validation*** (V & V).

- ***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.

Verification and validation

The ***goal of V & V*** is to establish confidence that the system is ***good enough for its intended use***, which depends on:

- ***Software purpose***: the level of confidence depends on how critical the software is to an organization.
- ***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 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.

Inspections and testing

Advantages of inspections include:

- 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.

Typically, a commercial software system has to go through ***three*** stages of testing:

- ***Development testing***: the system is tested during development to discover bugs and defects.
- ***Release testing***: a separate testing team test 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.

Development testing

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

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

Unit testing

When ***testing object classes***, tests should be designed to provide coverage of all of the features of the object:

- ***Test all operations*** associated with the object;
- Set and check the ***value of all attributes*** associated with the object;
- Put the object into ***all possible states***, i.e. simulate all events that cause a state change.

Unit 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.

An automated test has three parts:

- ***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 testing

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 ***two types of unit test cases***:

- 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.

Component testing

Software components are often composite components that are made up of ***several interacting objects***. 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. Objectives are to detect faults due to interface errors or invalid assumptions about interfaces. Interface types include:

- ***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.

Component testing

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 behavior 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.

Component testing

General *guidelines* for interface testing:

- 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 behavior*** of a system.

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.

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 ***document the components and their interactions*** that are being tested.

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. This is a differentiating feature of TDD versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code***. The code is developed 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.

Test-driven development

However, it can also be used in plan-driven development processes.

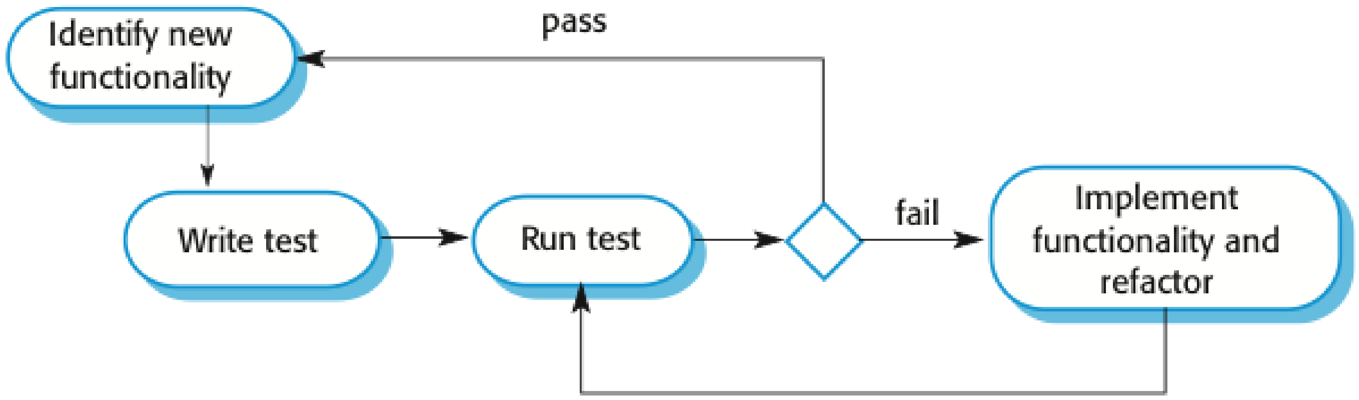
TDD example - *a string calculator*.

The ***goal of TDD*** isn't to ensure we write tests by writing them first, but t***o produce working software that achieves a targeted set of requirements using simple, maintainable solutions***. To achieve this goal, TDD provides strategies for ***keeping code working, simple, relevant, and free of duplication.***

Test-driven development

TDD process includes the following 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.


Identify new functionality
pass
Write test
Run test
fail
Implement functionality and refactor

Test-driven development

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 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 customer 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

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).

Release testing

Requirements-based testing involves examining each requirement and developing a test or tests for it. It is validation rather than defect testing: you are trying to demonstrate that the system has properly implemented its requirements.

Scenario testing is an approach to release testing where you devise typical scenarios of use and use these to develop test cases for the system. Scenarios should be realistic and real system users should be able to relate to them. If you have used scenarios as part of the requirements engineering process, then you may be able to reuse these as testing scenarios.

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 behavior.

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. Types of user testing include:

- ***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.

In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system. Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made. Main problem here is whether or not the embedded user is 'typical' and can represent the interests of all system stakeholders.

End of session

UNIVERSITY OF GHANA